МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)

Кафедра алгоритмов и технологий программирования

А.С. Хританков

ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

Учебно-методическое пособие

МОСКВА
МФТИ
2016

УДК 004.41+004.02+372.8

Рецензенты:
Кафедра системного программирования НИУ
Южно-Уральского государственного университета
Кандидат физико-математических наук *А.А. Ивахненко*

Принципы объектно-ориентированного проектирования : учеб.-метод. пособие / А.С. Хританков. – М.: МФТИ, 2016. – 32 с.

В данном издании подробно изложена тема принципов объектно-ориентированного проектирования, известных как SOLID, в расширение лекционного курса по проектированию программных систем, читаемого студентам Московского физико-технического института (государственного университета).

Предназначено для студентов, преподавателей высших учебных заведений и специалистов в проектировании программного обеспечения.

УДК 004.41+004.02+372.8

© Федеральное государственное автономное
образовательное учреждение высшего образования
«Московский физико-технический институт
(государственный университет)», 2016
© Хританков А.С., 2016

Содержание

1. Введение 5
  1.1. Зачем нужны принципы проектирования? 5
  1.2. Признаки плохого дизайна 5
2. Принципы SOLID 9
  2.1. Принцип ограничения ответственности 11
  2.2. Принцип открытости-закрытости 12
  2.3. Принцип подстановки и построение иерархий типов 16
  2.4. Принцип разделения интерфейса 19
  2.5. Принцип обращения зависимостей и каркасы 23
3. Вопросы для самопроверки 29
4. Заключение 30
5. Литература 31

Предисловие

Целью разработки данного учебно-методического пособия было расширить достаточно краткое изложение принципов проектирования SOLID, даваемое в лекционном курсе. А также взглянуть на эти принципы с точки зрения их взаимосвязей с другими принципами и направлениями в программной инженерии и computer science. Предпринята попытка прояснить связи между принципами и качеством программного обеспечения – идеей, которая лежит в основе базового курса по проектированию.

Пособие состоит из введения, основного раздела, вопросов для самопроверки и списка литературы. Во введении дается краткая характеристика рассматриваемых принципов и их связи с характеристиками качества программного обеспечения. В основном разделе по очереди рассматриваются принципы проектирования: ограничения ответственности, открытости-закрытости, подстановки, разделения интерфейса и обращения зависимостей. При рассмотрении принципов дается необходимая теоретическая справка и поясняющие примеры. Содержание пособия соответствует программе курса и образовательному стандарту.

Предназначено для преподавателей высших учебных заведений и студентов, обучающихся по специальности 010900 «Прикладные математика и физика», и специалистов в проектировании программного обеспечении.

1. Введение

1.1. Зачем нужны принципы проектирования?

В современном состоянии методология объектно-ориентированного программирования основывается на устоявшихся типовых решениях часто встречающихся проблем (паттернах), эскизных эталонных архитектурах и принципах проектирования, состав и порядок применения которых отличает один метод проектирования от другого.

Метод ООП определяет тактики (эвристики) проектирования, процедуры их применения и принципы, которым нужно следовать при принятии проектировочных решений. Некоторые методы используют для этого вспомогательные понятия, например, понятие обязанности в RDD [5].

Рассматриваемый набор принципов SOLID не может считаться отдельным методом проектирования, поэтому его обычно дополняют процедурой итеративной постепенной декомпозиции решения задачи либо процессом рефакторинга, в которых руководствуются этими принципами. Для принципов SOLID характерна попытка описать требования к конечной структуре системы, классам и связям между ними, для обеспечения такой нефункциональной характеристики качества, как сопровождаемость (maintainability).

Интересно, что данные принципы были собраны Робертом Мартином сначала в статьях для некогда популярного журнала «C++ Report[1]», выходившего в 90-х годах, а впоследствии вошли в состав его книги по объектно-ориентированному проектированию [1]. Многие статьи в итоге опубликованы и доступны онлайн[2].

1.2. Признаки плохого дизайна

Многие современные методики выполнения проектов по разработке программных систем, модели жизненного цикла таких проектов, основаны на итеративной или эволюционной процедуре разработки. При этом реализованные ранее в проекте части программной системы могут быть переделаны, изменены, заменены целиком исходя из изменившихся требований, полученных новых знаний в предметной области, поступившей обратной связи от заказчика или результатов тестирования текущей промежуточной версии программной системы. Применение таких, итератив-

---

[1] См. статью в Wikipedia, посвященную журналу, http://en.wikipedia.org/wiki/C++_Report

[2] www.objectmentor.com

ных, методов влечет необходимость поддержания достаточной сопровождаемости программной системы для успеха проекта в целом. Вне рамок отдельного проекта сопровождаемость важна, если предполагается выпуск нескольких версий программной системы, дорабатываемых по плану или в виде реакции на изменения среды или требований к системе в процессе ее эксплуатации. Кроме того, не стоит упускать из виду, что исправление выявленных дефектов, по сути, является внесением изменений в систему. Известны случаи, когда невозможность внесения изменений, доработки или исправления системы, приводили к необходимости повторной разработки с нуля системы целиком.

Вернемся к уточнению понятия «сопровождаемость». В текущей версии международного стандарта, определяющего характеристики качества программной системы, ISO/IEC 25010[3] (часть цикла стандартов SQuaRE), к подхарактеристикам сопровождаемости относятся:

- анализируемость (analysability) отражает простоту исследования и понимания внутренней структуры и функционирования программной системы;
- модульность (modularity) говорит, насколько система составлена из независимых частей (модулей) так, что изменения в одном модуле имеют минимальное влияние на другие;
- изменяемость (modifiability) показывает, насколько программная система может быть эффективно и продуктивно изменена без внесения дефектов или снижения других характеристик качества;
- перерабатываемость (reusability) указывает степень возможного повторного использования системы или ее частей при разработке новой системы;
- тестируемость (testability) отражает, насколько просто установить критерии качества тестирования по отношению к системе или ее части и насколько просто реализовать и выполнить тестовые процедуры по проверке этих критериев.

При разработке принципов SOLID были предложены следующие характеристики качества структуры программной системы, называемые «признаками плохого дизайна». Сравнение их с подхарактеристиками в стандарте приведено в табл. 1.

Неделимость (immobility) – невозможность выделить для повторного использования только нужную часть системы, без лишних зависимостей.

Пример. Пусть у вас есть структура модулей (рис. 1), вы хотите использовать вот этот модуль `PDFProc`, реализующий нужную вам функцию

[3] На май 2015 г. принята в качестве национальных стандартов в РФ лишь часть из цикла SQuaRE. Например, продолжает действовать более старый ГОСТ Р ИСО/МЭК 9126-93.

печати PDF. При попытке переноса выясняется, что для небольшого расчета размеров изображения модуль использует мощный математический пакет `WAlpha`. А этот пакет, помимо всего, использует старую библиотеку на Фортране `LAPAG`. Библиотека умеет в диалоговом режиме запрашивать у пользователя параметры вызываемых функций, а потому зависит от пакета `QuT`, переносимого с Windows на Unix, и еще одной библиотеки `gFortan`, которая выполняет трансляцию. Библиотека вывода на печать может дополнительно зависеть от модуля чтения файлов, и так далее. Получается, что вы не можете повторно использовать нужную вам часть программы.

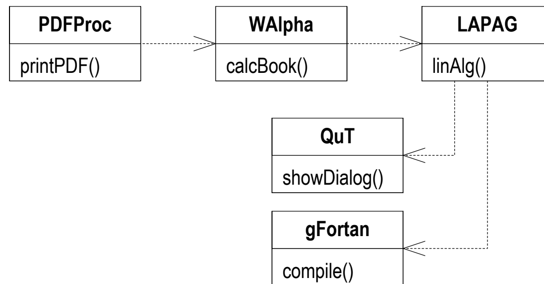

Рис. 1. Пример к пояснению характеристик дизайна

Жесткость (Rigidity) – это явная нелокальность даже небольших изменений. При необходимости внесения изменений в одну часть системы вы видите, что, исходя из ее структуры связей и зависимостей между частями, придется изменять еще несколько, на первый взгляд несвязанных, частей.

На все том же рис. 1 представьте зависимость от конкретной устаревшей версии компилятора, необходимой для сборки библиотеки LAPAG, в результате чего не получается перейти на новый стандарт языка программирования. Или необходимость использования для записи чисел в пользовательском интерфейсе в вашей программе плавающей точки вместо запятой, потому что библиотека QuT, например, некорректно отображает формат числа в противном случае.

Хрупкость (Fragility) – это скрытая нелокальность изменений. Похоже на жесткость, но при хрупком дизайне вы не знаете, что и где сломается, не знаете, как вносимое изменение отразится на системе.

Вязкость (viscosity) – это сложность следовать дизайну по сравнению с его нарушением. Например, Вы, как архитектор, считаете, что для расчета размеров страницы в `PDFProc` нужно использовать математический пакет `WAlpha`, так как в нем выполняется корректная обработка переполнения и реализованы адаптеры под выбранный язык программирования.

Но разработчикам проще, понятнее и удобнее вызывать эту функцию напрямую в `LAPAG`, так как в Интернет было найдено руководство, как это делать, и суммарные трудозатраты на реализацию данного варианта ниже. Если такое свойство в дизайне присутствует, то это и есть вязкость дизайна. Если такое отрицательное свойство в дизайне выражено, значит, предписания дизайна программной системы часто будут нарушать.

Таблица 1. Сравнение признаков плохого дизайна с характеристиками сопровождаемости

	Жесткость (Rigidity)	**Хрупкость (Fragility)**	**Неделимость (Immobility)**	**Вязкость (Viscosity)**
Анализируемость (Analyzability)		Хрупкую структуру сложно анализировать		Низкая анализируемость скорее всего ведет к высокой вязкости
Модульность (Modularity)	В модульной структуре жесткость невысокая	Модульная структура не может быть хрупкой		
Изменяемость (Changeability)	Жесткую структуру сложно изменять	Хрупкость препятствует изменениям		Высокая вязкость препятствует изменениям
Повторное использование (Reusability)			Неделимую структуру нельзя повторно использовать	Высокая вязкость препятствует повторному использованию
Тестируемость (Testability)		Хрупкость вследствие роста затрат снижает тестируемость	Неделимость обычно ведет к плохой тестируемости	

2. Принципы SOLID

Принципы проектирования классов и интерфейсов[4]. Аббревиатура SOLID расшифровывается по первым буквам сокращений названий принципов проектирования классов (см. рис. 2):

- *Single responsibility principle*, принцип ограничения ответственности, говорит, что модуль или класс должен иметь только один набор функционально сходных обязанностей;
- *Open-closed principle*, принцип открытости-закрытости, указывает, что класс или модуль должен быть закрыт для изменений, но открыт для расширения;
- *Liskov substitution principle*, принцип подстановки подтипа (описан в статье Барбары Лисков, отсюда название), дает правило построения иерархии типов так, что любой подтип или дочерний класс подставим вместо базового типа или класса;
- *Interface segregation principle*, принцип разделения интерфейса, говорит, что для обозначения разных ролей, которые играет класс в разных взаимодействиях, следует использовать разные интерфейсы;
- *Dependency inversion principle*, принцип обращения зависимостей, указывает на корректное применение принципа сокрытия информации в объектно-ориентированном подходе, когда зависимости направлены к выделенным абстракциям – описаниям типа.

Принципы проектирования структуры пакетов. Помимо указанных принципов, которые применяются при проектировании классов и интерфейсов, сформулированы принципы проектирования структуры пакетов[5]:

- *Reuse-release equivalence principle*, принцип соответствия повторного использования, указывает, что модуль, предназначенный для переработки, должен быть проведен через формальную процедуру выпуска с присвоением версии, тестированием и составлением документации;
- *Common closure principle*, принцип общего замыкания, говорит, что модули, которые при рассмотрении некоторого набора сценариев изменений изменяются вместе, следует размещать в одном пакете;

---

[4] Неплохой расширенный перевод материалов по SOLID с примерами на C++ и C# дается С. Тепляковым на его страничке (получено 07.2015) http://sergeyteplyakov.blogspot.ru/2014/10/about-design-principles.html

[5] Под *пакетом* понимается логический элемент модели программы, группирующий пакеты, модули или классы. Пакеты используются для управления группами элементов для снижения сложности модели.

– *Common reuse principle*, принцип общего повторного использования, указывает, что модули, которые предполагается повторно использовать совместно, следует поместить в один пакет;
– *Acyclic dependencies principle*, принцип ациклических зависимостей, говорит, что производные от зависимостей элементов пакетов зависимости самих пакетов не должны образовывать циклы;
– *Stable dependencies principle*, принцип стабильных зависимостей, говорит, что пакет (и модуль вообще) может зависеть только от пакетов (или модулей) с большим показателем стабильности, под которым понимается отношение входящих и исходящих зависимостей;
– *Stable abstractions principle*, принцип стабильности абстракций, указывает, что пакеты с высоким показателем абстрактности (то есть содержащие большую долю абстрактных классов или интерфейсов в качестве элементов), должны быть стабильными.

В последующих разделах мы подробно рассмотрим принципы проектирования классов и интерфейсов. Принципы построения структуры пакетов оставим для другого пособия.

Рис. 2. Принципы объектно-ориентированного проектирования классов и интерфейсов SOLID

2.1. Принцип ограничения ответственности

(S)ingle Responsibility Principle (SRP) – принцип одной ответственности или принцип сходства. Основой для принципа служит идея разделения сходства по нескольким степеням, следование этому принципу равнозначно поддержанию лучшего в структурном проектировании [6] функционального сходства среди элементов класса.

Принцип ограничения ответственности
Класс должен иметь только один набор функционально сходных обязанностей, направленных на решение одной задачи.

Следование данному принципу заключается в назначении классам реализуемой ими основной идеи, понятия предметной области, системной функции. Операции и атрибуты класса должны быть связаны между собой в реализации идеи класса. При развитии класса нужно отслеживать согласованность вносимых изменений с основной идеей класса.

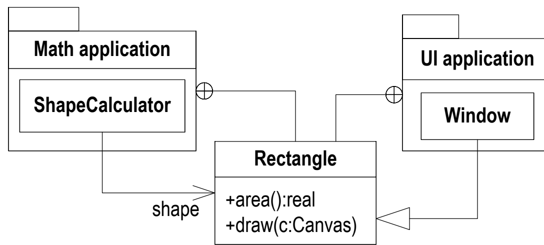

Рис. 3. Структура программы с нарушением принципа ограничения ответственности

Обнаружить нарушение принципа ограничения ответственности можно при проведении критических просмотров кода и дизайна, оценивая соответствие фактической реализации класса положенной в его основу идеи. Метрика LCOM4 (Lack of Cohesion between Methods[6]) также может быть использована для выявления классов, нарушающих SRP. Значение метрики равно количеству несвязанных между собой подмножеств элементов класса. Если таких несколько, полагается, что класс реализует несколько разных несвязанных идей.

---

[6] См. Hitz M., Montazeri B.: Measuring Coupling and Cohesion In Object-Oriented Systems. Proc. Int. Symposium on Applied Corporate Computing, Oct. 25-27, Monterrey, Mexico, 75-76, 197, 78-84. http://www.isys.uni-klu.ac.at/PDF/1995-0043-MHBM.pdf

Пример с фигурами. Пусть есть класс прямоугольник `Rectangle`, который умеет считать свою площадь и рисовать себя на экране. Его использует математическая библиотека `MathLib` для расчета площади прямоугольников и графический инструментарий (toolkit) `DrawTools` для отображения прямоугольников на экране. Допустим, мы хотим использовать математическую библиотеку в составе другого программного продукта `NewProduct`. Соответственно операция отображения на экране `Draw` будет не нужна, не понадобятся и все зависимости, требуемые для реализации этой функции. Второе. Если мы захотим доработать класс для математической библиотеки, то потребуется также отслеживать, чтобы класс продолжал работать и в составе графического инструментария (см. рис. 3).

Какое здесь решение? Из исходного класса прямоугольника `Rectangle` выделить прямоугольник, который умеет рисовать себя на экране `RectangleShape`, и геометрическую фигуру-прямоугольник `GeometricRectangle`, которая умеет рассчитывать свою площадь. В итоге получаем, что для расчета площади прямоугольника не нужно уметь его рисовать на экране (см. рис. 4).

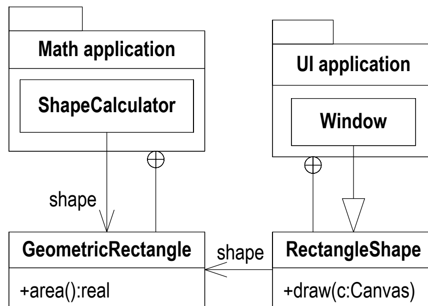

Рис. 4. Переработанная структура программы с выполнением принципа ограничения ответственности

2.2. Принцип открытости-закрытости

Следование Open-Closed Principle (OCP) или, по-другому, принципу открытости для расширения и закрытости для изменений позволяет выделять устойчивые абстракции, инкапсулировать их в классах и при этом обеспечивать возможность их уточнения под конкретные случаи по необходимости.

Обычно следование принципу усложняет дизайн за счет добавления механизмов расширения класса. Поэтому принцип применяют к классам с

большим числом входящих зависимостей, где стоимость изменения высока, либо моделирующим ключевые абстракции в системе, инкапсулирующие проектировочные решения [8].

Принцип открытости-закрытости

Класс (модуль) должен быть открыт для расширения, но закрыт для изменения.

Другими словами, у вас должна быть возможность изменять поведение класса, видимое для вызывающих его модулей, не изменяя исходный код класса. Один подход состоит в том, чтобы предусмотреть некоторые ожидаемые изменения в программной системе на этапе проектирования. Другой подход – это когда в системе нужно предусмотреть вариативность (изменчивость) выделенной абстракции в определенных местах. В отличие от первого подхода, варианты известны заранее.

Выделяют несколько базовых механизмов расширения поведения модуля или класса. Первый – это параметризация. Вместо использования конкретных значений, констант при реализации методов, используют атрибуты или параметры операций. Другой механизм – это шаблонизация, или параметризация класса одним или несколькими типами. Он используется, когда нужно реализовать одинаковое поведение класса для разных типов, имеющих, например, общего предка в иерархии наследования. Третий механизм связан с применением паттернов проектирования [3]. Для реализации принципа обычно используют паттерны шаблонный метод Template Method (через наследование), декоратор Decorator (делегирование), наблюдатель Observer и команда Command (обратный вызов, делегирование).

Классический пример с модемами [1]. Пусть у вас есть класс пользователя модема `Client`. Он связан с классом `Modem`, через который он звонит Интернет-провайдеру (рис. 5). Сейчас у абонента есть старый такой модем `Courier`. И пользователь через него звонит: вызывает операцию вызова `call`, а класс отправляет модему цепочку настроечных кодов, устанавливает соединение и выполняет прочие операции.

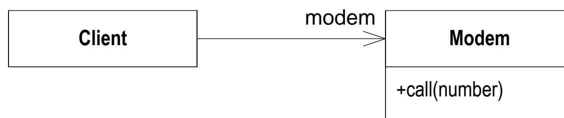

Рис. 5. Первоначальная реализация модема

К новому году пользователь хочет добавить еще одного провайдера и поставить модем Zyxel. Если при добавлении нового модема нам придется изменить класс Modem или метод call, то получим нарушение принципа закрытости-открытости: в каком-то месте класса придется добавить условный оператор по выбору поведения, в зависимости от типа модема. При добавлении еще одного модема нужно добавлять еще строки в условную конструкцию. Понятно, что мы получаем класс, который знает, как работать с модемами всех типов, как показано на рис. 6.

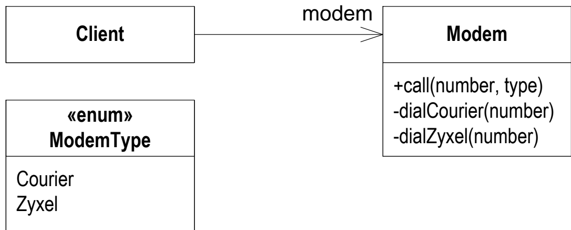

Рис. 6. Параметризация класса Modem типом модема, выделение специфичных типу модема операций

Как решить задачку, следуя OCP? У нас есть общее поведение, которое связано со всеми модемами. Поместим его в класс Modem. И это поведение мы расширяем или уточняем для каждого модема: Courier или Zyxel. Воспользуемся паттерном шаблонный метод Template Method. В классе Modem у нас может быть процедура call следующего вида.

Листинг 1. Деятельность набора номера call класса Modem

```
activity call(number:string) {
    // инициализация и проверка корректности номера
    <...>
    dial(number);
    <...>
    // завершающие действия по установлению соединения
}
```

Порядок и состав действий в реализации операции call не зависит от модема, но класс Modem не знает конкретно, какие действия выполнять при наборе номера dial. Поэтому эта операция определена без реализации и сам класс Modem абстрактный. Конкретные дочерние классы модемов Courier, Zyxel знают, как выполнять вызов dial. Они реализуют указанные абстрактные операции. Поэтому, если нужно добавить новый модем, например 3COM, нужно создать новый дочерний класс, в котором

реализовать только данную операцию (рис. 7). Клиент `Client` сможет работать с новым модемом без изменений. Этим мы выполняем принцип открытости-закрытости.

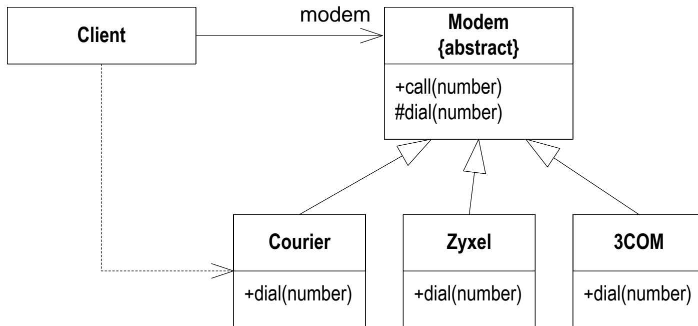

Рис. 7. Реализация модема с ожидаемым добавлением новых типов модемов с помощью паттерна шаблонный метод

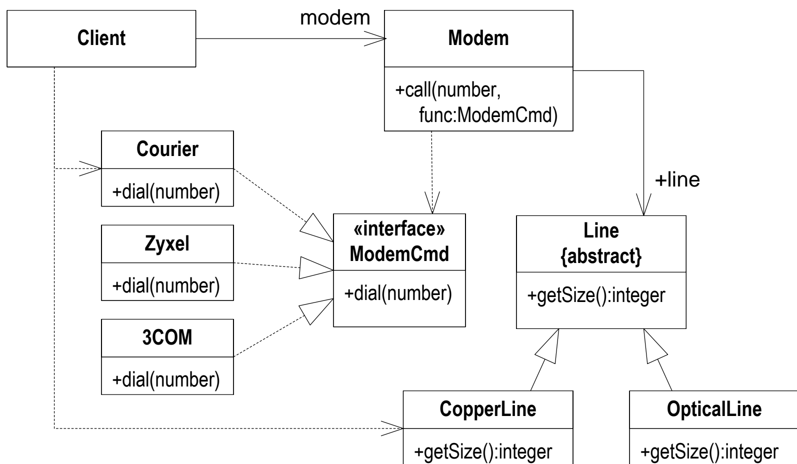

Рис. 8. Реализация модема с ожидаемым добавлением новых типов модемов с помощью паттернов команда Command и стратегия Strategy

В случае, когда возможно несколько расширений класса одновременно, например по виду линии и типу модема, рекомендуется использование паттернов команда Command или стратегия Strategy с делегированием специфичного поведения отдельным классам, как показано на рис. 8.

2.3. Принцип подстановки и построение иерархий типов

Современные объектно-ориентированные языки программирования и моделирования используют номинативную систему типов. Под **системой типов** понимается совокупность ограничений, накладываемых на использование значений и экземпляров объектов путем приписывания им именованных типов (номинативная система), и указания правил совместимости этих типов. Обычно языки определяют некоторый базовый набор типов, часто это примитивные типы, такие как числа, строки, и позволяют определять собственные типы для расширения возможностей языка. Распространенным механизмом определения новых типов является создание их как абстрактных типов данных – совокупности операций, ограничений на них и инвариантов для типа в целом [2]. В большинстве случаев объектно-ориентированные языки поддерживают создание структур данных или классов для описания новых типов, в том числе абстрактных типов данных (abstract data type, ADT).

Наследование было введено в объектно-ориентированном подходе как механизм порождения нового типа из базового для повторного использования. Выделяют два вида наследования – наследование реализации (частичной реализации) и наследование (уточнение) типа. Для наследования реализации создают подклассы, зависящие от реализации базового класса. При уточнении типа происходит уточнение описания абстрактного типа данных, при этом должен выполняться Liskov Substitution Principle (LSP) – принцип подстановки, следующий из определения этого типа наследования в работе [4]. С практической точки зрения он кратко формулируется следующим образом.

Принцип подстановки

Вместо экземпляра класса-предка можно подставить экземпляр класса-наследника без изменения видимого поведения любой программы, использующей класс-предок.

Или

Если какое-либо утверждение истинно для экземпляров класса-предка, то оно также должно быть истинно для экземпляров класса-потомка.

Тип A называется подтипом B, если выполняются условия:

- сигнатуры методов A согласованы по параметрам и возвращаемым значениям соответствующим методам B;
- абстракции пред-условий методов A являются следствием пред-условий соответствующих методов B и, наоборот, для пост-условий;
- инварианты типа B являются следствием аксиом типа A.

В языках программирования достаточно часто соответствие между методами подтипа и типа устанавливается по совпадению имен. В более общем случае для определения подтипа должны существовать функция абстракции, которая переводит используемые для определения A типы в те, что использованы для B, и отношение переименования методов A в методы B.

Заметим, что в UML [7] используемое для установления отношения тип–подтип отношение обобщения требует, чтобы класс-потомок был подставим вместо класс-предка. Если отношение в модели показано, но данные условия не выполняются, полагается, что существует ошибка в модели.

Выделяют следующие механизмы образования подтипов. Первый – это **расширение** типа путем добавления операций. Если среди добавляемых операций есть изменяющие состояние экземпляра (мутаторы), то эти изменения должны удовлетворять определенным в базовом типе инвариантам, либо, если подтип определяет собственное состояние, изменения этого состояния не должны влиять на выполнимость инвариантов базового типа. Второй механизм определения подтипа – это **ограничение**. Предположим, что базовый тип описан таким образом, что подразумевает наличие выбора внутреннего состояния. В этом случае подтип сужает возможные состояния, в которых могут находиться экземпляры.

Пример с эллипсом. Давайте рассмотрим классический пример [1] с эллипсом и окружностью для демонстрации введенного определения. Пусть у нас есть класс эллипс `Ellipse` с методами получения `getA():Integer`, `getB():Integer` и установки `setA(a:Integer)`, `setB(b:Integer)` длин осей `a` и `b`. Кроме того, в классе есть метод расчета площади эллипса `area():Real`. Предполагается, что устанавливаемые значения длин осей действительно сохраняются во внутреннем состоянии класса и могут быть получены позднее. Для конкретных значений проверку данной аксиомы класса можно записать в виде простого модульного теста[7] (листинг 2).

Теперь попробуем рассуждать следующим образом. Класс `Ellipse` определяет геометрические фигуры, которые при равных длинах осей будут окружностями. Введем класс `Circumference` как потомка `Ellipse` с ограничением в том, что допускаются только эллипсы с одинаковыми осями и расширение в том, что добавлены методы указания и получения радиуса окружности. Аналогично предполагаем, что записанное значение

[7] Понятно, что с помощью теста мы не сможем проверить, что аксиома выполняется для всех экземпляров класса в любом случае. Здесь тест нужен, чтобы обнаружить нарушение принципа подстановки.

радиуса методом `setR(r:Integer)` может быть получено позднее с помощью `getR():Integer`.

Листинг 2. Тесты для проверки сохранения значений осей класса `Ellipse` и корректности вычисления площади

```
activity saveAxesTest(e:Ellipse) {
    x = 1;
    y = 2;
    e.setA(x);
    e.setB(y);
    assertEquals(x, e.getA());
    assertEquals(y, e.getB());
}
activity areaTest(e:Ellipse) {
    x = 1;
    y = 2;
    e.setA(x);
    e.setB(y);
    assertEquals(PI*x*y, e.area());
}
```

Попробуем реализовать класс `Circumference`; есть следующие варианты:

1. добавляем атрибут `r` как часть внутреннего состояния `Circumference` и реализуем расчет площади, используя `r`;
2. изменяем реализацию `setA` и `setB`, чтобы поддерживать условие `a = b`, ожидаемое от окружности, реализацию расчета площади оставляем прежней;
3. при попытке установить значение `a`, отличное от установленного ранее значения `b` возбуждаем исключение, специфичное для класса `Circumference`;
4. изменим реализацию `area()` в `Circumference` и добавим проверку, для какого класса был вызван расчет площади, если для `Ellipse` – то вызываем расчет площади эллипса `super.area()`, иначе – считаем через радиус[8].

Реализация #1 не пройдет тест `areaTest`, будет вызван метод расчета площади по радиусу, хотя были установлены длины осей. Реализация №2 не пройдет тест `saveAxesTest`, так как не будет сохранено установленное значение оси `a`. Реализация №3 отличается от предыдущей тем, что будет явно указано, что нарушено ограничение, добавленное в дочернем классе `Circumference`. Реализация №4 – это исправленный первый вариант, он

[8] Вариант был предложен студентами на лекции в 2013 году.

не удовлетворяет принципу ограничения ответственности: в одном классе Circumference мы объединяем внутреннее состояния и методы расчета как эллипса, так и окружности.

Листинг 3. Тест для проверки сохранения значения радиуса класса Circumference и корректности вычисления площади

```
activity saveRadiusTest(e:Circumference) {
    r = 1;
    e.setR(r);
    assertEquals(r, e.getR());
}
activity areaTest(e:Circumference) {
    r = 1;
    e.setR(r);
    assertEquals(PI*r*r, e.area());
}
```

В чем причины сложностей в реализации? Дело в том, что описание класса Circumference как подтипа нарушает исходно заданные тестами предположения (инварианты) для класса Ellipse, кроме вырожденного случая, когда поведение дочернего класса не связано с поведением базового.

На практике инварианты для классов в явном виде выписываются редко, и обнаружение несоответствий принципу подстановки сводится к проверке прохождения классом-потомком тестов для базового класса.

Продолжим анализ вариантов. Решение №3 будет корректным нарушением принципа подстановки, так как исключения указывают на нарушение предположений, сделанных при проектировании класса. Вариант №4, по сути, компенсирует используемое в примере для классов подразумеваемое «виртуальное наследование» методов. В некоторых языках программирования можно отказаться от виртуального наследования явно для отдельных методов или использовать только для обозначенных (например, ключевым словом virtual).

Другим решением может быть ослабление требований (инвариантов) к классу Ellipse, отказ от наследования и применение другого механизма расширения классов с повторным использованием – применение паттерна декоратор Decorator [3].

2.4. Принцип разделения интерфейса

Под интерфейсом в проектировании программного обеспечения понимают общую для клиента и поставщика границу, через которую происходит

их взаимодействие. Поставщик реализует интерфейс, клиент его использует.

В объектно-ориентированном подходе интерфейсом называют модуль системы, который описывает контракт роли в некотором взаимодействии. Другими словами – это набор обязанностей, которые берет на себя поставщик интерфейса при совместной работе в определенном контексте взаимодействия с другими модулями. При этом следует отличать интерфейс от «интерфейса класса» или описания типа (см. раздел 2.3), который объединяет внешне видимые черты класса, такие как методы или атрибуты. Далее под интерфейсом мы будем понимать именно набор обязанностей в описании роли.

Элементами интерфейса могут быть операции, через которые происходит взаимодействие, объявления о приеме сигналов [7], атрибуты и свойства, а также инварианты и описания протокола взаимодействия (см. Protocol State Machine, [7]). В отсутствие реализации оценить сходство элементов интерфейса можно по тому, каким образом они используются клиентами во взаимодействии. Высоким уровнем сходства будет обладать интерфейс, все элементы которого используются для описания одной роли. Если часть элементов не используется, то интерфейс избыточен, если же для описания роли требуется несколько интерфейсов, то говорят о недостаточности интерфейса.

Interface Segregation Principle (ISP) – принцип разделения интерфейсов, применение которого преследует цель сохранения высокого уровня функционального сходства элементов каждого из них. Принцип формулируется следующим образом.

Принцип разделения интерфейсов
Если класс участвует в разных взаимодействиях в разных ролях, то для каждого из них следует выделить отдельный интерфейс.
Или
Каждому типу клиента следует предоставлять специализированный под его задачи интерфейс.

Пример с паттерном Facade. Пусть в нашей системе есть классы хранения наличности `CashBox`, чтения карты `CardReader`, проведения операций `PayService` и самой карты `Card`. У нашей системы два вида пользователей – клиенты `Client` и оператор `Operator`. Пусть мы применили паттерн проектирования Facade с тем, чтобы скрыть внутреннюю структуру классов от пользователя. Мы создали для этого класс `Console` и поместили в него необходимые пользователям операции получения наличности `getCash`, вноса денег `addCash`, оплаты по счету `payBill` и ввода пин-кода `inputPin` для авторизации (рис. 10). Так вот, принцип

разделения интерфейса гласит, что каждому виду клиента `Client` и `Operator` нужно предоставить отдельные интерфейсы `IClient` и `IOperator` только с требуемыми наборами операций: `getCash`, `payBill` и `inputPin`; `addCash` и `inputPin`. Оба этих интерфейса реализует наш класс `Console`, как показано на рис. 9.

Почему предпочтительнее предоставить каждому клиенту свой интерфейс? Давайте разберемся.

- Появляется возможность управлять направлением зависимостей между клиентами и системой, можно поставлять интерфейсы вместе с клиентами, а можно вместе с системой;
- при необходимости внесения изменений только для одного из клиентов другой не будет затронут;
- повышается сходство элементов каждого из интерфейсов.

В качестве примера рассмотрим, каким образом можно сохранить полную обратную совместимость для клиентов `IClient` при добавлении новой операции `check` оператору и изменении поведения операции `inputPin`.

Так как изменения вносятся в интерфейс `IOperator` без изменения внутренней структуры системы, то предпочтительнее воспользоваться паттерном Adapter для новой версии интерфейса. Добавляем новую реализацию проверки пин-кода `inputMasterPin`, в адаптере перенаправляем вызов `inputPin` в `inputMasterPin` и добавляем оставшиеся операции. Заметим, что при этом не была затронута реализация операций интерфейса `IClient`. Таким образом, мы получаем возможность дорабатывать наши интерфейсы с разными клиентами независимо друг от друга.

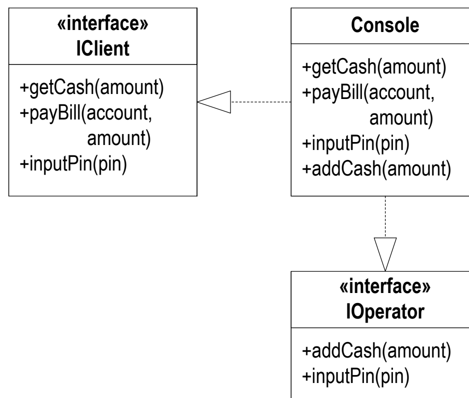

Рис. 9. Пример выделения отдельного интерфейса для каждого клиента

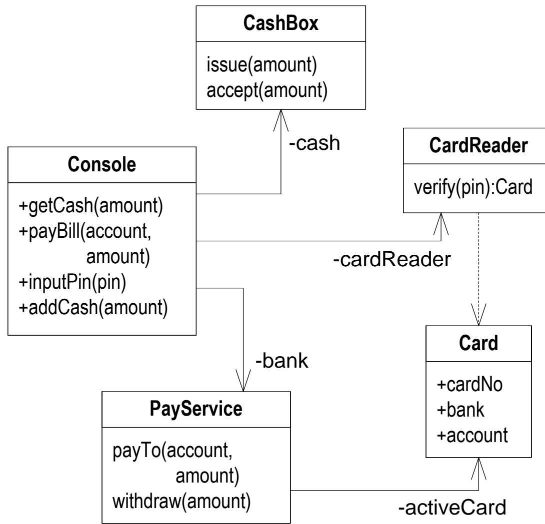

Рис. 10. Фрагмент диаграммы классов. Применен паттерн Facade для Console

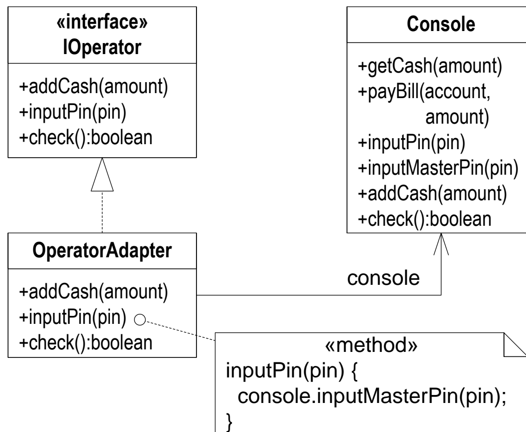

Рис. 11. Решение после изменения интерфейса IOperator с применением паттерна Adapter

2.5. Принцип обращения зависимостей и каркасы

Давайте начнем с небольшого примера. Рассмотрим программу, которая читает несколько байт из файла, сортирует их и записывает обратно. В зависимости от реализации, отсортированные байты могут быть записаны с начала или дописаны в конец.

Проектируя программу, например, методом постепенного уточнения, мы могли бы получить структурную схему, приведенную на рис. 12. На ней показаны модули и вызовы между ними. Напомним, что в процедурном подходе вызываемые процедуры и функции являются модулями. В нашем случае модули созданы для выполнения шагов конкретного алгоритма работы программы. В этом случае модуль `main` будет зависеть, как показано на рис. 13, от того, как реализованы остальные модули, при этом он сам описывает работу программы в целом, а остальные модули – шаги алгоритма. Алгоритм работы программы в целом зависит от реализации шагов алгоритма отдельными модулями.

Так вот, принцип обращения зависимостей гласит об обратном.

Принцип обращения зависимостей

Модуль не должен зависеть от реализации других модулей, которые он использует (Information hiding) [8].

Или

Абстракции не должны зависеть от деталей реализации. Детали реализации должны зависеть от используемых абстракций.

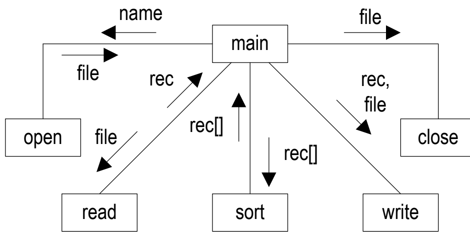

Рис. 12. Структурная схема программы сортировки файла при процедурном подходе

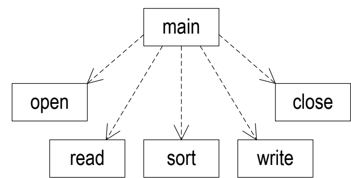

Рис. 13. Диаграмма зависимостей между модулями при процедурном подходе

Давайте разберемся, о каких зависимостях идет речь и что понимается под абстракциями. В том же UML под **зависимостью** понимается отношение между элементами модели, при котором для того, чтобы зависимый элемент имел смысл в модели, был полностью определен, необходи-

мо, чтобы тот элемент, от которого он зависит, также был полностью определен в модели.

В применении к ООП, помимо других отношений, выделяют следующие виды прямых зависимостей[9]:

- вызова, когда реализация метода одного класса вызывает методы другого,
- использования, когда другой класс используется для указания типа параметра метода или атрибута класса,
- создания и уничтожения, когда один класс создает или уничтожает экземпляры другого.

Под **абстракцией** в данном случае понимается независимое от реализации описание внешне видимых свойств и поведения класса (формальное описание типа). Распространенные современные объектно-ориентированные языки не требуют определения абстракций, позволяют разрабатывать классы, не прибегая к ним[10]. Если вернуться к обсуждению принципа подстановки, то определение класса в ADT будет примером абстракции.

В большинстве случаев на практике для отделения абстракции класса от его реализации прибегают к созданию интерфейсов или абстрактных классов. Полноценными абстракциями они будут только в случае описания инвариантов и протоколов вызова методов или обмена сообщениями, что не всегда выполняется. Тем не менее, даже незаконченные абстракции оказываются полезными.

Интерфейсы создают для обозначения роли во взаимодействии в некотором контексте. **Абстрактные классы** используют для обозначения не реализуемых в системе, но полезных для ее моделирования, абстракций.

Заметим, что не только в ООП используются абстракции модулей, отделяемые от их реализации. На рис. 14 показана схема программы, также разработанной в процедурном стиле, но для каждого модуля выделены интерфейсы – полноценные сигнатуры процедур, пред- и постусловия (не показаны на схеме). Для описания интерфейсов использованы типы данных, относящиеся к языку программирования, платформе реализации, но не к конкретной реализации модулей. Это важно, иначе получим зависимость интерфейсов от реализации, что противоречит принципу обращения зависимостей. Если для определения интерфейсов нужны сложные типы

---

[9] Бывают еще *косвенные*, например, взаимодействия, когда один класс ожидает получения сигнала или вызова метода, который осуществляет другой класс.

[10] Речь идет о программировании по контракту (programming-by-contract). Некоторые примеры языков, которые позволяют описывать абстракции средствами языка: Eiffel, а также D. Для распространенных языков существуют расширения с поддержкой контрактов.

данных, они также должны входить в описание интерфейса и быть частью абстракции модуля.

В качестве примера, рассмотрим, как работают с абстракциями в процедурном языке C. Считается хорошей практикой выносить сигнатуры реализуемых модулем подпрограмм в отдельный заголовочный файл, который затем используется в вызывающей программе и, что важно, в файле исходного кода для реализации объявленных подпрограмм. Дополнение этих сигнатур неформальным текстовым описанием инвариантов дает некоторую абстракцию. Получаем, что каждый из модулей может быть скомпилирован отдельно от других, так как реализации модулей не зависят друг от друга, но зависят от абстракций в заголовочных файлах, как показано на рис. 15. Получаем «обращение зависимостей» по сравнению с рис. 13.

Что произойдет, если мы внесем изменения в какой-либо листовой модуль `open` в обоих случаях. Если изменяем реализацию подпрограммы, и она соответствует интерфейсу модуля, то нет необходимости проверять работоспособность остальных частей программы, так как они не зависят от реализации модуля, а только от его интерфейса. Если изменяется интерфейс листового модуля, то нужно проверить все модули, которые используют этот интерфейс. Ранее, любые изменения в `open` потенциально могут привести к изменениям в `main`, поэтому требуют проверки. Если несколько модулей используют `open`, то понадобится проверить каждый из них и все, от них зависящие. В целом следование принципу обращения зависимости позволяет ограничить распространение изменений в системе.

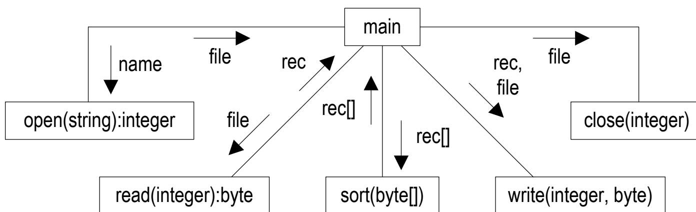

Рис. 14. Структурная схема программы сортировки файла при процедурном подходе, выделены интерфейсы модулей

Вернемся к объектно-ориентированному подходу. Модулем, в отличие от процедурного, будут не отдельные подпрограммы, а классы. Для выделения абстракции класса в отдельный модуль используют, как упоминалось выше, интерфейсы и абстрактные классы. Выделим абстрактный класс `File` для работы с файлами и интерфейс сортировщика `Sorter`. Структурная схема для получившейся объектно-

ориентированной программы представлена на рис. 16 и больше напоминает диаграмму коммуникаций UML[11].

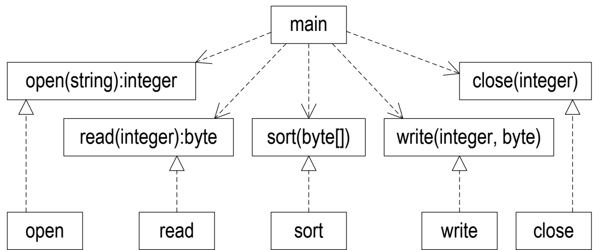

Рис. 15. Диаграмма зависимостей с учетом выделенных интерфейсов

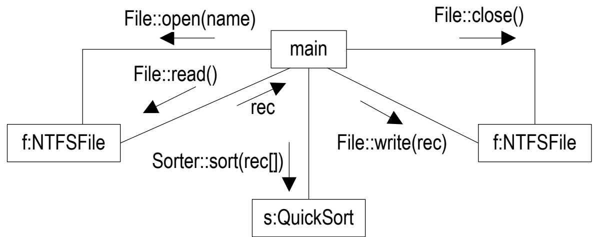

Рис. 16. Структурная схема программы сортировки файла при объектно-ориентированном подходе, выделены абстрактный класс File и интерфейс Sorter

Абстракции `File` и `Sorter` зависят от определяемых ими операций, методы, реализующие эти операции, зависят от класса, в которых они определены, и, транзитивно, от самих операций. Снова получаем обращение зависимостей по сравнению с рис. 13, но схема более сложная, чем для процедурного решения. Что это дает?

Возможность изменить используемую реализацию абстракции, интерфейса или класса, в процессе выполнения программы – **динамический полиморфизм**. В процедурных языках также присутствует возможность выбора реализации, но статически на этапе сборки программы[12].

---

[11] Можно считать достаточно близким аналогом структурной схемы. Взаимодействие, описываемое на диаграмме коммуникаций, можно показать и на диаграмме последовательности.

[12] Существует также поддержка динамически загружаемых модулей на уровне операционных систем, но механизм не является частью языка.

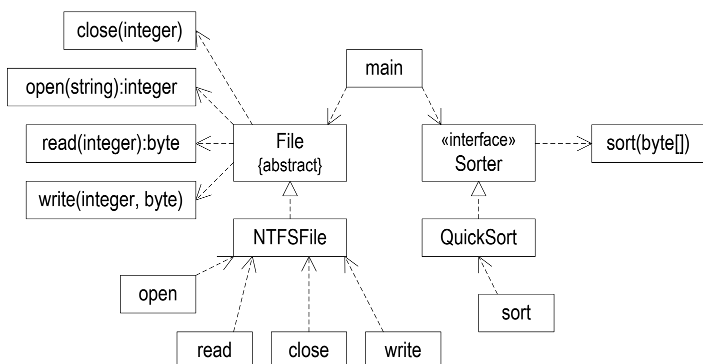

Рис. 17. Диаграмма зависимостей при объектно-ориентированном подходе с учетом выделенных абстракций

Принцип обращения зависимостей – это очень важный принцип. Многие, кто впервые с ним встретился, принимает его за откровение и пытается использовать при любом удобном случае. Наиболее распространенное следствие этого принципа – это каркасы разработки (frameworks), выполняющие **внедрение зависимостей** (dependency injection, **DI**) для связывания экземпляров классов приложения между собой и с библиотеками.

Вы давно последний раз устраивались на работу в крупную компанию в Голливуде? Попробуйте. Проверите на себе **принцип обращения управления** (Inversion-of-Control, IoC) или «Голливудский принцип» (не звоните нам, мы сами вам перезвоним)[13].

Идея в следующем. У нас есть класс, ему для работы нужен, например, ресурс, доступ к базе данных, стороннему удаленному сервису и так далее. Обычно ресурс представлен каким-либо интерфейсом. Первый вариант – класс обращается к другому «менеджеру ресурсов», второй вариант – класс заявляет о потребности в ресурсе каким-то ранее определенным образом, каркас рассматривает потребность в ресурсе и назначает его классу. В первом случае ваш класс будет зависеть от используемого каркасом менеджера ресурсов. При обновлении каркаса или переходе на другой каркас потребуется внесение изменений в наш класс. Во втором варианте этого не требуется. И класс, и каркас зависят от общего механизма обозначения зависимостей от ресурсов. Этот механизм обычно фиксиру-

[13] По-видимому, впервые сформулирован в статье Richard E. Sweet "The Mesa Programming Environment", SigPLAN Notices 20(7):216-229, July 1985. Заметим, что работа написана в Xerox PARC.

ют в виде индустриального стандарта[14]: он и будет абстракцией, требуемой принципом обращения зависимости.

Каркасы разработки, контейнеры IoC – это в большинстве случаев готовые библиотеки (commercial off-the shelf, COTS), используемые в программе. Контроля над этими библиотеками у вас нет, в первом случае любое значимое изменение потребует перепроверки работы вашей программы, а во втором – нет.

Значение каркасов для проектирования заключается в том, что наряду с паттернами они являются одним из основных средств повторного использования проектировочных решений в современной практике разработки.

Примеры каркасов. Ruby-on-Rails, ASP.NET, Autofac, Spring framework, Guice, Baratine и другие. Все эти каркасы реализуют примерно одинаковые системные механизмы, общие для серверных приложений, – работу с ресурсами, управление многозадачностью, перехват и обработка ошибок, механизмы объединения компонентов в готовую систему.

[14] См. например, Java Enterprise Edition, или, де-факто, Spring Framework.

3. Вопросы для самопроверки

1. Зачем нужны принципы проектирования, как они используются в процессе разработки программных систем?
2. Какие характеристики качества объектно-ориентированного дизайна вы знаете? В чем они заключаются?
3. В чем суть принципа ограничения ответственности? Приведите по одному примеру дизайна, который следует и который нарушает данный принцип.
4. Как вы понимаете, в чем может состоять «обязанность» класса или модуля?
5. В чем заключается принцип открытости-закрытости? Приведите по одному примеру, когда он нарушается и когда выполняется.
6. Какие паттерны проектирования могут быть использованы для реализации принципа?
7. Сформулируйте принцип подстановки, в чем состоит правило иерархии типов?
8. Поясните разницу между определением типа и реализацией класса.
9. Помимо наследования, какие существуют механизмы повторного использования классов, при которых есть возможность изменять их поведение? Какие паттерны проектирования могут быть применены?
10. В чем заключается принцип разделения интерфейса? Приведите примеры, когда он выполняется и когда не выполняется.
11. Что такое сходство элементов класса или модуля? Как оценивается сходство для интерфейсов?
12. Каким образом применение принципа разделения интерфейса способствует лучшей сопровождаемости системы?
13. Сформулируйте принцип обращения зависимостей, поясните, почему он так называется.
14. В чем различие между реализацией принципа средствами структурного (процедурного) программирования и объектно-ориентированного?
15. Каким образом принцип обращения зависимостей используется при разработке современного программного обеспечения?

4. Заключение

Мы рассмотрели принципы объектно-ориентированного проектирования, разработки структуры программных продуктов, которые в современной практике проектирования принято обозначать аббревиатурой SOLID по первым буквам названий принципов.

Следование данным принцип при проектировании и разработке способствует повышению сопровождаемости программной системы, что важно и необходимо, если предполагается длительный срок ее использования и развития, если в создании системы принимает участие несколько разработчиков.

В проектировании принципы служат основой для построения методов, а также используются отдельно при сравнении альтернативных вариантов проекта системы и выработке новых альтернатив, выборе подходящего варианта проекта для реализации.

Для дальнейшего, более глубокого, изучения принципов проектирования рекомендуется обратиться к исследованиям по верификации принципов при помощи метрик на реальных проектах разработки программных систем. В дополнение к рассмотренным принципам для обеспечения сопровождаемости применяются принципы установления зависимостей между пакетами, компонентами и модулями, подробнее о них можно узнать в книге [1].

ЛИТЕРАТУРА

1. Martin R.C. Agile Software Development, Principles, Patterns, and Practices. – Pearson, 2002. – 529 p.
2. Мейер Б. Объектно-ориентированное конструирование программных систем + приложение. – М.: Русская редакция, 2005. – 1232 с.
3. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. – СПб.: Питер, 2015. – 368 с.
4. Liskov B., Wing J.M. A Behavioral Notion of Subtyping. ACM Trans. Program. Lang. Syst. 16(6): 1811-1841 (1994).
5. Wirfs-Brock R., McKean A. Object Design: Roles, Responsibilities, and Collaborations. – Addison-Wesley, 2003.
6. Budgen D. Software Design. 2nd Ed. – Pearson, 2003. – 468 p.
7. Буч Г., Якобсон А., Рамбо Дж. UML. Классика CS. 2-е издание / пер. с англ. – СПб: Питер, 2006. – 736 с.
8. Parnas D.L. On a criteria to be used when decomposing system into modules // Communications of the ACM, V. 15, 1972. – pp. 1053–1058.

Учебное издание

Хританков Антон Сергеевич

ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

Учебно-методическое пособие

Редактор *Л.В. Себова.* Корректор *В.А. Дружинина*

Подписано в печать 12.07.2016. Формат 60 × 84 $^{1}/_{16}$. Усл. печ. л. 2,0
Уч.-изд. л. 1,9. Тираж 100 экз. Заказ № 274

Федеральное государственное автономное образовательное учреждение высшего образования «Московский физико-технический институт (государственный университет)», 141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-58-22, e-mail: rio@mipt.ru

Отдел оперативной полиграфии «Физтех-полиграф»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-84-30, e-mail: polygraph@mipt.ru