Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Шаблоны проектирования — элементы многократно используемого объектно-ориентированного программного обеспечения.

Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес

patterns@narod.ru

Глава 1

Введение

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Проектировать объектно-ориентированного (ОО) программного обеспечения (ПО) трудно, а проектировать *многократно используемое* ОО ПО еще труднее. Вам нужно найти подходящие объекты, объединить их в классы правильного размера, определить интерфейсы классов и иерархию наследования, установить ключевые взаимосвязи между ними. Ваш проект должен быть специфичным для решаемой задачи, но также достаточно общим, чтобы удовлетворять будущим требованиям и задачам. Вам также хочется избежать перепроектирования, или хотя бы минимизировать его. Опытные ОО проектировщики скажут вам, что многократно используемый и гибкий проект трудно, если не невозможно разработать "правильно" с первого раза. До окончания проекта, они обычно пытаются использовать его несколько раз, каждый раз изменяя.

Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it?

Все же опытные ОО проектировщики делают хорошие проекты. В то же время начинающие проектировщики перегружены доступными возможностями и имеют тенденцию откатываться к не-ОО техникам, которые они использовали раньше. Новичкам требуется много времени, чтобы понять, что же такое хороший ОО проект. Опытные проектировщики, очевидно, знают что-то неизвестное неопытным. Что же это?

One thing expert designers know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

Одна из вещей которую эксперты знают *не* делать — это решение каждой задачи с нуля. Наоборот, они используют решения, которые работали в прошлом. Когда они находят хорошее решение, они используют его снова и снова. Такой опыт — часть того, что делает их экспертами. Следовательно, вы найдете повторяющиеся шаблоны классов и сообщающиеся объекты во многих ОО системах. Эти шаблоны решают специфические проектные проблемы и делают ОО проекты более гибкими, элегантными и чрезвычайно подходящими для многократного использования. Они помогают проектировщикам использовать успешные проекты, основывая новые проекты на предыдущем опыте. Проектировщик, который знаком с такими шаблонами, может сразу же использовать их в текущих задачах без необходимости переоткрывать их.

An analogy will help illustrate the point. Novelists and playwrights rarely design their plots from scratch. Instead, they follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels). In the same way, object-oriented designers follow patterns like "represent states with objects" and "decorate objects so you can easily add/remove features." Once you know the pattern, a lot of design decisions follow automatically.

Эту идею можно проиллюстрировать с помощью аналогии. Писатели и драматурги редко разрабатывают свои сюжеты с нуля. Вместо этого они следуют шаблонам, таким как "Трагический герой" (Макбет, Гамлет и т.д.) или "Романтическая история" (бесчисленные романтические повести). Аналогично ОО проектировщики следуют таким образцам как "представление состояний с помощью объектов" и "декорация объектов для упроще-

ния добавления/удаления свойств". Если вы знаете шаблон, множество решений следуют автоматически.

We all know the value of design experience. How many times have you had design *deja-vu* – that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it. However, we don't do a good job of recording experience in software design for others to use.

Все мы знаем цену проектному опыту. Как много раз у вас было проектное *дежавю* — такое чувство, что вы уже решали подобную задачу раньше, но не помните точно где или как? Если бы вы могли запомнить детали предыдущей задачи и как вы ее решали, то вы смогли бы использовать этот опыт вместо перепридумывания его. Тем не менее, мы не записываем свой опыт в проектировании ПО для того, чтобы его могли использовать другие.

The purpose of this book is to record experience in designing object-oriented software as **design patterns**. Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively. To this end we have documented some of the most important design patterns and present them as a catalog.

Цель этой книги записать опыт в разработке ОО ПО в виде **шаблонов проектирования.** Каждый шаблон систематизировано именует, объясняет и оценивает важное и повторяющееся проектное решение в ОО системах. Наша цель зафиксировать проектный опыт в такой форме, чтобы люди могли использовать его эффективно. Для этого мы документировали некоторые из самых важных проектных шаблонов и представили их в виде каталога.

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster.

Шаблоны проектирования упрощают повторное использование успешных проектных и архитектурных решений. Выражение проверенных техник в виде шаблонов проектирования делает их более доступными для разработчиков новых систем. Шаблоны проектирования помогают вам выбрать проектные альтернативы, которые позволяют многократно

использовать систему и избежать альтернатив, которые этого не позволяют. Шаблоны проектирования могут даже улучшить документацию и сопровождение существующих систем, предоставляя явную спецификацию взаимодействий классов, объектов и их назначения. Проще говоря, шаблоны проектирования помогают сделать проект "правильно" быстрее.

None of the design patterns in this book describes new or unproven designs. We have included only designs that have been applied more than once in different systems. Most of these designs have never been documented before. They are either part of the folklore of the object-oriented community or are elements of some successful object-oriented systems – neither of which is easy for novice designers to learn from. So although these designs aren't new, we capture them in a new and accessible way: as a catalog of design patterns having a consistent format.

Ни один из проектных шаблонов в этой книге не описывает новое или непроверенное проектное решение. Мы включили только решения, которые использовались более одного раза в разных системах. Большинство из них нигде раньше не документировались. Они являются либо частью фольклора ОО сообщества, либо элементами некоторых успешных ОО систем, и то и другое сложно изучить неопытным проектировщикам. Таким образом, хотя эти проектные шаблоны не новы, мы зафиксировали их новым и доступным способом: в виде каталога проектных шаблонов имеющих последовательный формат.

Despite the book's size, the design patterns in it capture only a fraction of what an expert might know. It doesn't have any patterns dealing with concurrency or distributed programming or real-time programming. It doesn't have any application domain-specific patterns. It doesn't tell you how to build user interfaces, how to write device drivers, or how to use an object-oriented database. Each of these areas has its own patterns, and it would be worthwhile for someone to catalog those too.

Несмотря на размер книги, шаблоны проектирования охватывают только часть того, что может знать эксперт. В ней нет шаблонов для параллельного или распределенного программирования, программирования систем реального времени. В ней нет никаких шаблонов для приложений специфических для какой- либо предметной области. Она не рассказывает, как строить интерфейс пользователя, как писать драйверы устройств или как использовать ОО базы данных. Каждая из этих областей имеет свои собственные шаблоны, и их каталогизирование будет стоящим занятием для кого-либо другого.

1.1 What is a Design Pattern?
Что такое шаблон проектирования?

Christopher Alexander says, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” [AIS+77, page x]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context.

Кристофер Александер говорит: “Каждый шаблон сначала описывает проблему, которая случается снова и снова в нашем окружении и затем описывает суть решения этой проблемы, таким образом, что вы можете использовать ее снова миллион раз, не повторяя один и тот же путь дважды”. Даже хотя Александер говорит о шаблонах в строениях и городах, его слова истинны и для ОО проектных шаблонов. Наши решения выражены в терминах объектов и интерфейсов вместо стен и дверей, но суть обоих видов шаблонов состоит в решении задачи в определенном контексте.

In general, a pattern has four essential elements:

В общем шаблон состоит из 4 существенных элементов.

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.

   Название шаблона — это то, что мы можем использовать для описания проектной задачи, ее решений и последствий в одном двух словах. Именование шаблона сразу же увеличивает наш проектный словарь. Оно позволяет нам проектировать на более высоком уровне абстракции. Наличие словаря для шаблонов позволяет нам говорить о них с нашими коллегами, в нашей документации и даже с самими собой. Это облегчает обдумывание проектов и сообщение другим о них и их оптимальных вариантах. Поиск хороших имен был одной из сложнейших задач при разработке нашего каталога.

2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

   Задача описывает когда применять шаблон. Она объясняет задачу и ее контекст. Она может описывать специфические проектные проблемы, например как представлять алгоритмы в виде объектов. Она может описывать структуры классов или объектов являющихся симптомами негибкого проекта. Иногда задача включает список условий, которые должны выполняться для того, чтобы имело смысл применить шаблон.

3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

   Решение описывает элементы из которых состоит проектное решение, их связи, обязанности и взаимодействие. Решение не описывает конкретный проект или реализацию, т.к. шаблон может применяться во многих различных ситуациях. Вместо этого, решение представляет абстрактное описание проектной задачи и общее размещение элементов (в нашем случае классов или объектов) решающих ее.

4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

   Последствия — это результаты и оптимальные варианты применения шаблона. Хотя последствия часто замалчиваются, когда мы обсуждаем проектные решения, они важны для оценки проектных альтернатив и для понимания издержек и преимуществ применения шаблона. Последствия для ПО часто включают издержки использования памяти и времени. Они также могут относиться к деталям языка и реализации.

Поскольку многократность использования часто является фактором в ОО проектировании, последствия шаблона включают его влияние на гибкость, расширяемость и переносимость системы. Явное перечисление этих последствий помогает вам понять и оценить их.

Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block. For this book we have concentrated on patterns at a certain level of abstraction. Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem. The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

Интерпретация того, что является шаблоном проектирования, а что нет, зависит от точки зрения. То, что является шаблоном для одного человека, для другого является примитивным строительным блоком. В этой книге мы сконцентрировались на шаблонах на определенном уровне абстракции. Шаблоны проектирования не описывают такие проектные решения как связные списки или хэш-таблицы, которые могут быть закодированы в виде классов и таким образом многократно использованы. Не являются они и сложными проблемно-ориентированными решениями для всего приложения или подсистемы. Проектные шаблоны в этой книге являются описанием сообщающихся классов и объектов, настроенных для решения общей проектной задачи в определенном контексте.

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample C++ and (sometimes) Smalltalk code to illustrate an implementation.

Проектный шаблон называет, резюмирует, и идентифицирует ключевые аспекты общей проектной структуры, что делает его полезным для создания многократно используемого ОО проекта. Проектный шаблон идентифицирует участвующие классы и их экземпляры, их роли и взаимодействие, а также распределение обязанностей. Каждый проектный шаблон фокусируется на конкретной задаче или вопросе ОО проекта. Шаблон описывает, когда он приложим, может ли он быть использован, учитывая прочие проектные ограничения, а также последствия и результаты его использования. Поскольку в конечном счете

мы должны реализовывать наши проекты, шаблон также включает примерный код на C++ и (иногда) на Smalltalk для того, чтобы проиллюстрировать реализацию.

Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages like Smalltalk and C++ rather than procedural languages (Pascal, C, Ada) or more dynamic object-oriented languages (CLOS, Dylan, Self). We chose Smalltalk and C++ for pragmatic reasons: Our day-to-day experience has been in these languages, and they are increasingly popular.

Хотя проектные шаблоны описывают ОО проекты, они основаны на практических решениях реализованных на наиболее распространенных ОО языках программирования, таких как Smalltalk и C++, а не на процедурных языках (Pascal, C, Ada) или более динамичных ОО языках (CLOS, Dylan, Self). Мы выбрали Smalltalk и C++ из прагматических соображений: наш повседневный опыт был в этих языках, и они все более популярны.

The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor (page 331). In fact, there are enough differences between Smalltalk and C++ to mean that some patterns can be expressed more easily in one language than the other. (See Iterator (257) for an example.)

Выбор языка программирования важен поскольку он влияет на точку зрения человека. Наши шаблоны подразумевают язык содержащий возможности уровня Smalltalk/C++ , и этот выбор определяет, что может быть легко реализовано, а что нет. Если бы мы выбрали процедурные языки, мы могли бы включить такие шаблоны, как "Наследование", "Инкапсуляция", "Полиморфизм". Аналогично, некоторые из наших шаблонов напрямую поддерживаются менее распространенными ОО языками. CLOS, например, имеет мульти-методы, которые уменьшают необходимость в таком шаблоне, как Посетитель (331). Фактически между Smalltalk и C++ существует достаточно различий, чтобы какие-либо шаблоны легче реализовывались в одном языке, чем в другом. (см. например Итератор (257)).

1.2 Design Patterns in Smalltalk MVC Шаблоны проектирования в Smalltalk MVC

The Model/View/Controller (MVC) triad of classes [KP88] is used to build user interfaces in Smalltalk-80. Looking at the design patterns inside MVC should help you see what we mean by the term “pattern.”

Триада классов Модель/Вид/Диспетчер (Model/View/Controller – MVC) использовалась для построения интерфейса пользователя в Smalltalk-80. Просмотр проектных шаблонов внутри MVC должен помочь вам понять что мы понимаем под термином “шаблон”.

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

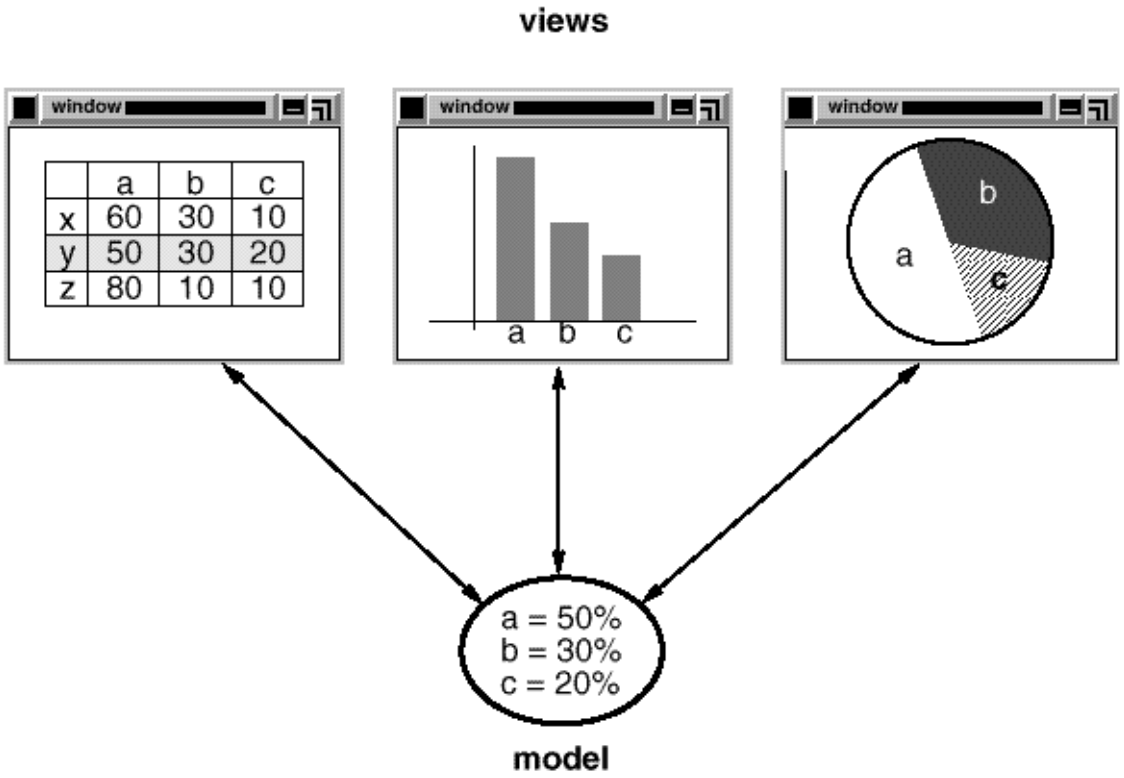
MVC состоит из трех видов объектов. Модель — это объект приложения, Вид — это его экранное представление, а Диспетчер определяет, как интерфейс реагирует на действия пользователя. До MVC проекты пользовательских интерфейсов имели тенденцию соединять эти объекты вместе. MVC разделяет их чтобы увеличить гибкость и степень многократного использования.

MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model’s data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself. This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it.

MVC разделяет виды и модели, устанавливая между ними протокол подписки/уведомления. Вид должен удостовериться, что его появление отражает состояние модели. Всякий раз, когда данные модели меняются, модель уведомляет виды, которые зависят от нее. В ответ каждый вид получает возможность обновить себя. Этот подход позволяет вам присоединять несколько видов к модели для получения различных представлений. Также вы можете создавать новые виды для модели без ее переписывания.

The following diagram shows a model and three views. (We’ve left out the controllers for simplicity.) The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways. The model communicates with its views when its values change, and the views communicate with the model to access these values.

Следующая диаграмма показывает модель и три вида. (Для простоты мы опустили объекты-диспетчеры). Модель содержит некоторые данные, а виды определяют электронную таблицу, гистограмму и круговую диаграмму отображающую эти данные различными способами. Модель взаимодействует со своими видами когда данные меняются, а виды сообщаются с моделью для доступа к этим данным.



Taken at face value, this example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. This more general design is described by the Observer (page 293) design pattern.

Номинально этот пример отражает проект, который отделяет виды от моделей. Но проектное решение применимо к более общей задаче: разделение объектов, таким образом, чтобы изменения в одном объекте могли влиять на любое количество других, без необходимости для изменяемого объекта знать о них детали. Этот более общее решение описано шаблоном Наблюдатель (стр. 293).

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. The user interface for an object inspector can consist of nested views that may be reused in a debugger. MVC supports nested views with the CompositeView class, a subclass of View.

CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

Другим свойством MVC является возможность вложения видов. Например, контрольная панель с кнопками может быть реализована как составной вид содержащий вложенные виды кнопок. Пользовательский интерфейс для инспектора объектов может состоять из вложенных видов, которые могут быть повторно использованы в отладчике. MVC поддерживает вложенные виды с помощью класса CompositeView, являющегося подклассом View. Объекты класса CompositeView действуют в точности как объекты класса View; составной вид может быть использован везде где может использоваться вид, но он также содержит вложенными виды и управляет ими.

Again, we could think of this as a design that lets us treat a composite view just like we treat one of its components. But the design is applicable to a more general problem, which occurs whenever we want to group objects and treat the group like an individual object. This more general design is described by the Composite (163) design pattern. It lets you create a class hierarchy in which some subclasses define primitive objects (e.g., Button) and other classes define composite objects (CompositeView) that assemble the primitives into more complex objects.

И снова, мы можем думать об этом, как о проектном решении, которое позволяет работать с составным видом так же как с одним из его компонентов. Но проектное решение применимо к более общей задаче, которая появляется всегда, когда мы хотим сгруппировать объекты и работать с группой, как с отдельным объектом. Это более общее решение описано в шаблоне Композит(163). Он позволяет вам создать иерархию классов в которых некоторые подклассы определяют примитивные объекты (например Кнопка), а другие классы определяют составные объекты (CompositeView), которые собирают примитивы в более сложные объекты.

MVC also lets you change the way a view responds to user input without changing its visual presentation. You might want to change the way it responds to the keyboard, for example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a Controller object. There is a class hierarchy of controllers, making it easy to create a new controller as a variation on an existing one.

Также MVC позволяет изменять реакцию вида на ввод пользователя без изменения его визуального представления. Например вы хотите изменить его реакцию на ввод с клавиатуры, или использовать управляющее меню вместо командных клавиш. MVC инкапсулирует механизм ответа в объекте Диспетчер. Существует иерархия классов диспетчеров, упрощающая создание нового диспетчера в виде варианта уже существующего.

A view uses an instance of a Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller. It's even possible to change a view's controller at run-time to let the view change the way it responds to user input. For example, a view can be disabled so that it doesn't accept input simply by giving it a controller that ignores input events.

Вид использует экземпляр подкласса Диспетчер для реализации конкретной стратегии ответа; для реализации другой стратегии, просто замените экземпляр другим видом диспетчера. Можно даже заменить диспетчер вида на другой во время выполнения, чтобы позволить виду изменить реакцию на ввод пользователя. Например, вид может быть отключен таким образом, чтобы он не воспринимал ввод пользователя просто подключением к нему диспетчера игнорирующего ввод.

The View-Controller relationship is an example of the Strategy (315) design pattern. A Strategy is an object that represents an algorithm. It's useful when you want to replace the algorithm either statically or dynamically, when you have a lot of variants of the algorithm, or when the algorithm has complex data structures that you want to encapsulate.

Связь Вид-Диспетчер является примером проектного шаблона Стратегия (315). Стратегия — это объект, который представляет собой алгоритм. Он полезен, когда вы хотите заменить алгоритм статически или динамически, когда у вас есть много вариантов алгоритма, или когда алгоритм содержит сложные структуры данных, которые вы хотите инкапсулировать.

MVC uses other design patterns, such as Factory Method (107) to specify the default controller class for a view and Decorator (175) to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite, and Strategy design patterns.

MVC использует другие шаблоны проектирования, такие как Заводской Метод (107) для указания диспетчера вида по умолчанию и Декоратор(107) для добавления скроллинга к виду. Но основные связи в MVC представлены проектными шаблонами Наблюдатель, Композит и Стратегия.

1.3 Describing Design Patterns
Описание Проектных Шаблонов

How do we describe design patterns? Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action.

Как мы описываем проектные шаблоны? Графические обозначения важны и полезны, но недостаточны. Они просто фиксируют конечный результат процесса проектирования в виде связей между классами и объектами. Для многократного использования проекта, мы должны также записать наши решения, альтернативы и оптимальные варианты, которые к нему привели. Конкретные примеры также важны, потому что они помогают увидеть проект в действии.

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

Мы описываем проектные шаблоны, используя последовательный формат. Описание каждого шаблона делится на параграфы по следующему шаблону :). Этот шаблон предоставляет единообразную структуру для информации, упрощая изучение, использование и сравнение проектных шаблонов.

Pattern Name and Classification Название шаблона и классификация

> The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the scheme we introduce in Section 1.5.
>
> Название шаблона кратко передает его сущность. Хорошее имя важно, поскольку оно станет частью вашего проектного словаря. Классификация шаблонов отражает схему из раздела 1.5.

Intent Намерение

> A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Короткое утверждение отвечающее на следующие вопросы: Что делает проектный шаблон? Каково его обоснование и назначение? Для какой именно проектной задачи он предназначен?

Also Known As Также Известен Как

Other well-known names for the pattern, if any.

Другие хорошо известные имена для шаблона, если есть.

Motivation Мотивация

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Сценарий иллюстрирующий проектную задачу и показывающий как структуры классов и объектов в шаблоне решают ее. Сценарий поможет вам понять более абстрактное описание шаблона.

Applicability Сфера применения

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Каковы ситуации в которых может быть применен проектный шаблон? Каковы примеры плохих проектов в которых он может быть применен? Как вы можете распознать такие ситуации?

Structure Структура

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RBP+91]. We also use interaction diagrams [JCJO92, Boo94] to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.

Графическое представление классов в шаблоне (используемая нотация основана на Технике Моделирования Объектов) (OMT) [RBP+91]. Мы также используем диаграммы взаимодействия [JCJO92, Boo94] для иллюстрации последовательности запросов и взаимодействий между объектами. В Приложении В эти нотации описываются подробно.

Participants Участники

The classes and/or objects participating in the design pattern and their responsibilities.

Классы и/или объекты, участвующие в проектном шаблоне, а также их обязанности.

Collaborations Взаимодействия

How the participants collaborate to carry out their responsibilities.

Как участники взаимодействуют для выполнения своих обязанностей.

Consequences Последствия

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Как шаблон реализует свои цели? Каковы издержки и результаты использования шаблона? Какой аспект структуры системы он позволяет вам изменять независимо?

Implementation Реализация

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

В наличии каких ловушек, трюков и техник вы должны отдавать себе отчет при реализации шаблона? Существуют ли проблемы специфические для определенного языка?

Sample Code Пример кода

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Фрагменты кода, которые показывают как вы можете реализовать шаблон на C++ или Smalltalk.

Known Uses Известные использования

Examples of the pattern found in real systems. We include at least two examples from different domains.

Примеры использования шаблона в реальных системах. Мы включили как минимум два примера из разных областей применения.

Related Patterns Связь с шаблонами

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Какие шаблоны проектирования тесно связаны с этим? Каковы важные различия? С какими другими шаблонами должен использоваться данный шаблон?

The appendices provide background information that will help you understand the patterns and the discussions surrounding them. Appendix A is a glossary of terminology we use. We've already mentioned Appendix B, which presents the various notations. We'll also describe aspects of the notations as we introduce them in the upcoming discussions. Finally, Appendix C contains source code for the foundation classes we use in code samples.

Приложения предоставляют дополнительную информацию, которая поможет вам понять шаблоны и их обсуждение. Приложение A — это словарь использованной терминологии. Мы уже упоминали приложение B в котором представлены различные нотации. Мы также опишем аспекты нотаций когда мы представим их в последующих обсуждениях. И наконец приложение C содержит исходный код для базовых классов используемых нами в примерах.

1.4 The Catalog of Design Patterns Каталог проектных шаблонов

The catalog beginning on page 79 contains 23 design patterns. Their names and intents are listed next to give you an overview. The number in parentheses after each pattern name gives the page number for the pattern (a convention we follow throughout the book).

Каталог начинается на странице 79 и содержит 23 проектных шаблона. Их имена и намерения перечислены ниже в виде обзора. Номер в скобках после каждого шаблона означает номер страницы на которой описан шаблон (этого соглашения мы придерживаемся во всей книге)

Абстрактная фабрика (Abstract Factory) (87)

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов.

Адаптер (Adapter) (139)

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Преобразует интерфейс класса в другой интерфейс, ожидаемый клиентами. Адаптер позволяет классам с несовместимыми интерфейсами работать вместе.

Мост (Bridge) (151)

Decouple an abstraction from its implementation so that the two can vary independently.

Отцепляет абстракцию от ее реализации так, чтобы они могли изменяться независимо.

Строитель (Builder) (97)

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Отделяет конструирование сложных объектов от его представления, так чтобы один конструирующий процесс мог создавать различные представления.

Цепь ответственности (Chain of Responsibility) (223)

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Избегает связывания объекта пославшего запрос с его получателем путем предоставления возможности обработки запроса более чем одному объекту. Сцепляет объекты получатели и передает запрос по цепочке до тех пор пока какой-либо объект не обработает его.

Команда (Command) (233)

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Инкапсулирует запрос в объект, таким образом, позволяя параметризовать клиентов различными запросами, ставить запросы в очередь или протоколировать их, а также поддерживать отменяемые операции.

Композит (Composite) (163)

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Соединяет объекты в древовидные структуры для представления иерархий вида целое-часть. Композит позволяет клиентам обращаться с отдельными объектами также как и с группами объектов.

Декоратор (Decorator) (175)

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Динамически присоединяет к объекту дополнительные обязанности. Декораторы предоставляют гибкую альтернативу сабклассингу для расширения функциональности.

Фасад (Facade) (185)

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Предоставляет унифицированный интерфейс для множества интерфейсов в подсистеме. Фасад определяет интерфейс более высокого уровня для облегчения использования подсистемы.

Заводской метод (Factory Method) (107)

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Определяет интерфейс для создания объекта, но позволяет подклассам решать экземпляр какого класса создать. Заводской метод позволяет классу уступать создание подклассам.

Легковес (Flyweight) (195)

Use sharing to support large numbers of fine-grained objects efficiently.

Использует разделение для эффективной работы с большим количеством маленьких объектов.

Интерпретатор (Interpreter) (243)

Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Для данного языка определяет представление его грамматики вместе с интерпретатором, который использует грамматику для интерпретации предложений на этом языке.

Итератор (Iterator) (257)

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Предоставляет способ последовательного доступа к элементам составного объекта без раскрытия его представления.

Посредник (Mediator) (273)

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Определяет объект, инкапсулирующий взаимодействие некоторого множества объектов. Посредник обеспечивает ослабление сцепления, не давая объектам ссылаться друг на друга явно, и позволяет независимо изменять их взаимодействие.

Память (Memento) (283)

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Без нарушения инкапсуляции фиксирует и сохраняет внутреннее состояние объекта, чтобы можно было вернуться к этому состоянию позже.

Наблюдатель (Observer) (293)

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Определяет отношение один ко многим между объектами таким образом, что когда один объект изменяет состояние, все зависящие от него объекты уведомляются и обновляются автоматически.

Прототип (Prototype) (117)

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Указывает какие виды объектов создавать, используя прототипный экземпляр и создает новые объекты копированием этого прототипа.

Полномочия (Proxy) (207)

Provide a surrogate or placeholder for another object to control access to it.

Предоставляет заменитель для другого объекта для контроля доступа к нему.

Одиночка (Singleton) (127)

Ensure a class only has one instance, and provide a global point of access to it.

Проверяет что класс имеет только один экземпляр и обеспечивает глобальную точку доступа к нему.

Состояние (State) (305)

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Позволяет объекту изменять свое поведение когда его внутреннее состояние меняется. Создает впечатление, что объект сменил свой класс.

Стратегия (Strategy) (315)

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Определяет семейство алгоритмов, инкапсулирует каждый и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритм независимо от клиентов его использующих.

Шаблонный метод (Template Method) (325)

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Определяет скелет алгоритма в операции, оставляя некоторые шаги для реализации в подклассах. Шаблонный метод позволяет подклассам переопределять определенные шаги алгоритма без изменения его структуры.

Посетитель [Visitor] (331)

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Представляет действие, которое необходимо произвести над элементами объектной структуры. Посетитель позволяет определить новое действие без изменения классов элементов над которыми это действие производится.

1.5 Organizing the Catalog
Организация каталога

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them. This section classifies design patterns so that we can refer to families of related patterns. The classification helps you learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

Проектные шаблоны различаются по степени детализации и уровню абстракции. Поскольку существует много проектных шаблонов, нам нужен способ их организации. В этом разделе проектные шаблоны классифицируются, чтобы мы могли ссылаться на семейства связанных шаблонов. Классификация позволяет вам изучать шаблоны в каталоге быстрее, а также она может направить ваши усилия для поиска новых шаблонов.

We classify design patterns by two criteria (Table 1.1). The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

	Creational	**Structural**	**Behavioral**
Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility(223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Таблица 1.1: Design pattern space

Мы классифицируем проектные шаблоны по двум критериям (Таблица 1.2). Первый критерий - назначение, показывает что делает шаблон. Шаблоны могут иметь создающее, структурное или поведенческое назначение. Создающие шаблоны имеют отношение к процессу создания объекта. Структурные шаблоны имеют дело с композицией классов или объектов. Поведенческие шаблоны характеризуют пути взаимодействия объектов и распределения ответственности между ними.

	Создающие	**Структурные**	**Поведенческие**
Классовые	Заводской метод (107)	Адаптер (139)	Интерпретатор (243) Шаблонный метод (325)
Объектные	Абстрактная фабрика(87) Строитель (97) Прототип (117) Одиночка (127)	Адаптер (139) Мост (151) Композит (163) Декоратор (175) Фасад (185) Полномочия(207)	Цепь ответственности(223) Команда (233) Итератор (257) Посредник (273) Память (283) Легковес (195) Наблюдатель (293) Состояние (305) Стратегия (315) Посетитель (331)

Таблица 1.2: Пространство проектных шаблонов

The second criterion, called **scope**, specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled “class patterns” are those that focus on class relationships. Note that most patterns are in the Object scope.

Второй критерий — **область применения**, указывает применим ли шаблон в основном к классам или объектам. Шаблоны классов имеют дело со связями между классами и их подклассами. Эти связи устанавливаются через наследование, таким образом они статически определены во время компиляции. Объектные шаблоны имеют дело со связями

между объектами, которые более динамичны и могут быть изменены во время выполнения. Почти все шаблоны в какой-то мере используют наследование. Поэтому только шаблоны отмеченные как “шаблоны классов” фокусируются на связях классов. Заметьте, что большинство шаблонов работают с объектами.

Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Создающие шаблоны класса оставляют часть процесса создания объекта подклассам, тогда как Создающие объектные шаблоны оставляют часть создания другому объекту. Структурные шаблоны класса используют наследование для соединения классов, тогда как Структурные шаблоны объектов описывают способы сборки объектов. Поведенческие шаблоны класса используют наследование для описания алгоритмов и потока управления, а Поведенческие шаблоны объекта описывают как группа объектов взаимодействует для решения задачи, которую ни один объект не может выполнить самостоятельно.

There are other ways to organize the patterns. Some patterns are often used together. For example, Composite is often used with Iterator or Visitor. Some patterns are alternatives: Prototype is often an alternative to Abstract Factory. Some patterns result in similar designs even though the patterns have different intents. For example, the structure diagrams of Composite and Decorator are similar.

Существуют другие способы организации шаблонов. Некоторые шаблоны часто используются вместе. Например Композит часто используется с Итератором и Посетителем. Некоторые шаблоны имеют альтернативы: Прототип часто является альтернативой Абстрактной Фабрике. Некоторые шаблоны ведут к похожим проектным решениям, даже если эти шаблоны имеют разные намерения. Например структурные диаграммы Композита и Декоратора похожи.

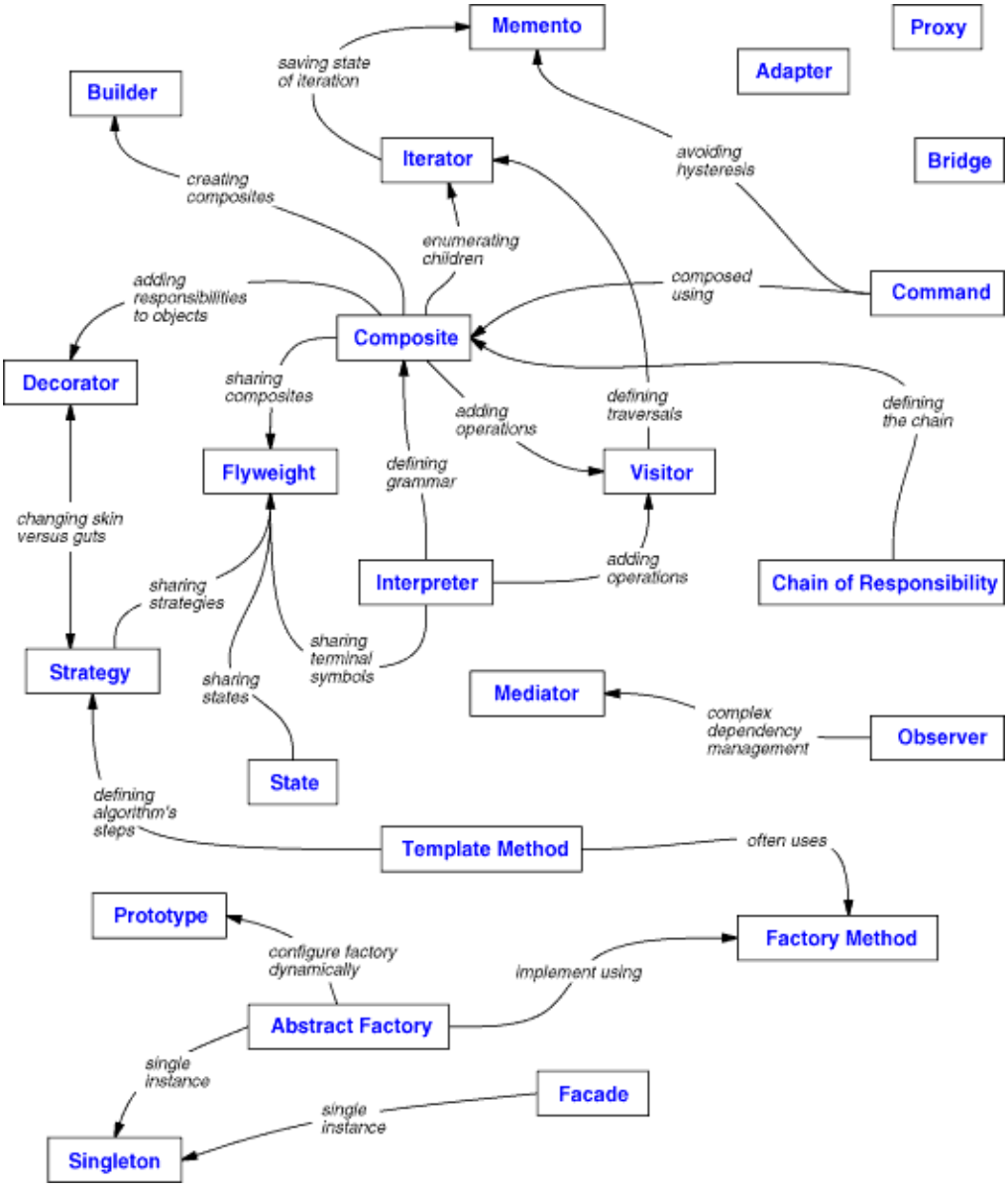
Yet another way to organize design patterns is according to how they reference each other in their “Related Patterns” sections. Figure 1.1 depicts these relationships graphically.

Еще один способ организации шаблонов — согласно их ссылкам друг на друга в разделе “Связанные шаблоны”. Рисунок 1.1 описывает эти связи графически.

Clearly there are many ways to organize design patterns. Having multiple ways of thinking about patterns will deepen your insight into what they do, how they compare, and when to apply them.

Несомненно существует много способов организации проектных шаблонов. Наличие нескольких способов размышления о шаблонах углубит ваше понимание того, что они делают, как они соотносятся и как их применять.

Рис. 1.1: Design pattern relationships Взаимосвязи проектных шаблонов



1.6 How Design Patterns Solve Design Problems
Как проектные шаблоны решают проектные задачи

Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways. Here are several of these problems and how design patterns solve them.

Проектные шаблоны решают множество ежедневных проблем с которыми сталкиваются ОО проектировщики, множеством разных способов. Далее перечислены несколько подобных задач и как проектные шаблоны их решают.

1.6.1 Finding Appropriate Objects
Поиск необходимых объектов

Object-oriented programs are made up of **objects**. An object packages both data and the procedures that operate on that data. The procedures are typically called **methods** or **operations**. An object performs an operation when it receives a **request** (or **message**) from a **client**.

ОО программы состоят из **объектов**. Объект содержит как данные так и процедуры работающие с этими данными. Процедуры обычно называются **методами** или **операциями**. Объект выполняет операцию когда он получает **запрос** (или **сообщение**) от **клиента**.

Requests are the only way to get an object to execute an operation. Operations are the only way to change an object's internal data. Because of these restrictions, the object's internal state is said to be **encapsulated**; it cannot be accessed directly, and its representation is invisible from outside the object.

Запросы это единственный способ заставить объект выполнить операцию. Операции это единственный способ изменения внутренних данных объекта. Из-за этих ограничений, внутреннее состояние объекта называется **инкапсулированным**; оно недоступно напрямую и его представление невидимо извне объекта.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, and on and on. They all influence the decomposition, often in conflicting ways.

Трудной частью ОО проектирования является декомпозиция системы на объекты. Эта задача трудна, потому что на нее влияют многие факторы: инкапсуляция, размер, зависимость, гибкость, производительность, эволюция, повторное использования и т.д. и т.п. Все это влияет на декомпозицию и часто противоречит друг другу.

Object-oriented design methodologies favor many different approaches. You can write a problem statement, single out the nouns and verbs, and create corresponding classes and operations. Or you can focus on the collaborations and responsibilities in your system. Or you can model the real world and translate the objects found during analysis into design. There will always be disagreement on which approach is best.

ОО методологии проектирования используют множество разных подходов. Вы можете написать задачу в виде утверждения, выделить существительные и глаголы и создать соответствующие классы и операции. Или вы можете сфокусироваться на взаимодействиях и ответственности в вашей системе. Или смоделировать реальный мир и перевести объекты найденные в ходе анализа в проект. Всегда будут существовать разногласия, о том какой подход лучше.

Many objects in a design come from the analysis model. But object-oriented designs often end up with classes that have no counterparts in the real world. Some of these are low-level classes like arrays. Others are much higher-level. For example, the Composite (163) pattern introduces an abstraction for treating objects uniformly that doesn't have a physical counterpart. Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible.

Многие объекты приходят в проект из аналитической модели. Но ОО проекты часто приходят к классам у которых нет эквивалентов в реальном мире. Некоторые из них это низкоуровневые классы, такие как массивы. Другие гораздо более высокоуровневые. Например шаблон Композит (163) представляет абстракцию для работы с объектами единообразно, не имеющую физического эквивалента. Строгое моделирование реального мира приводит к системе, которая отражает сегодняшние реалии, но необязательно завтрашние. Абстракции возникающие во время проектирования являются ключевыми для создания гибкого проекта.

Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs. The Strategy (315) pattern describes how to implement interchangeable families of algorithms. The State (305) pattern represents each state of an entity as an object. These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.

Проектные шаблоны помогают вам идентифицировать менее явные абстракции и объекты, которые могут использовать их. Например объекты, представляющие процесс или

алгоритм не существуют в природе, но они являются важной частью гибких проектов. Шаблон Стратегия (315) описывает как реализовать взаимозаменяемые семейства алгоритмов. Шаблон Состояние (305) представляет каждое состояние сущности в виде объекта. Эти объекты редко обнаруживаются во время анализа или даже на ранних стадиях проектирования; они обнаруживаются позже в процессе переделки проекта для большей гибкости и многократного использования.

1.6.2 Determining Object Granularity Определение размера объектов

Objects can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. How do we decide what should be an object?

Объекты могут очень сильно отличаться в размере и количестве. Они могут представлять все от аппаратной части до целых приложений. Как же решить что должно быть объектом?.

Design patterns address this issue as well. The Facade (185) pattern describes how to represent complete subsystems as objects, and the Flyweight (195) pattern describes how to support huge numbers of objects at the finest granularities. Other design patterns describe specific ways of decomposing an object into smaller objects. Abstract Factory (87) and Builder (97) yield objects whose only responsibilities are creating other objects. Visitor (331) and Command (233) yield objects whose only responsibilities are to implement a request on another object or group of objects.

Проектные шаблоны решают также и этот вопрос. Шаблон Фасад (185) описывает как представлять целые подсистемы в виде объектов, а шаблон Легковес (195) описывает как удерживать огромное количество объектов в небольшом объеме. Другие проектные шаблоны описывают специфические пути декомпозиции объекта на меньшие объекты. Абстрактная Фабрика (87) и Строитель (97) представляют объекты, единственной обязанностью которых является создание других объектов. Посетитель (331) и Команда (233) собирают объекты единственной обязанностью которых является реализация запроса другого объекта или группы объектов.

1.6.3 Specifying Object Interfaces Определение Объектных Интерфейсов

Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's **signature**. The

set of all signatures defined by an object's operations is called the **interface** to the object. An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object.

Каждая операция в объекте определяется именем, объектами, принимаемыми в качестве параметров, и результатом. Это так называемая **сигнатура** операции. Множество всех сигнатур, определенных в объекте операций, называется **интерфейсом** объекта. Интерфейс объекта характеризует полное множество запросов, которые можно направлять объекту. Любой запрос подходящий под сигнатуру в интерфейсе объекта может быть послан объекту.

A **type** is a name used to denote a particular interface. We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window." An object may have many types, and widely different objects can share a type. Part of an object's interface may be characterized by one type, and other parts by other types. Two objects of the same type need only share parts of their interfaces. Interfaces can contain other interfaces as subsets. We say that a type is a **subtype** of another if its interface contains the interface of its **supertype**. Often we speak of a subtype *inheriting* the interface of its supertype.

Тип это имя используемое для ссылки на соответствующий интерфейс. Мы говорим, что объект имеет тип "Окно", если он принимает все запросы для операций определенных в интерфейсе "Окно". Объект может иметь много типов и многие различные объекты могут разделять один тип. Часть интерфейса объекта может быть описана одним типом, а другие части другими типами. Два объекта одного и того же типа должны разделять только части их интерфейсов. Интерфейсы могут содержать другие интерфейсы в качестве подмножеств. Мы говорим, что тип является **подтипом** другого типа, если его интерфейс содержит интерфейс его **супертипа**. Часто мы говорим о подтипе *наследующем* интерфейс своего супертипа.

Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. There is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation – different objects are free to implement requests differently. That means two objects having completely different implementations can have identical interfaces.

Интерфейсы фундаментальны для ОО систем. Объекты известны только через их интерфейсы. Невозможно узнать что-нибудь об объекте или попросить его сделать что-то не через интерфейс. Интерфейс объекта ничего не сообщает о его реализации — разные

объекты могут обрабатывать запросы по разному. Это означает, что два объекта, имеющие совершенно разные реализации, могут иметь идентичные интерфейсы.

When a request is sent to an object, the particular operation that's performed depends on both the request and the receiving object. Different objects that support identical requests may have different implementations of the operations that fulfill these requests. The run-time association of a request to an object and one of its operations is known as **dynamic binding**.

Когда запрос послан объекту, выполнение соответствующей операции зависит и от запроса и от получающего объекта. Различные объекты поддерживающие одинаковые запросы могут по разному реализовывать операции выполняющие эти запросы. Связывание запроса к объекту с одной из его операций во время выполнения известно как **динамическое связывание**.

Dynamic binding means that issuing a request doesn't commit you to a particular implementation until run-time. Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request. Moreover, dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as **polymorphism**, and it's a key concept in object-oriented systems. It lets a client object make few assumptions about other objects beyond supporting a particular interface. Polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.

Динамическое связывание означает что вызов запроса не привязывает вас к конкретной реализации до времени выполнения. Следовательно вы можете писать программы, которые ожидают объект с конкретным интерфейсом, зная, что каждый объект с корректным интерфейсом обработает запрос. Более того, динамическое связывание позволяет вам заменять объекты с идентичными интерфейсами друг на друга во время выполнения. Такая замещаемость называется **полиморфизмом** и является ключевой концепцией в ОО системах. Это позволяет клиентским объектам не делать много предположений о других объектах, кроме поддержки соответствующего интерфейса. Полиморфизм упрощает определения клиентов, отделяет объекты друг от друга и позволяет им менять связь друг с другом во время выполнения.

Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what not to put in the interface. The Memento (283) pattern is a good example. It describes how to encapsulate and save the internal state of an object so that the object can be restored to that state later. The pattern stipulates that Memento objects must define two interfaces: a

restricted one that lets clients hold and copy mementos, and a privileged one that only the original object can use to store and retrieve state in the memento.

Проектные шаблоны помогают вам определить интерфейсы, идентифицируя их ключевые элементы и данные пересылаемые через интерфейс. Хороший пример этого — шаблон Память (283). Он описывает как инкапсулировать и сохранить внутреннее состояние объекта так, чтобы объект мог вернуться к этому состоянию позже. В шаблоне указывается, что объекты Память должны определять два интерфейса: ограниченный, который позволяет клиентам содержать и копировать объекты, и привилегированный, который может использовать исходный объект для сохранения и получения своего состояния.

Design patterns also specify relationships between interfaces. In particular, they often require some classes to have similar interfaces, or they place constraints on the interfaces of some classes. For example, both Decorator (175) and Proxy (207) require the interfaces of Decorator and Proxy objects to be identical to the decorated and proxied objects. In Visitor (331), the Visitor interface must reflect all classes of objects that visitors can visit.

Проектные шаблоны также определяют связи между интерфейсами. В частности они часто требуют, чтобы некоторые классы имели похожие интерфейсы. Например шаблоны Декоратор (175) и Полномочия (207) требуют чтобы интерфейсы их объектов были идентичны. В шаблоне Посетитель (331), интерфейс должен отражать все классы объектов которые он может посещать.

1.6.4 Specifying Object Implementations Определение Реализаций Объектов

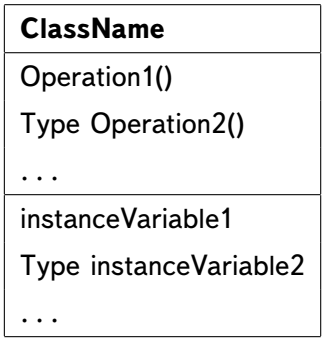
So far we've said little about how we actually define an object. An object's implementation is defined by its **class**. The class specifies the object's internal data and representation and defines the operations the object can perform.

До этого мы мало говорили о том, как мы действительно определяем объект. Реализация объекта определяется его **классом**. Класс специфицирует внутренние данные объекта и его представление, а также определяет операции, которые объект может выполнять.

Our OMT-based notation (summarized in Appendix B) depicts a class as a rectangle with the class name in bold. Operations appear in normal type below the class name. Any data that the class defines comes after the operations. Lines separate the class name from the operations and the operations from the data:

Наша нотация основанная на OMT (описанная в приложении B) изображает класс как прямоугольник, в котором жирным шрифтом написано имя класса. Операции написаны

нормальным шрифтом ниже имени класса. Любые данные, которые определяет класс идут после действий. Имя класса, действия и данные разделяются линиями.



Return types and instance variable types are optional, since we don't assume a statically typed implementation language.

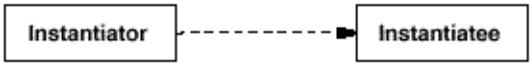
Типы возвращаемых данных и тип экземпляра класса опциональны, т.к. мы не предполагаем использование статически типизированного языка.

Objects are created by **instantiating** a class. The object is said to be an **instance** of the class. The process of instantiating a class allocates storage for the object's internal data (made up of **instance variables**) and associates the operations with these data. Many similar instances of an object can be created by instantiating a class.

Объекты создаются **инстанцированием** класса. Объект называется **экземпляром** класса. Процесс инстанцирования класса выделяет пространство для данных объекта и связывает операции с этими данными. Многие одинаковые экземпляры объектов могут созданы инстанцированием класса.

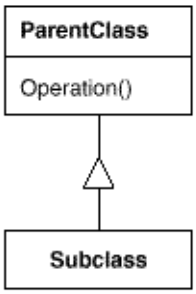
A dashed arrowhead line indicates a class that instantiates objects of another class. The arrow points to the class of the instantiated objects.

Пунктирная линия со стрелкой показывает класс, который инстанцирует объекты другого класса. Стрелка указывает на класс создаваемых объектов.



New classes can be defined in terms of existing classes using **class inheritance**. When a **subclass** inherits from a **parent class**, it includes the definitions of all the data and operations that the parent class defines. Objects that are instances of the subclass will contain all data defined by the subclass and its parent classes, and they'll be able to perform all operations defined by this subclass and its parents. We indicate the subclass relationship with a vertical line and a triangle:

Новые классы могут быть определены в терминах существующих классов с использованием **наследования**. Когда **подкласс** наследует от **родительского класса**, он включает определения всех данных и операций родительского класса. Объекты экземпляры подкласса будут содержать все данные определенные подклассом и его родительскими классами, и они смогут выполнять все операции определенные в подклассе и в его родителях. Мы показываем отношения подкласса вертикальной чертой с треугольником.



An **abstract class** is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated. The operations that an abstract class declares but doesn't implement are called **abstract operations**. Classes that aren't abstract are called **concrete classes**.

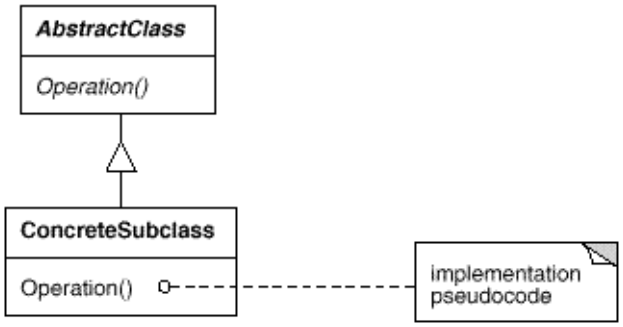
Абстрактный класс это класс, основная цель которого определить общий интерфейс для его подклассов. Абстрактный класс оставляет часть или всю свою реализацию операциям определенных в подклассах. Таким образом, абстрактный класс не может быть инстанцирован. Операции, которые абстрактный класс определяет, но не реализовывает называются **абстрактными операциями**. Не абстрактные классы называются **конкретными**.

Subclasses can refine and redefine behaviors of their parent classes. More specifically, a class may **override** an operation defined by its parent class. Overriding gives subclasses a chance to handle requests instead of their parent classes. Class inheritance lets you define classes simply by extending other classes, making it easy to define families of objects having related functionality.

Подклассы могут совершенствовать и переопределять поведение родительских классов. Точнее класс может **заменить** операцию определенную в родительском классе. Переопределение дает подклассам возможность обрабатывать запросы вместо их родительских классов. Наследование позволяет вам определять классы, просто расширяя другие классы, и упрощая тем самым создание семейств объектов с похожей функциональностью.

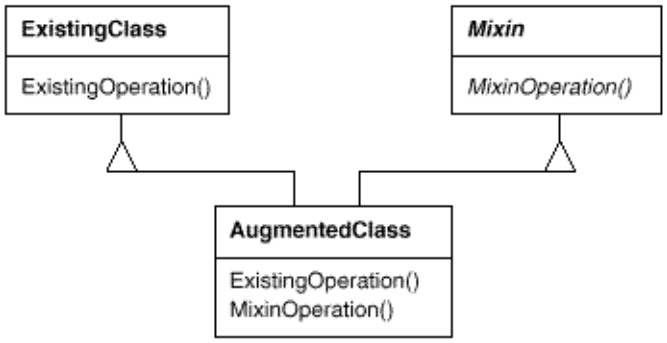
The names of abstract classes appear in slanted type to distinguish them from concrete classes. Slanted type is also used to denote abstract operations. A diagram may include pseudocode for an operation's implementation; if so, the code will appear in a dog-eared box connected by a dashed line to the operation it implements.

Имена абстрактных классов пишутся наклонным шрифтом для того, чтобы отличить их от конкретных классов. Наклонный шрифт также используется для определения абстрактных операций. Диаграмма может включать псевдокод для реализации операций; в таком случае код изображается в прямоугольнике с загнутым уголком подсоединенным пунктирной линией к операции которую он реализует.



A **mixin class** is a class that's intended to provide an optional interface or functionality to other classes. It's similar to an abstract class in that it's not intended to be instantiated. Mixin classes require multiple inheritance:

Класс-примесь — это класс, который предоставляет опциональный интерфейс или функциональность для других классов. Он похож на абстрактный класс, в том, что он также не предназначен для инстанцирования. Классы-примеси требуют наличия множественного наследования:



Class versus Interface Inheritance
Наследование классов против наследования интерфейсов.

It's important to understand the difference between an object's *class* and its *type*.

Важно понять разницу между *классом* объекта и его *типом*.

An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations. In contrast, an object's type only refers to its interface-the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.

Класс определяет внутреннее состояние объекта и реализацию его операций. С другой стороны тип объекта только ссылается на его интерфейс — множество запросов, на которые объект реагирует. Объект может иметь много типов, и объекты разных классов могут иметь один тип.

Of course, there's a close relationship between class and type. Because a class defines the operations an object can perform, it also defines the object's type. When we say that an object is an instance of a class, we imply that the object supports the interface defined by the class.

Конечно, существует тесная взаимосвязь между классом и типом. Поскольку класс определяет операции выполняемые объектом, он также определяет тип объекта. Когда мы говорим, что объект это экземпляр класса, мы подразумеваем, что объект поддерживает интерфейс определенный классом.

Languages like C++ and Eiffel use classes to specify both an object's type and its implementation. Smalltalk programs do not declare the types of variables; consequently, the compiler does not check that the types of objects assigned to a variable are subtypes of the variable's type. Sending a message requires checking that the class of the receiver implements the message, but it doesn't require checking that the receiver is an instance of a particular class.

Такие языки как C++ и Eiffel используют классы для указания как типа объекта так и его реализации. Smalltalk программы не определяют типы переменных; следовательно компилятор не проверяет, что типы объектов присвоенных переменной являются подтипами типа этой переменной. Посылка сообщения требует проверки, что класс получателя обрабатывает это сообщение, но не требует проверки того, что получатель является экземпляром определенного класса.

It's also important to understand the difference between class inheritance and interface inheritance (or subtyping). Class inheritance defines an object's implementation in terms of another object's implementation. In short, it's a mechanism for code and representation

sharing. In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another.

Также важно понять разницу между наследованием классов и интерфейсов. Наследование классов определяет реализацию объекта в терминах реализации другого объекта. Вкратце это механизм совместного использования кода и представления данных. С другой стороны наследование интерфейсов объясняет, когда один объект может быть использован вместо другого.

It's easy to confuse these two concepts, because many languages don't make the distinction explicit. In languages like C++ and Eiffel, inheritance means both interface and implementation inheritance. The standard way to inherit an interface in C++ is to inherit publicly from a class that has (pure) virtual member functions. Pure interface inheritance can be approximated in C++ by inheriting publicly from pure abstract classes. Pure implementation or class inheritance can be approximated with private inheritance. In Smalltalk, inheritance means just implementation inheritance. You can assign instances of any class to a variable as long as those instances support the operation performed on the value of the variable.

Легко перепутать эти две концепции, поскольку многие языки явно не разделяют их. В таких языках как C++ и Eiffel, наследование означает одновременно наследование интерфейса и реализации. Стандартный способ унаследовать интерфейс в C++ — унаследовать открыто от класса, который имеет (чисто) виртуальные функции-члены. Чистое наследование интерфейсов может быть приблизительно реализовано в C++ открытым наследованием от чисто абстрактных классов. Чистое наследование реализации или класса приблизительно реализуется закрытым наследованием. В Smalltalk наследуется только реализация. Вы можете присваивать экземпляры любого класса переменной, если они поддерживают операции осуществляемые над значением переменной.

Although most programming languages don't support the distinction between interface and implementation inheritance, people make the distinction in practice. Smalltalk programmers usually act as if subclasses were subtypes (though there are some well-known exceptions [Coo92]); C++ programmers manipulate objects through types defined by abstract classes.

Хотя многие языки программирования не поддерживают различие между наследованием реализации и интерфейса, люди делают различие на практике. Программирующие на Smalltalk обычно действуют так если бы подклассы были подтипами (хотя есть несколько известных исключений); Программирующие на C++ работают с объектами через типы определенные в абстрактных классах.

Many of the design patterns depend on this distinction. For example, objects in a Chain of Responsibility (223) must have a common type, but usually they don't share a common

implementation. In the Composite (163) pattern, Component defines a common interface, but Composite often defines a common implementation. Command (233), Observer (293), State (305), and Strategy (315) are often implemented with abstract classes that are pure interfaces.

Многие из шаблонов проектирования зависят от этого различия. Например объекты в Цепи Ответственности (223) должны иметь общий тип, но обычно они не разделяют общую реализацию. В шаблоне Композит (163) Компонент определяет общий интерфейс, но Композит часто определяет общую реализацию. Команда (233), Наблюдатель (293), Состояние (305) и Стратегия (315) часто реализуются с помощью абстрактных классов и чистых интерфейсов.

Programming to an Interface, not an Implementation Программирование для интерфейса, а не реализации

Class inheritance is basically just a mechanism for extending an application's functionality by reusing functionality in parent classes. It lets you define a new kind of object rapidly in terms of an old one. It lets you get new implementations almost for free, inheriting most of what you need from existing classes.

Наследование классов в основном только механизм для расширения функциональности приложения с помощью повторного использования функциональности родительских классов. Оно позволяет вам быстро определять новый вид объекта в терминах старого. Оно позволяет получать новые реализации практически даром, наследуя большую часть того что вам нужно от существующих классов.

However, implementation reuse is only half the story. Inheritance's ability to define families of objects with *identical* interfaces (usually by inheriting from an abstract class) is also important. Why? Because polymorphism depends on it.

Тем не менее, использование реализации это только половина дела. Также важна способность наследования определять семейства объектов с *идентичными* интерфейсами (обычно наследуя от абстрактного класса). Почему? Потому что от этого зависит полиморфизм.

When inheritance is used carefully (some will say *properly*), all classes derived from an abstract class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class. *All* subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class.

Когда наследование используется аккуратно (некоторые скажут *правильно*), все классы унаследованные от абстрактного класса разделят его интерфейс. Это подразумевает,

что подкласс просто добавляет или переопределяет операции, и не прячет операции родительского класса. *Все* подклассы могут отвечать на запросы в интерфейсе абстрактного класса, что делает их подтипами абстрактного класса.

There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

У работы с объектами исключительно в терминах интерфейса определенного абстрактным классом есть два преимущества:

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.

   Клиенты не беспокоятся о конкретных типах используемых объектов, пока объекты придерживаются интерфейса ожидаемого клиентами.

2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

   Клиенты не беспокоятся о классах, которые реализуют эти объекты. Клиенты знают только об абстрактных класс(ах) определяющих интерфейс.

This so greatly reduces implementation dependencies between subsystems that it leads to the following principle of reusable object-oriented design:

Это так значительно ослабляет зависимости между подсистемами, что приводит к следующему принципу ОО проектирования:

Program to an interface, not an implementation.

Программируйте для интерфейса, а не реализации.

Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class. You will find this to be a common theme of the design patterns in this book.

Не определяйте переменные экземпляры какого-то конкретного класса. Вместо этого используйте только интерфейс определенный абстрактным классом. Вы обнаружите, что это общая тема проектных шаблонов в этой книге.

You have to instantiate concrete classes (that is, specify a particular implementation) somewhere in your system, of course, and the creational patterns (Abstract Factory (87), Builder (97), Factory Method (107), Prototype (117), and Singleton (127) let you do just that. By abstracting the process of object creation, these patterns give you different ways to associate an interface with its implementation transparently at instantiation. Creational patterns ensure that your system is written in terms of interfaces, not implementations.

Конечно вам придется создавать конкретные классы (то есть указывать конкретную реализацию) где-то в вашей системе, и создающие шаблоны (Абстрактная фабрика (87), Строитель (97), Заводской метод (107), Прототип (117) и Одиночка (127) позволяют вам сделать это. Абстрагированием процесса создания объектов эти шаблоны предоставляют вам разные способы прозрачного ассоциирования интерфейса с реализацией на этапе создания. Создающие шаблоны гарантируют, что ваша система написана в терминах интерфейсов, а не реализаций.

1.6.5 Putting Reuse Mechanisms to Work
Применение механизмов многократного использования

Most people can understand concepts like objects, interfaces, classes, and inheritance. The challenge lies in applying them to build flexible, reusable software, and design patterns can show you how.

Большинство людей могут понять концепции вроде объектов, интерфейсов, классов и наследования. Трудность заключается в их применении для создания гибких, многократно используемых программ, и шаблоны проектирования могут показать вам как это сделать.

Inheritance versus Composition
Наследование против Композиции

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and **object composition**. As we've explained, class inheritance lets you define the implementation of one class in terms of another's. Reuse by subclassing is often referred to as **white-box reuse**. The term "white-box" refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.

Две самые распространенные техники для повторного использования функциональности в ОО системах это наследование классов и **композиция объектов**. Как мы объясняли, наследование классов позволяет вам определить реализацию одного класса в терминах другого. Использование созданием подклассов часто называют использованием **белого ящика**. Термин "белый ящик" говорит о видимости: при наследовании внутренности родительских классов часто видны подклассам.

Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or *composing* objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style

of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as “black boxes.”

Альтернативой наследованию является композиция объектов. Здесь новая функциональность получается объединением или *компоновкой* объектов. Композиция объектов требует наличия у них хорошо определенных интерфейсов. Этот стиль использования называется использованием **черного ящика**, поскольку внутреннее состояние объектов не видно. Объекты предстают только в виде “черных ящиков”.

Inheritance and composition each have their advantages and disadvantages. Class inheritance is defined statically at compile-time and is straightforward to use, since it’s supported directly by the programming language. Class inheritance also makes it easier to modify the implementation being reused. When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.

Наследование и композиция имеют свои преимущества и недостатки. Наследование классов определено статически во время компиляции и просто в использовании, поскольку поддерживается непосредственно языком программирования. Наследование также облегчает изменение многократно используемой реализации. Когда подкласс переопределяет некоторые, но не все операции, это может также влиять на операции которые он наследует, если они вызывают переопределенные операции.

But class inheritance has some disadvantages, too. First, you can’t change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time. Second, and generally worse, parent classes often define at least part of their subclasses’ physical representation. Because inheritance exposes a subclass to details of its parent’s implementation, it’s often said that “inheritance breaks encapsulation” [Sny86]. The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent’s implementation will force the subclass to change.

Но наследование имеет также и недостатки. Во-первых, вы не можете изменять реализации унаследованные от родительского класса во время выполнения, потому что наследование определено во время компиляции. Во-вторых, что обычно хуже, родительские классы часто определяют как минимум часть физического представления их подклассов. Поскольку наследование посвящает подкласс в детали реализации его родительского класса, часто говорят, что “наследование нарушает инкапсуляцию” [Sny86]. Реализация подкласса становится так связана с реализацией родительского класса, что любое изменение в родительском классе приведет также и к изменению подкласса.

Implementation dependencies can cause problems when you're trying to reuse a subclass. Should any aspect of the inherited implementation not be appropriate for new problem domains, the parent class must be rewritten or replaced by something more appropriate. This dependency limits flexibility and ultimately reusability. One cure for this is to inherit only from abstract classes, since they usually provide little or no implementation.

Зависимости реализаций могут вызвать проблемы когда вы пытаетесь повторно использовать подкласс. Если какой-либо аспект унаследованной реализации не подходит для новых специфических задач, родительский класс нужно будет переписать, или заменить на что-либо более подходящее. Эта зависимость ограничивает гибкость и многократное использование. Единственное лекарство против этого — наследовать только от абстрактных классов, т.к. в них мало или совсем нет реализации.

Object composition is defined dynamically at run-time through objects acquiring references to other objects. Composition requires objects to respect each others' interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others. But there is a payoff. Because objects are accessed solely through their interfaces, we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type. Moreover, because an object's implementation will be written in terms of object interfaces, there are substantially fewer implementation dependencies.

Композиция объектов определяется динамически во время выполнения через ссылки объектов на другие объекты. Композиция требует, чтобы объекты соблюдали интерфейсы других объектов, что в свою очередь требует тщательно разработанных интерфейсов, которые бы не препятствовали использованию объекта со многими другими. Но она того стоит. Поскольку работа с объектами происходит исключительно через их интерфейсы мы не нарушаем инкапсуляцию. Любой объект может быть заменен на другой во время выполнения пока он имеет тот же тип. Более того, поскольку реализация объекта будет написана в терминах интерфейсов, зависимостей будет значительно меньше.

Object composition has another effect on system design. Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task. Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters. On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.

Композиция объектов имеет другой эффект на проект системы. Предпочтение композиции объектов перед наследованием помогает вам поддерживать инкапсуляцию каждого класса и фокусировать его на одной задаче. Ваши классы и иерархии классов останутся

небольшими и с меньше вероятностью вырастут в огромных неуправляемых монстров. С другой стороны проект основанный на композиции объектов будет иметь больше объектов (т.к. меньше классов), и поведение системы будет зависеть от их взаимоотношений вместо того, чтобы определяться в одном классе.

That leads us to our second principle of object-oriented design:

Это приводит нас ко второму принципу ОО проектирования:

Favor object composition over class inheritance.

Предпочитайте композицию объектов наследованию классов.

Ideally, you shouldn't have to create new components to achieve reuse. You should be able to get all the functionality you need just by assembling existing components through object composition. But this is rarely the case, because the set of available components is never quite rich enough in practice. Reuse by inheritance makes it easier to make new components that can be composed with old ones. Inheritance and object composition thus work together.

В идеале, вы не должны создавать новых компонентов для достижения многократного использования. Вы должны иметь возможность получить всю необходимую функциональность, просто объединяя существующие компоненты, с помощью объектной композиции. Но такие случаи редки, поскольку набор доступных компонент практически никогда не бывает достаточно богат. Многократное использование путем наследования упрощает создание новых компонент, которые могут быть соединены с уже имеющимися. Таким образом, наследование и композиция объектов работают вместе.

Nevertheless, our experience is that designers overuse inheritance as a reuse technique, and designs are often made more reusable (and simpler) by depending more on object composition. You'll see object composition applied again and again in the design patterns.

Однако, согласно нашему опыту проектировщики чаще используют наследование в качестве техники повторного использования, тогда как проекты зачастую лучше многократно используются (и становятся проще), если больше полагаться на композицию объектов. Вы увидите, что композиция объектов снова и снова применяется в шаблонах проектирования.

Delegation
Делегирование

Delegation is a way of making composition as powerful for reuse as inheritance [Lie86, JZ91]. In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**. This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object through

the `this` member variable in C++ and `self` in Smalltalk. To achieve the same effect with delegation, the receiver passes itself to the delegate to let the delegated operation refer to the receiver.

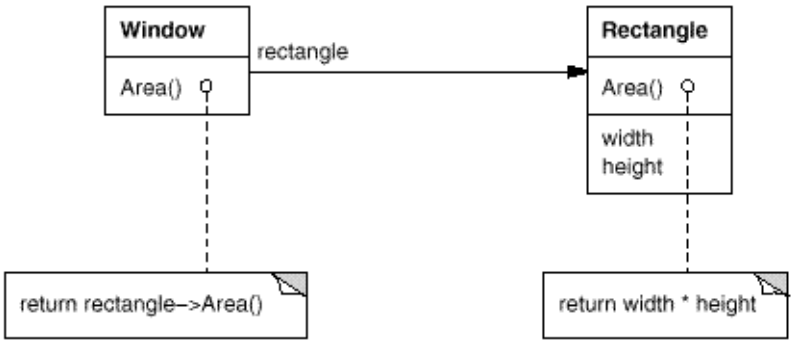
Делегирование — это способ сделать композицию такой же мощной для повторного использования как наследование. При делегировании в обработку запроса вовлечены *два* объекта: получающий объект делегирует операции своему **делегату**. Это похоже на подклассы передающие запросы родительским классам. Но при наследовании, наследуемая операция всегда может сослаться на получающий объект через переменную-член `this` в C++ и `self` в Smalltalk. Для получения аналогичного эффекта при делегировании, получатель передает себя делегату, чтобы позволить делегируемой операции ссылаться на получателя.

For example, instead of making class Window a subclass of Rectangle (because windows happen to be rectangular), the Window class might reuse the behavior of Rectangle by keeping a Rectangle instance variable and *delegating* Rectangle-specific behavior to it. In other words, instead of a Window *being* a Rectangle, it would *have* a Rectangle. Window must now forward requests to its Rectangle instance explicitly, whereas before it would have inherited those operations.

Например, вместо создания класса Окно как подкласса Прямоугольника (поскольку окна чаще всего прямоугольны), класс Окно может использовать поведение Прямоугольника включением экземпляра переменной Прямоугольник и *делегированием* ей поведения специфичного для Прямоугольника. Другими словами, Окно вместо того, чтобы *быть* Прямоугольником, будет *содержать* Прямоугольник. Теперь Окно должно явно передавать запросы экземпляру Прямоугольника, тогда как раньше оно бы наследовало эти операции.

The following diagram depicts the Window class delegating its Area operation to a Rectangle instance.

Следующая диаграмма описывает класс Окно делегирующий операцию Область экземпляру прямоугольника.



A plain arrowhead line indicates that a class keeps a reference to an instance of another class. The reference has an optional name, “rectangle” in this case.

Линия со стрелкой показывает, что класс содержит ссылку на экземпляр другого класса. Ссылка имеет опциональное имя, в данном случае “прямоугольник”

The main advantage of delegation is that it makes it easy to compose behaviors at run-time and to change the way they’re composed. Our window can become circular at run-time simply by replacing its Rectangle instance with a Circle instance, assuming Rectangle and Circle have the same type.

Основное преимущество делегирования в том, что оно облегчает объединение поведения объектов во время выполнения и изменение способа этого объединения. Наше окно может стать круглым во время выполнения просто заменой экземпляра Прямоугольника на экземпляр Окружности, в предположении, что Прямоугольник и Окружность имеют одинаковый тип.

Delegation has a disadvantage it shares with other techniques that make software more flexible through object composition: Dynamic, highly parameterized software is harder to understand than more static software. There are also run-time inefficiencies, but the human inefficiencies are more important in the long run. Delegation is a good design choice only when it simplifies more than it complicates. It isn’t easy to give rules that tell you exactly when to use delegation, because how effective it will be depends on the context and on how much experience you have with it. Delegation works best when it’s used in highly stylized ways – that is, in standard patterns.

У делегирования есть недостаток, который также присущ другим техникам, делающим ПО более гибким через композицию объектов: Динамичное, сильно параметризованное ПО понять труднее, чем более статичное ПО. Также присутствуют программные издержки во время выполнения, но человеческие издержки более важны в долгосрочной перспективе. Делегирование является хорошим проектным выбором, только когда оно упрощает больше, чем усложняет. Непросто дать правила, которые точно говорят когда

использовать делегирование, поскольку его эффективность зависит от контекста и вашего опыта его использования. Делегирование работает лучше всего при использовании хорошо определенными способами — то есть в стандартных шаблонах.

Several design patterns use delegation. The State (305), Strategy (315), and Visitor (331) patterns depend on it. In the State pattern, an object delegates requests to a State object that represents its current state. In the Strategy pattern, an object delegates a specific request to an object that represents a strategy for carrying out the request. An object will only have one state, but it can have many strategies for different requests. The purpose of both patterns is to change the behavior of an object by changing the objects to which it delegates requests. In Visitor, the operation that gets performed on each element of an object structure is always delegated to the Visitor object.

Несколько шаблонов проектирования используют делегирование. Состояние (305), Стратегия (315), Посетитель (331) зависят от него. В шаблоне Состояние объект делегирует запросы объекту Состояние, который представляет его текущее состояние. В шаблоне Стратегия, объект делегирует специфический запрос объекту, который представляет стратегию для выполнения запроса. Объект будет иметь только одно состояние, но он может иметь множество стратегий для различных запросов. Цель обоих шаблонов — изменить поведение объекта, заменяя объекты, которым он делегирует запросы. В Посетителе действие производимое над каждым элементов структуры объекта всегда делегируется объекту Посетитель.

Other patterns use delegation less heavily. Mediator (273) introduces an object to mediate communication between other objects. Sometimes the Mediator object implements operations simply by forwarding them to the other objects; other times it passes along a reference to itself and thus uses true delegation. Chain of Responsibility (223) handles requests by forwarding them from one object to another along a chain of objects. Sometimes this request carries with it a reference to the original object receiving the request, in which case the pattern is using delegation. Bridge (151) decouples an abstraction from its implementation. If the abstraction and a particular implementation are closely matched, then the abstraction may simply delegate operations to that implementation.

Остальные шаблоны меньше используют делегирование. Посредник (273) представляет объект-посредник между другими взаимодействующими объектами. Иногда объект Посредник реализует операции, просто передавая их другим объектам; в других случаях он передает ссылку на себя самого и таким образом использует настоящее делегирование. Цепь ответственности (223) обрабатывает запросы, передавая их от одного объекта к другому по цепочке. Иногда этот запрос передается вместе со ссылкой на исходный объект,

получающий запрос, в таком случае шаблон использует делегирование. Мост (151) отделяет абстракцию от реализации. Если абстракция и соответствующая реализация тесно согласованы, то абстракция может просто делегировать операции этой реализации.

Delegation is an extreme example of object composition. It shows that you can always replace inheritance with object composition as a mechanism for code reuse.

Делегирование это экстремальный пример композиции объектов. Он показывает, что вы всегда можете заменить наследование на композицию объектов в качестве механизма многократного использования кода.

Inheritance versus Parameterized Types
Наследование против параметризованных типов

Another (not strictly object-oriented) technique for reusing functionality is through **parameterized types**, also known as **generics** (Ada, Eiffel) and **templates** (C++). This technique lets you define a type without specifying all the other types it uses. The unspecified types are supplied as *parameters* at the point of use. For example, a List class can be parameterized by the type of elements it contains. To declare a list of integers, you supply the type "integer" as a parameter to the List parameterized type. To declare a list of String objects, you supply the "String" type as a parameter. The language implementation will create a customized version of the List class template for each type of element.

Другая (не строго объектно-ориентированная) техника для многократного использования функциональности это **параметризованные типы**, также известные, как **родовые** (generics) (Ada, Eiffel) и **шаблоны** (templates) (C++). Эта техника позволяет вам определять тип без указания всех типов которые он использует. Неуказанные типы передаются в качестве *параметров* в точке использования. Например класс Список может быть параметризован типом содержащихся в нем элементов. Для определения списка целых чисел, вы предоставляете тип "целое" как параметр для параметризованного типа Список. Для определения списка строк вы используете тип "Строка" в качестве параметра. Реализация языка создаст соответствующую версию шаблона класса Список для каждого типа элемента.

Parameterized types give us a third way (in addition to class inheritance and object composition) to compose behavior in object-oriented systems. Many designs can be implemented using any of these three techniques. To parameterize a sorting routine by the operation it uses to compare elements, we could make the comparison

Параметризованные типы дают нам третий способ (в добавку к наследованию классов и композиции объектов) для компоновки поведения в ОО системах. Многие проекты

могут быть реализованы с использованием любого из этих способов. Для параметризования процедуры сортировки операцией которая сравнивает элементы мы можем сделать сравнение:

1. an operation implemented by subclasses (an application of Template Method (325),

   операцией реализованной подклассами (используем Шаблонный Метод) (325)

2. the responsibility of an object that's passed to the sorting routine (Strategy (315), or

   ответственностью объекта передаваемого в процедуру сортировки (Стратегия (315) или

3. an argument of a C++ template or Ada generic that specifies the name of the function to call to compare the elements.

   аргументом шаблона C++ или родового типа Ada, который указывает имя функции вызываемой для сравнения элементов.

There are important differences between these techniques. Object composition lets you change the behavior being composed at run-time, but it also requires indirection and can be less efficient. Inheritance lets you provide default implementations for operations and lets subclasses override them. Parameterized types let you change the types that a class can use. But neither inheritance nor parameterized types can change at run-time. Which approach is best depends on your design and implementation constraints.

Существуют важные различия между этими техниками. Композиция объектов позволяет вам изменять создаваемое поведение во время выполнения, но также требует косвенности и может быть менее эффективна. Наследование позволяет вам предоставить реализации операций по умолчанию и позволяет классам переопределять их. Параметризованные типы позволяют вам изменять типы используемые классом. Но ни наследование, ни параметризованные типы не могут изменяться во время выполнения. Какой подход лучше зависит от ограничений проекта и реализации.

None of the patterns in this book concerns parameterized types, though we use them on occasion to customize a pattern's C++ implementation. Parameterized types aren't needed at all in a language like Smalltalk that doesn't have compile-time type checking.

Ни один из шаблонов в этой книге не использует параметризованные типы, хотя мы используем их при случае для указания реализации шаблона на C++. Параметризованные типы не нужны в языках вроде Smalltalk, которые не проводят проверку типов во время компиляции.

1.6.6 Relating Run-Time and Compile-Time Structures
Соотношение структур времени компиляции и времени выполнения

An object-oriented program's run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.

Структура объектно-ориентированной программы во время выполнения зачастую несет слабый отпечаток структуры ее кода. Структура кода заморожена во время выполнения; она состоит из классов жестко связанных отношениями наследования. Структура программы во время выполнения состоит из быстро изменяющихся сетей взаимодействующих объектов. Действительно эти две структуры практически независимы. Попытка понять одну по другой подобна попытке понять динамику живых экосистем по статической анатомии растений и животных и наоборот.

Consider the distinction between object **aggregation** and **acquaintance** and how differently they manifest themselves at compile- and run-times. Aggregation implies that one object owns or is responsible for another object. Generally we speak of an object *having* or being *part of* another object. Aggregation implies that an aggregate object and its owner have identical lifetimes.

Рассмотрим различие между **агрегацией** и **взаимодействием** объектов и как они различно обнаруживают себя во время компиляции и во время выполнения. Агрегация подразумевает, что один объект владеет или является ответственным за другой объект. В общем мы говорим объекте являющемся *частью* другого объекта. Агрегация подразумевает, что агрегируемый объект и его владелец имеют одинаковое время существования.

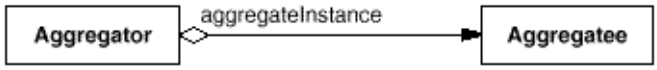
Acquaintance implies that an object merely knows of another object. Sometimes acquaintance is called "association" or the "using" relationship. Acquainted objects may request operations of each other, but they aren't responsible for each other. Acquaintance is a weaker relationship than aggregation and suggests much looser coupling between objects.

Взаимодействие подразумевает что объект только *знает* о другом объекте. Иногда взаимодействие называется отношением "ассоциирования" или "использования". Взаимодействующие объекты могут запрашивать операции друг друга, но они не отвечают друг

за друга. Отношение взаимодействия слабее агрегации и предполагает гораздо меньшую связь между объектами.

In our diagrams, a plain arrowhead line denotes acquaintance. An arrowhead line with a diamond at its base denotes aggregation:

В наших диаграммах простая линия со стрелкой означает взаимодействие. Стрелка с ромбиком у основания означает агрегацию:



It's easy to confuse **aggregation** and **acquaintance**, because they are often implemented in the same way. In Smalltalk, all variables are references to other objects. There's no distinction in the programming language between aggregation and acquaintance. In C++, aggregation can be implemented by defining member variables that are real instances, but it's more common to define them as pointers or references to instances. Acquaintance is implemented with pointers and references as well.

Легко перепутать **агрегацию** и **взаимодействие**, поскольку они часто реализуются одинаково. В Smalltalk все переменные являются ссылками на другие объекты. В языке нет различия между агрегацией и взаимодействием. В C++ агрегация может быть реализована определение переменных-членов, которые являются непосредственными экземплярами, но более общепринятая практика определять их как ссылки или указатели на экземпляры. Взаимодействие также реализуется с помощью указателей и ссылок.

Ultimately, acquaintance and aggregation are determined more by intent than by explicit language mechanisms. The distinction may be hard to see in the compile-time structure, but it's significant. Aggregation relationships tend to be fewer and more permanent than acquaintance. Acquaintances, in contrast, are made and remade more frequently, sometimes existing only for the duration of an operation. Acquaintances are more dynamic as well, making them more difficult to discern in the source code.

В конце концов взаимодействие и агрегация определяются больше намерениями, чем явными языковыми механизмами. Различие может быть трудно увидеть в структуре времени компиляции, но оно значительно. Отношений агрегации обычно меньше и они более постоянны. Наоборот, взаимодействия создаются и переделываются более часто, иногда существуя только во время выполнения операции. Также взаимодействия более динамичны, что усложняет их выявление в исходном коде.

With such disparity between a program's run-time and compile-time structures, it's clear that code won't reveal everything about how a system will work. The system's run-time

structure must be imposed more by the designer than the language. The relationships between objects and their types must be designed with great care, because they determine how good or bad the run-time structure is.

Учитывая такое несоответствие между программными структурами времени компиляции и времени выполнения, ясно что код не расскажет всего о том как будет работать система. Структура системы во время выполнения должна больше определяться проектировщиком, чем языком. Отношения между объектами и их типами должны разрабатываться очень осторожно, потому что они определяют насколько хороша или плоха структура времени выполнения.

Many design patterns (in particular those that have object scope) capture the distinction between compile-time and run-time structures explicitly. Composite (163) and Decorator (175) are especially useful for building complex run-time structures. Observer (293) involves run-time structures that are often hard to understand unless you know the pattern. Chain of Responsibility (223) also results in communication patterns that inheritance doesn't reveal. In general, the run-time structures aren't clear from the code until you understand the patterns.

Многие шаблоны проектирования (в особенности применяемые к объектам) явно фиксируют различие между структурами времени выполнения и времени компиляции. Композит (163) и Декоратор (175) особенно полезны для построения сложных структур времени выполнения. Наблюдатель (293) включает структуры времени выполнения, которые зачастую трудно понять, если вы не знаете шаблона. Цепь ответственности (223) также приводит к взаимодействующим шаблонам, которые не выявляются наследованием. В общем структуры времени выполнения непонятны из кода, пока вы не поймете шаблоны.

1.6.7 Designing for Change Разработка для изменения

The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.

Ключ к максимизации повторного использования лежит в предвидении новых требований и изменений к существующим требованиям, и в разработке ваших систем так, чтобы они могли развиваться соответственно изменениям.

To design the system so that it's robust to such changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future. Those changes might involve class redefinition and reimplementation, client modification, and retesting. Redesign affects many parts of the software system, and unanticipated changes are invariably expensive.

Для разработки системы устойчивой к подобным изменениям вы должны предположить, как система может измениться в течении своего существования. Проект не принимающий в расчет изменения рискует глобально перерабатываться в будущем. Эти изменения могут включать переопределение классов и их реализаций, изменение клиентов и новое тестирование. Переработка влияет на многие части программной системы и непредвиденные изменения неизменно дороги.

Design patterns help you avoid this by ensuring that a system can change in specific ways. Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.

Проектные шаблоны помогают вам избежать этого, гарантируя, что система может изменяться в определенных направлениях. Каждый проектный шаблон позволяет какому либо аспекту структуры системы изменяться независимо от других аспектов, тем самым делая систему более устойчивой к определенным видам изменений.

Here are some common causes of redesign along with the design pattern(s) that address them:

Здесь приведены некоторые наиболее общие причины перепроектирования вместе с проектными шаблонами для таких случаев:

1. *Creating an object by specifying a class explicitly.* Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface. This commitment can complicate future changes. To avoid it, create objects indirectly.

   Явное указание класса при создании объекта. Указание имени класса при создании объекта привязывает вас к определенной реализации вместо определенного интерфейса. Эта привязка может усложнить будущие изменения. Во избежание этого создавайте объекты косвенно.

   Design patterns: Abstract Factory (87), Factory Method (107), Prototype (117).

   Проектные шаблоны: Абстрактная Фабрика (87), Заводской Метод (107), Прототип (117).

2. *Dependence on specific operations.* When you specify a particular operation, you commit to one way of satisfying a request. By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and at run-time.

   Зависимость от определенных операций. Когда вы указываете конкретную операцию, вы привязываетесь к одному способу выполнения запроса. Избегая жестко

заданных запросов, вы облегчаете изменение способа выполнения запроса как в во время компиляции, так и во время выполнения.

Design patterns: Chain of Responsibility (223), Command (233).

Проектные шаблоны: Цепь ответственности (223), Команда (233).

3. *Dependence on hardware and software platform.* External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms. Software that depends on a particular platform will be harder to port to other platforms. It may even be difficult to keep it up to date on its native platform. It's important therefore to design your system to limit its platform dependencies.

   Зависимость от программной или аппаратной платформы. Внешние интерфейсы операционной системы или интерфейсы программирования приложений (APIs) различны на разных программных и аппаратных платформах. ПО зависящее от конкретной платформы будет труднее переносить на другие платформы. Возможно его будет сложно обновлять на его родной платформе. Поэтому важно так проектировать вашу систему, чтобы ограничить ее зависимости от конкретной платформы.

   Design patterns: Abstract Factory (87), Bridge (151).

   Проектные шаблоны: Абстрактная Фабрика (87), Мост (151).

4. *Dependence on object representations or implementations.* Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes. Hiding this information from clients keeps changes from cascading.

   Зависимость от представлений и реализаций объекта. Клиенты, которые знают как объект представляется, где хранится и находится или как реализуется могут потребовать изменения при изменении объекта. Сокрытие этой информации от клиентов удерживает изменения от каскадного увеличения.

   Design patterns: Abstract Factory (87), Bridge (151), Memento (283), Proxy (207).

   Проектные шаблоны: Абстрактная Фабрика (87), Мост (151), Память (283), Полномочия (207).

5. *Algorithmic dependencies.* Algorithms are often extended, optimized, and replaced during development and reuse. Objects that depend on an algorithm will have to change when the algorithm changes. Therefore algorithms that are likely to change should be isolated.

Алгоритмические зависимости. Алгоритмы часто расширяются, оптимизируются и заменяются во время разработки и использования. Объекты, зависящие от алгоритма должны будут измениться при его изменении. Поэтому алгоритмы наиболее подверженные изменению должны быть изолированы.

Design patterns: Builder (97), Iterator (257), Strategy (315), Template Method (325), Visitor (331).

Проектные шаблоны: Строитель (97), Итератор (257), Стратегия (315), Шаблонный метод (325), Посетитель (331).

6. *Tight coupling.* Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. The system becomes a dense mass that's hard to learn, port, and maintain. Loose coupling increases the probability that a class can be reused by itself and that a system can be learned, ported, modified, and extended more easily. Design patterns use techniques such as abstract coupling and layering to promote loosely coupled systems.

   Тесное сцепление. Тесно сцепленные классы трудно использовать изолированно, поскольку они зависят друг от друга. Тесное сцепление ведет к монолитным системам, в которых вы не можете изменить или убрать класс без понимания и изменения многих других классов. Система становится тяжелой массой, которую трудно изучать, переносить и поддерживать. Ослабление сцепления увеличивает вероятность того, что класс может быть многократно использован самостоятельно, и что система сможет быть изучена, перенесена и модифицирована гораздо проще. Проектные шаблоны используют такие техники как абстрактное сцепление и наслоение для получение слабо сцепленных систем.

   Design patterns: Abstract Factory (87), Bridge (151), Chain of Responsibility (223), Command (233), Facade (185), Mediator (273), Observer (293).

   Проектные шаблоны: Абстрактная фабрика (87), Мост (151), Цепь ответственности (223), Команда (233), Фасад (185), Посредник (273), Наблюдатель (293).

7. *Extending functionality by subclassing.* Customizing an object by subclassing often isn't easy. Every new class has a fixed implementation overhead (initialization, finalization, etc.). Defining a subclass also requires an in-depth understanding of the parent class. For example, overriding one operation might require overriding another. An overridden operation might be required to call an inherited operation. And subclassing can lead to

an explosion of classes, because you might have to introduce many new subclasses for even a simple extension.

Расширение функциональности сабклассингом. Настройка объекта сабклассингом часто непроста. Каждый новый класс имеет фиксированные издержки в реализации (инициализация, завершение, и т.д.). Определение подкласса также требует глубокого понимания его родительского класса. Например переопределение одной операции может потребовать переопределения другой. Переопределенная операция возможно должна будет вызвать унаследованную операцию. Также сабклассинг может привести к взрывному увеличению числа классов, поскольку вам придется добавлять много новых подклассов даже для простого расширения.

Object composition in general and delegation in particular provide flexible alternatives to inheritance for combining behavior. New functionality can be added to an application by composing existing objects in new ways rather than by defining new subclasses of existing classes. On the other hand, heavy use of object composition can make designs harder to understand. Many design patterns produce designs in which you can introduce customized functionality just by defining one subclass and composing its instances with existing ones.

Композиция объектов вообще и делегирование в частности предоставляют гибкие альтернативы наследованию для комбинирования поведения. Новая функциональность может быть добавлена в приложение соединением существующих объектов новыми способами вместо определения новых подклассов существующих классов. С другой стороны большое использование композиции объектов может усложнить понимание проектов. Многие шаблоны проектирования производят проекты в которых вы можете добавить функциональность просто определением одного подкласса и соединением его экземпляров с уже существующими классами.

Design patterns: Bridge (151), Chain of Responsibility (223), Composite (163), Decorator (175), Observer (293), Strategy (315).

Проектные шаблоны: Мост (151), Цепь ответственности (223), Композит(163), Декоратор (175), Наблюдатель (293), Стратегия (315).

8. *Inability to alter classes conveniently.* Sometimes you have to modify a class that can't be modified conveniently. Perhaps you need the source code and don't have it (as may be the case with a commercial class library). Or maybe any change would require modifying lots of existing subclasses. Design patterns offer ways to modify classes in such circumstances.

Невозможность простого изменения классов. Иногда вам нужно изменить класс, который не может быть модифицирован просто. Возможно вам нужен исходный код, а у вас его нет (как например в случае коммерческой библиотеки классов). Или любое изменение потребует множества модификаций существующих подклассов. Проектные шаблоны предлагают способы изменения классов в таких условиях.

Design patterns: Adapter (139), Decorator (175), Visitor (331).

Проектные шаблоны: Адаптер (139), Декоратор (175), Посетитель (331).

These examples reflect the flexibility that design patterns can help you build into your software. How crucial such flexibility is depends on the kind of software you're building. Let's look at the role design patterns play in the development of three broad classes of software: application programs, toolkits, and frameworks.

Эти примеры показывают гибкость, которую проектные шаблоны помогают встроить в ваше ПО. Насколько критична такая гибкость зависит от типа ПО которое вы создаете. Давайте посмотрим на роль проектных шаблонов в разработке трех широких классов ПО: прикладных программ, пакетов разработчика и сред разработки.

Application Programs Прикладные программы

If you're building an application program such as a document editor or spreadsheet, then internal reuse, maintainability, and extension are high priorities. Internal reuse ensures that you don't design and implement any more than you have to. Design patterns that reduce dependencies can increase internal reuse. Looser coupling boosts the likelihood that one class of object can cooperate with several others. For example, when you eliminate dependencies on specific operations by isolating and encapsulating each operation, you make it easier to reuse an operation in different contexts. The same thing can happen when you remove algorithmic and representational dependencies too.

Если вы создаете прикладную программу, такую как редактор документов или электронную таблицу, то внутреннее многократное использование, сопровождаемость, и расширяемость имеют высокий приоритет. Внутреннее многократное использование гарантирует, что вы не проектируете и не реализовываете больше чем вы должны. Проектные шаблоны, которые уменьшают зависимости, могут увеличить внутреннее многократное использование. Ослабление сцепления увеличивает вероятность того, что один класс или объект сможет сотрудничать с несколькими другими. Например, когда вы устраните зависимости определенных операций изолировав и инкапсулировав каждую операцию, вы,

тем самым, упростите многократное использование операции в разных контекстах. То же самое может случиться если вы также удалите зависимости алгоритмов и представлений.

Design patterns also make an application more maintainable when they're used to limit platform dependencies and to layer a system. They enhance extensibility by showing you how to extend class hierarchies and how to exploit object composition. Reduced coupling also enhances extensibility. Extending a class in isolation is easier if the class doesn't depend on lots of other classes.

Проектные шаблоны также делают приложение более сопровождаемым, когда они используются для ограничения зависимостей от платформы и расслоения системы. Они улучшают расширяемость, показывая как расширить иерархии классов и как использовать объектную композицию. Уменьшенное сцепление также улучшает расширяемость. Расширение класса изолированно проще, если класс не зависит от многих других классов.

Toolkits Пакеты разработчика

Often an application will incorporate classes from one or more libraries of predefined classes called toolkits. A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. An example of a toolkit is a set of collection classes for lists, associative tables, stacks, and the like. The C++ I/O stream library is another example. Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do its job. They let you as an implementer avoid recoding common functionality. Toolkits emphasize *code reuse*. They are the object-oriented equivalent of subroutine libraries.

Часто приложение включает классы из одной или более библиотек классов, называемых пакетами разработчика. Пакет разработчика это набор взаимосвязанных классов, разработанных с целью предоставления полезной, целевой функциональности. Примером пакета разработчика может быть набор классов для списков, ассоциативных таблиц, стеков и тому подобного. Другой пример — потоковая библиотека ввода/вывода в C++. Пакеты не навязывают определенную структуру проекта вашего приложения; они просто предоставляют функции, которые могут помочь вашему приложению делать свою работу. Они позволяют вам, как программисту избежать повторного кодирования общей функциональности. Пакеты разработчика делают ударение на *многократном использовании кода*. Они являются ОО эквивалентом библиотек подпрограмм.

Toolkit design is arguably harder than application design, because toolkits have to work in many applications to be useful. Moreover, the toolkit writer isn't in a position to know

what those applications will be or their special needs. That makes it all the more important to avoid assumptions and dependencies that can limit the toolkit's flexibility and consequently its applicability and effectiveness.

Разработка пакетов разработчика бесспорно труднее, чем разработка прикладных программ, поскольку для того, чтобы быть полезными, пакеты разработчика должны работать во многих приложениях. Более того, разработчик пакета не знает что это будут за приложения, и каковы будут их специфические требования. Это делает наиболее важным избежание предположений и зависимостей, которые могут ограничить гибкость пакета и, следовательно, его применимость и эффективность.

Frameworks Среды разработки

A framework is a set of cooperating classes that make up a reusable design for a specific class of software [Deu89, JF88]. For example, a framework can be geared toward building graphical editors for different domains like artistic drawing, music composition, and mechanical CAD [VL90, Joh92]. Another framework can help you build compilers for different programming languages and target machines [JML92]. Yet another might help you build financial modeling applications [BE93]. You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework.

Среда разработки это набор взаимодействующих классов, которые составляют многократно используемый проект для специфического типа программного обеспечения [Deu89, JF88]. Например, среда разработки может быть сделана для построения графических редакторов для различных областей применения таких, как художественное рисование, сочинение музыки и механические САПР [VL90, Joh92]. Другая среда разработки может помочь вам создавать компиляторы для разных языков программирования и целевых платформ [JML92]. Еще одна помогает в создании приложения для финансового моделирования [BE93]. Вы настраиваете среду разработки для определенного приложения созданием специфических для него подклассов существующих абстрактных классов.

The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. A framework predefines these design parameters so that you, the application designer/implementer, can concentrate on the specifics of your application. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize *design reuse* over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

Среда разработки диктует архитектуру вашего приложения. Она определит общую структуру, ее разделение на классы и объекты и, следовательно, ключевые ответственности, взаимодействие классов и объектов, поток управления. Среда разработки предопределяет эти проектные параметры, так чтобы вы, проектировщик/программист приложения, могли сконцентрироваться на специфике вашего приложения. Среда разработки фиксирует общие проектные решения для своей предметной области. Таким образом, среды разработки подчеркивают преобладание *многократного использования проектных решений* над многократным использованием кода, хотя среда разработки обычно включает конкретные классы, которые вы сразу же можете использовать.

Reuse on this level leads to an inversion of control between the application and the software on which it's based. When you use a toolkit (or a conventional subroutine library for that matter), you write the main body of the application and call the code you want to reuse. When you use a framework, you reuse the main body and write the code *it* calls. You'll have to write operations with particular names and calling conventions, but that reduces the design decisions you have to make.

Многократное использование на этом уровне ведет к инверсии управления между приложением и программным обеспечением на котором оно основано. Когда вы используете пакет разработчика (или стандартную библиотеку подпрограмм), вы пишите основное тело приложения и вызываете код, который хотите использовать. Когда вы используете среду разработки, вы многократно используете основное тело и пишите код, который *оно* вызывает. Вам придется писать операции с определенными именами и условиями вызова, но это уменьшает количество проектных решений, которые вам нужно принять.

Not only can you build applications faster as a result, but the applications have similar structures. They are easier to maintain, and they seem more consistent to their users. On the other hand, you lose some creative freedom, since many design decisions have been made for you.

В результате вы не только можете строить приложения быстрее, но также эти приложения имеют похожую структуру. Их проще сопровождать и они более единообразны для пользователей. С другой стороны, вы теряете некоторую свободу творчества, т.к. многие проектные решения были сделаны за вас.

If applications are hard to design, and toolkits are harder, then frameworks are hardest of all. A framework designer gambles that one architecture will work for all applications in the domain. Any substantive change to the framework's design would reduce its benefits considerably, since the framework's main contribution to an application is the architecture it

defines. Therefore it's imperative to design the framework to be as flexible and extensible as possible.

Если приложения разрабатывать трудно, а пакеты разработчика труднее, то среды разработки труднее всего. Проектировщик среды разработки ставит на то, что одна архитектура подойдет для всех приложений в предметной области. Любое существенное изменение в проекте среды разработки значительно уменьшит его преимущества, т.к. основной вклад среды разработки в приложение — это архитектура, которую она определяет. Следовательно, необходимо проектировать среду разработки так, чтобы она была гибкой и расширяемой насколько возможно.

Furthermore, because applications are so dependent on the framework for their design, they are particularly sensitive to changes in framework interfaces. As a framework evolves, applications have to evolve with it. That makes loose coupling all the more important; otherwise even a minor change to the framework will have major repercussions.

Кроме того, поскольку приложения так зависят от среды разработки, в проекте, они особенно чувствительны к изменениям в ее интерфейсах. В ходе развития среды разработки приложения вынуждены развиваться вместе с ней. Это увеличивает важность слабого сцепления; в противном случае даже небольшое изменение в среде разработки будет иметь серьезные последствия.

The design issues just discussed are most critical to framework design. A framework that addresses them using design patterns is far more likely to achieve high levels of design and code reuse than one that doesn't. Mature frameworks usually incorporate several design patterns. The patterns help make the framework's architecture suitable to many different applications without redesign.

Обсужденные выше вопросы проектирования наиболее критичны для проектирования среды разработки. Среда разработки, которая решает их, используя шаблоны проектирования, с большей вероятностью достигнет высокого уровня проектирования и многократного использования кода, чем та, которая не использует шаблоны. Зрелые среды разработки обычно включают несколько проектных шаблонов. Шаблоны помогают сделать архитектуру среды разработки подходящей для многих приложений без перепроектирования.

An added benefit comes when the framework is documented with the design patterns it uses [BJ94]. People who know the patterns gain insight into the framework faster. Even people who don't know the patterns can benefit from the structure they lend to the framework's documentation. Enhancing documentation is important for all types of software, but it's particularly important for frameworks. Frameworks often pose a steep learning curve that must be overcome before they're useful. While design patterns might not flatten the learning

curve entirely, they can make it less steep by making key elements of the framework's design more explicit.

Дополнительная выгода получается, когда среда разработки документирована с помощью используемых ей проектных шаблонов [BJ94]. Люди знающие шаблоны быстрее осваивают среды разработки. Даже люди, которые не знают шаблонов, могут выиграть от структуры которую шаблоны привносят в документацию среды разработки. Улучшение документации важно для всех видов ПО, но особенно важно для сред разработки. Среды разработки часто имеют крутую кривую обучения, которая должна быть преодолена прежде чем они станут полезны. Хотя проектные шаблоны не могут полностью сгладить кривую обучения, они могут сделать ее менее крутой за счет более явного представления ключевых элементов среды разработки.

Because patterns and frameworks have some similarities, people often wonder how or even if they differ. They are different in three major ways:

Поскольку шаблоны и среды разработки имеют некоторые общие черты, люди часто интересуются чем они отличаются, и отличаются ли вообще. Они различаются в трех основных вопросах:

1. *Design patterns are more abstract than frameworks.* Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, the design patterns in this book have to be implemented each time they're used. Design patterns also explain the intent, trade-offs, and consequences of a design.

   Шаблоны проектирования более абстрактны, чем среды разработки. Среды разработки могут быть воплощены в коде, тогда как только *примеры* шаблонов могут быть воплощены в коде. Сила сред разработки в том, что они могут быть написаны на языках программирования, и не только изучены, но и напрямую выполнены и многократно использованы. В противоположность этому шаблоны проектирования из этой книги должны каждый раз реализовываться, когда необходимо их использовать. Шаблоны проектирования также объясняют намерения, выгоды и последствия проекта.

2. *Design patterns are smaller architectural elements than frameworks.* A typical framework contains several design patterns, but the reverse is never true.

Шаблоны проектирования являются меньшими архитектурными элементами чем среды разработки. Типичная среда разработки содержит несколько проектных шаблонов, обратное же никогда не верно.

3. *Design patterns are less specialized than frameworks.* Frameworks always have a particular application domain. A graphical editor framework might be used in a factory simulation, but it won't be mistaken for a simulation framework. In contrast, the design patterns in this catalog can be used in nearly any kind of application. While more specialized design patterns than ours are certainly possible (say, design patterns for distributed systems or concurrent programming), even these wouldn't dictate an application architecture like a framework would.

   Проектные шаблоны менее специализированы чем среды разработки. Среды разработки всегда имеют определенную область применения. Среда разработки графического редактора может быть использована в моделировании фабрики, но ее не спутаешь со средой разработки для моделирования. Наоборот, проектные шаблоны в этом каталоге могут быть использованы почти в любых видах приложений. В то же время вполне возможны шаблоны более специализированные чем наши (скажем шаблоны проектирования для распределенных систем или параллельного программирования), даже они не будут диктовать архитектуру приложения так как среда разработки.

Frameworks are becoming increasingly common and important. They are the way that object-oriented systems achieve the most reuse. Larger object-oriented applications will end up consisting of layers of frameworks that cooperate with each other. Most of the design and code in the application will come from or be influenced by the frameworks it uses.

Среды разработки становятся все более важны и широкоупотребимы. Они являются способом с помощью которого ОО системы достигают наибольшей степени многократного исполльзования. БОльшие ОО приложения заканчивают как набор слоев взаимодействующих сред разработки. Большая часть проекта и кода в приложении прийдет из используемых сред разработки, или подвергнется их влиянию.

1.7 How to Select a Design Pattern.
Как выбрать проектный шаблон.

With more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you. Here are several different approaches to finding the design pattern that's right for your problem:

При наличии более 20 шаблонов в каталоге, поиск того, который предназначен для конкретной проектной задачи может быть затруднен, особенно если каталог незнаком вам. Далее приведены несколько разных подходов к поиску проектного шаблона подходящего к вашей задаче:

1. *Consider how design patterns solve design problems.* Section 1.6 discusses how design patterns help you find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems. Referring to these discussions can help guide your search for the right pattern.

   Посмотрите как проектные шаблоны решают проектные задачи. В разделе 1.6 обсуждается как проектные шаблоны помогают вам в поиске подходящих объектов, определяют размер объектов, определяют интерфейсы объектов и несколько других путей в которых шаблоны проектирования решают проектные задачи. Ссылки на эти обсуждения могут направить ваш поиск подходящего шаблона.

2. *Scan Intent sections.* Section 1.4 (page 18) lists the Intent sections from all the patterns in the catalog. Read through each pattern's intent to find one or more that sound relevant to your problem. You can use the classification scheme presented in Table 1.1 (page 23) to narrow your search.

   Просмотрите параграфы Намерение. В разделе 1.4 (стр. 18) перечисляются параграфы Намерение всех шаблонов в каталоге. Прочтите намерения каждого шаблона для того чтобы найти одно или несколько намерений звучащих подходяще к вашей задаче. Вы можете использовать схему классификации представленную в Таблице 1.2 (страница 24) для сужения области поиска.

3. *Study how patterns interrelate.* Figure 1.1 (page 27) shows relationships between design patterns graphically. Studying these relationships can help direct you to the right pattern or group of patterns.

Изучите как шаблоны соотносятся. Рисунок 1.1 (страница 27) графически показывает взаимосвязи между шаблонами. Изучение этих взаимосвязей может направить вас на нужный шаблон или группу шаблонов.

4. *Study patterns of like purpose.* The catalog (page 79) has three chapters, one for creational patterns, another for structural patterns, and a third for behavioral patterns. Each chapter starts off with introductory comments on the patterns and concludes with a section that compares and contrasts them. These sections give you insight into the similarities and differences between patterns of like purpose.

   Изучите шаблоны схожего назначения. Каталог (страница 79) содержит 3 главы, одна для создающих шаблонов, другая для структурных шаблонов и третья для поведенческих шаблонов. Каждая глава начинается со вступительных комментариев о шаблонах и завершается параграфом, который сравнивает и противопоставляет их. Эти параграфы дают вам понимание сходств и различий между шаблонами имеющими схожие цели.

5. *Examine a cause of redesign.* Look at the causes of redesign starting on page 53 to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign.

   Проверьте причину перепроектирования. Просмотрите причины перепроектирования на странице 53 возможно ваша задача включают одну или несколько из них. Затем посмотрите шаблоны помогающие избегать причин перепроектирования.

6. *Consider what should be variable in your design.* This approach is the opposite of focusing on the causes of redesign. Instead of considering what might *force* a change to a design, consider what you want to be *able* to change without redesign. The focus here is on *encapsulating the concept that varies*, a theme of many design patterns. Table 1.3 lists the design aspect(s) that design patterns let you vary independently, thereby letting you change them without redesign.

   Определите что в вашем проекте должно быть меняющимся. Этот подход противоположен фокусировке на причинах перепроектирования. Вместо рассмотрения того что может *потребовать* изменений в проекте, рассмотрите что бы вы хотели видеть изменяющимся без перепроектирования. Здесь упор делается на *инкапсулировании меняющейся концепции*, это тема многих проектных шаблонов. Таблица 1.4 перечисляет аспекты проекта которые изменяются с помощью шаблонов проектирования, тем самым позволяя изменять их без перепроектирования.

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Таблица 1.3: Design aspects that design patterns let you vary

Назначение	**Проектный шаблон**	**Аспект(ы), которые могут изменяться**
Создающие	Абстрактная фабрика (87)	семейства результирующих объектов
	Строитель (97)	как составные объекты создаются
	Заводской метод (107)	подкласс создаваемого объекта
	Прототип (117)	класс создаваемого объекта
	Одиночка (127)	единственный экземпляр класса
Структурные	Адаптер (139)	интерфейс объекта
	Мост (151)	реализация объекта
	Композит (163)	структура и композиция объекта
	Декоратор (175)	ответственности объекта без сабклассинга
	Фасад (185)	интерфейс подсистемы
	Легковес (195)	издержки хранения объектов
	Полномочия (207)	осуществляение доступа к объекту; его положение
Поведенческие	Цепь ответственности (223)	объект, который может выполнить запрос
	Команда (233)	когда и как запрос был выполнен
	Интерпретатор (243)	грамматика и интерпретация языка
	Итератор (257)	доступ к элементам агрегата
	Посредник (273)	как и какие объекты взаимодействуют друг с другом
	Память (283)	какая закрытая информация хранится вне объекта и когда
	Наблюдатель (293)	число объектов зависящих от другого объекта; как зависимые объекты поддерживаются в актуальном состоянии
	Состояние (305)	состояния объекта
	Стратегия (315)	алгоритм
	Шаблонный метод (325)	шаги алгоритма
	Посетитель (331)	операции, которые могут быть произведены над объектом (объектами) без изменения его класса (классов)

Таблица 1.4: Аспекты проекта, которые проектные шаблоны позволяют вам изменять

1.8 How to Use a Design Pattern
Как использовать проектный шаблон

Once you've picked a design pattern, how do you use it? Here's a step-by-step approach to applying a design pattern effectively:

Как только вы выбрали проектный шаблон, как вам его использовать? Ниже приведена пошаговая инструкция по эффективному применению проектного шаблона:

1. *Read the pattern once through for an overview.* Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.

   Прочтите обзор шаблона. Особое внимание уделите параграфам Применимость и Последствия, чтобы удостовериться, что шаблон подходит для вашей задачи.

2. *Go back and study the Structure, Participants, and Collaborations sections.* Make sure you understand the classes and objects in the pattern and how they relate to one another.

   Вернитесь и изучите параграфы Структура, Участники и Взаимодействия. Удостоверьтесь, что вы понимаете классы и объекты в шаблоне и как они соотносятся.

3. *Look at the Sample Code section to see a concrete example of the pattern in code.* Studying the code helps you learn how to implement the pattern.

   Посмотрите параграф Пример Кода для конкретного примера кодирования шаблона. Изучение этого кода помогает вам понять как реализовать шаблон.

4. *Choose names for pattern participants that are meaningful in the application context.* The names for participants in design patterns are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application. That helps make the pattern more explicit in the implementation. For example, if you use the Strategy pattern for a text compositing algorithm, then you might have classes SimpleLayoutStrategy or TeXLayoutStrategy.

   Выберите имена для участников шаблона, значимые в контексте приложения. Имена участников в проектных шаблонах обычно слишком абстрактны для непосредственного применения в приложении. Тем не менее полезно вставить имя участника в то имя, которое появится в приложении. Это поможет сделать шаблон более явным в реализации. Например если вы используете шаблон Стратегия для алгоритма компоновки текста, то вы можете иметь классы ПростаяСтратегияРасположения ТеХСтратегияРасположения.

5. *Define the classes.* Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.

   Определите классы. Определите их интерфейсы, установите отношения наследования и определите переменные экземпляры, которые представляют ссылки на данные и объекты. Выделите существующие классы, на которые повлияет шаблон и измените их соответственно.

6. *Define application-specific names for operations in the pattern.* Here again, the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions. For example, you might use the "Create-" prefix consistently to denote a factory method.

   Определите специфические для приложения имена операций в шаблоне. Здесь снова имена в основном зависят от приложения. Используйте ответственность и взаимодействия каждой операции в качестве руководства. Также будьте последовательны в правилах именования. Например вы можете последовательно использовать префикс "Создать-" для указания на заводской метод.

7. *Implement the operations to carry out the responsibilities and collaborations in the pattern.* The Implementation section offers hints to guide you in the implementation. The examples in the Sample Code section can help as well.

   Реализуйте операции для выполнения обязанностей и взаимодействий в шаблоне. Параграф Реализация предлагает подсказки для этого. Также могут помочь примеры из параграфа Пример Кода.

These are just guidelines to get you started. Over time you'll develop your own way of working with design patterns.

Это только основные принципы для того чтобы вы могли начать. Со временем вы разработаете свой собственный метод работы с проектными шаблонами.

No discussion of how to use design patterns would be complete without a few words on how *not* to use them. Design patterns should not be applied indiscriminately. Often they achieve flexibility and variability by introducing additional levels of indirection, and that can complicate a design and/or cost you some performance. A design pattern should only be applied when the flexibility it affords is actually needed. The Consequences sections are most helpful when evaluating a pattern's benefits and liabilities.

Никакое обсуждение того как использовать шаблоны не может быть завершено без нескольких слов о том как их *не* использовать. Проектные шаблоны не должны применяться неразборчиво. Часто гибкость и варьируемость достигаются добавлением дополнительных уровней косвенности, и это может усложнить проект и/или уменьшить производительность. Проектный шаблон должен применяться только тогда, когда предлагаемая им гибкость действительно нужна. Параграф Последствия наиболее полезен при оценке достоинств и недостатков шаблона.