

## ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ. Ч. I

Предложен набор паттернов проектирования, применимых при создании архитектуры информационной системы. Рассмотренные в статье паттерны позволяют решать проблему управления произвольными организационными структурами и проблему управления произвольными бизнес-процессами. Кроме того, применение ряда предлагаемых приемов, таких, как, например, определение множества классов, трассируемых от прецедента, позволяет упростить переход от требований, накладываемых на информационную систему, к созданию проекта системы.

Введем несколько определений. Под информационной системой (ИС) будем понимать совокупность программных, аппаратных и людских ресурсов, задействованных в управлении бизнес-процессами и/или в обеспечении информационной поддержки бизнес-процессов предприятия или организации. Если такое воздействие затрагивает все или большую часть бизнес-процессов предприятия или организации, информационную систему можно называть корпоративной (КИС).

Под паттерном будем понимать классическое определение К. Александра: паттерн – это описание задачи, которая многократно возникает в нашей работе, а также принципа ее решения, причем так, что это решение можно потом использовать всякий раз, когда это необходимо, ничего не изобретая заново [1]. Хотя сам классик имел в виду паттерны, возникающие при проектировании зданий и городов, его слова применимы и в отношении паттернов программной инженерии.

### ИС КАК МОДЕЛЬ ОКРУЖАЮЩЕЙ ДЕЯТЕЛЬНОСТИ

Как и любая программа, ИС представляет собой модель деловой среды, для которой она создается. Действительно, все, что происходит в бизнесе, должно находить свое отражение в ИС: принят ли на работу новый сотрудник или в учебный план поставлен новый курс – все эти данные, раз они появились в реальном мире, должны появиться и в ИС в виде документов, элементов справочников, фотографий и т.д. Если этого не происходит, ИС не является адекватной моделью бизнеса и, следовательно, не может использоваться как инструмент для достижения успеха.

Говоря об ИС как о модели бизнеса, мы должны отдавать себе отчет в том, что бизнес постоянно меняется. Изменения бизнеса, в свою очередь, незамедлительно приводят к изменению требований, налагаемых на ИС. Помимо изменения требований к ИС, порожденных изменением бизнеса, возможны также изменения требований, порожденные изменившимся восприятием системы со стороны заинтересованных лиц и/или пользователей.

Хотя проблемы, связанные с изменением требований, относятся к области системного анализа, их последствия носят в том числе и инженерный характер – кому, как не инженерам, переписывать систему в результате изменившихся требований. Поэтому совершенно понятно стремление создавать системы, устойчивые к изменениям, то есть системы, позволяющие вносить изменения максимально быстро и дешево.

Чтобы говорить о создании устойчивых к изменениям систем, необходимо понять, что может изменяться в бизнесе и что не может изменяться в бизнесе. Авторы считают, что на стратегическом уровне неизменными в любом бизнесе являются факт существования организационной структуры бизнеса и факт

существования бизнес-процессов в контексте этой организационной структуры. Все остальное: состав организационной структуры, содержание бизнес-процессов и т.д. – может изменяться.

В данной работе рассматриваются два паттерна: «Организационная структура», «Бизнес-процесс».

При описании паттернов используется шаблон, принятый в [1].

### ПАТТЕРН «ОРГАНИЗАЦИОННАЯ СТРУКТУРА»

#### Назначение

Предоставляет возможность задавать произвольную иерархию подчинения при описании организационной структуры предприятия или организации.

#### Решение

Для описания организационной структуры в модели данных предлагается ввести 3 сущности (таблицы). Первая из них – это тип подразделения (*subdivision\_type*). Таблица хранит информацию о типах подразделений, присутствующих в организации. Примерами записей в такой таблице могут быть: факультет, кафедра, лаборатория, склад, цех и т.д.;

- правила подчинения (*subordination\_rule*). Таблица хранит пары ключей {идентификатор родительского типа (*parent\_type\_id*), идентификатор дочернего типа (*child\_type\_id*)}. Записи в данной таблице показывают, как типы подразделений могут подчиняться друг другу. Например, пусть 1 – номер типа Факультет, 2 – номер типа Кафедра, 3 – номер типа Лаборатория. Тогда пары {1,2}, {1,3} задают возможность подчинения кафедры факультету и лаборатории факультету.

- подразделение (*subdivision*). В данной таблице хранится информация о конкретных подразделениях предприятия или организации; например: ФПМК, РФФ, кафедра программирования, лаборатория вакцин и сыворток. Пара {*subdivision\_type\_id*, *parent\_type\_id*} формирует внешний ключ на таблицу *subordination\_rule*. Пара {*parent\_type\_id*, *parent\_id*} формирует внешний ключ к таблице *subdivision*. Наконец, поля {*subdivision\_type\_id*} и {*parent\_type\_id*} являются внешними ключами на таблицу *subdivision\_type*.

Описанные сущности представлены на ER-диаграмме (рис. 1).

#### Результаты

Основными результатами применения данного паттерна являются:

- возможность описывать произвольные организационные структуры;
- контроль над целостностью данных, касающихся организационной структуры, осуществляется на дек-

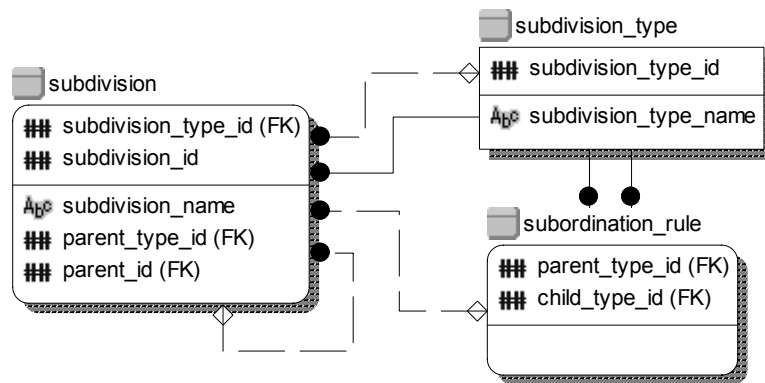


Рис. 1. Модель данных организационной структуры

ларативном уровне, что делает паттерн независимым от используемой системы управления базами данных.

Прежде чем начать рассмотрение паттерна «Бизнес-процесс», рассмотрим используемый в нем паттерн Рефлексия, так как для паттерна «Бизнес-процесс» возможность создавать объекты по имени класса является необходимой.

## ПАТТЕРН «РЕФЛЕКСИЯ»

### Назначение

Обеспечить возможность создания объектов по имени класса для языков программирования, не поддерживающих рефлексия явно.

### Решение

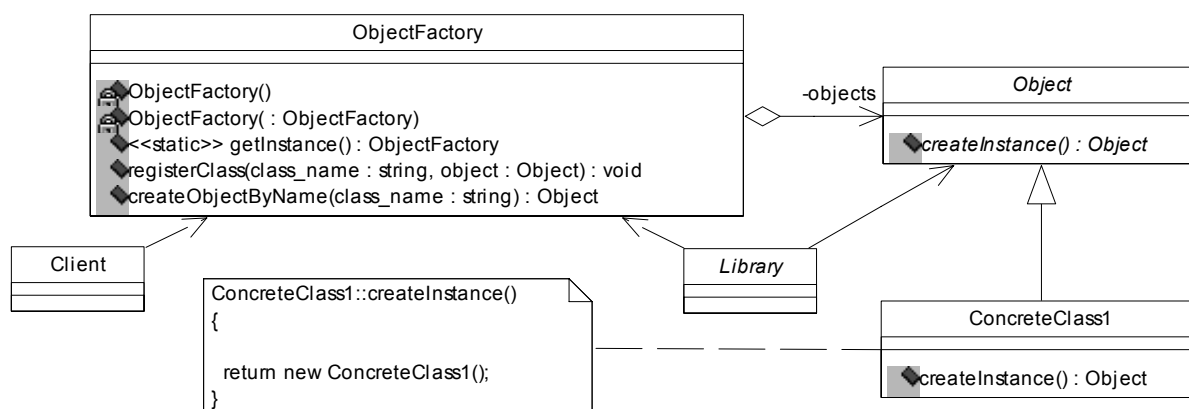


Рис. 2. Структура паттерна Рефлексия

### Участники

Object – базовый класс для тех классов системы, для которых необходимо создание объектов по имени. В простейшем случае Object объявляет интерфейс с одним абстрактным методом createInstance().

ConcreteClass1 – конкретный подкласс Object. Реализует метод createInstance(). Наиболее простой реализацией этого метода в конкретных подклассах Object является создание самого себя (рис. 2).

ObjectFactory – фабрика объектов. ObjectFactory реализуется в соответствии с шаблоном Singleton [1]. Данный класс предоставляет интерфейс для регистрации классов (метод registerClass()) и для создания объектов (метод createObjectByName()) (рис. 2).

Library предоставляет интерфейс для модулей системы. Конкретными подклассами Library могут являться классы, инкапсулирующие загрузку динамических библиотек, таких, как Dynamic Load Library (DLL) в Win32-системах или Shared Library в Unix-системах. В задачу данного класса входит регистрация содержащихся в них классов у ObjectFactory (рис. 3)

Client. Описывает классы, объекты которых являются клиентами относительно ObjectFactory. Клиенты используют ObjectFactory для создания объектов, вызывая у последней createObjectByName() (рис. 4)

### Взаимодействие

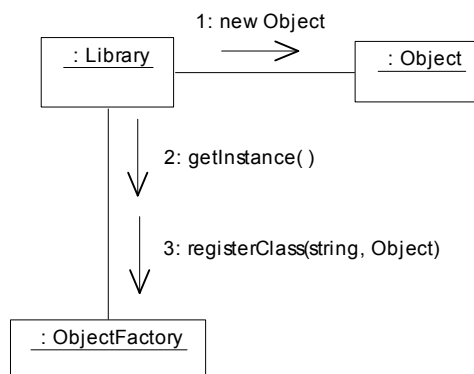


Рис. 3. Регистрация класса

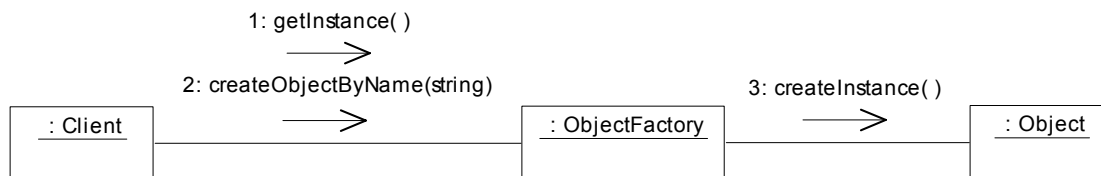


Рис. 4. Создание объекта

## Результаты

Основным результатом применения данного паттерна является возможность создавать объекты классов по имени, что, в свою очередь, повышает гибкость программного решения. С другой стороны, паттерн подразумевает организацию программы как набора динамически подгружаемых библиотек, что может оказаться неприемлемым. Так, например, динамически загружаемые библиотеки могут не поддерживаться целевой операционной системой.

## ПАТТЕРН «БИЗНЕС-ПРОЦЕСС»

### Назначение

- Упрощает прямую трассировку артефактов потока работ Требования [3,4] в артефакты потока работ Анализ и проектирование [4].

- Упрощает процесс внесения в систему изменений.

### Решение

Следуя рекомендациям, приведенным в [4], разобьем классы анализа на 3 категории:

- Бизнес-объекты (BusinessObject) – классы, представляющие понятия деловой среды.

- Граничные классы (BoundaryClass) – классы, отвечающие за взаимодействие системы с ее акторами (Actor, [4]).

- Классы управления – классы, отвечающие за координацию взаимодействия в контексте прецедентов [2,4]

В соответствии с паттерном MVC [1] все коммуникации между граничными классами и бизнес-объектами будем осуществлять через классы управления.

Для каждого прецедента, выявленного на этапе Требования, введем один класс управления, который будем называть контроллер прецедента (UseCase-Controller) или просто контроллер. Будем считать, что  $i$ -й контроллер в каждый момент времени может находиться в одном состоянии из множества  $\{S_{i1}, S_{i2}, \dots, S_{iN}\}$ . Переход из состояния  $S_{ij}$  в состояние  $S_{ik}$  происходит при получении контроллером сообщения  $M$ . Сообщение  $M$  может быть послано контроллеру любым объектом системы. Правила перехода из состояния в состояние инкапсулированы в контроллере (каждый контроллер хранит у себя таблицу состояний и переходов).

Таким образом, мы получим архитектуру, в которой для каждого прецедента деловой среды, описываемого конечным автоматом, явно выделяется своя группа классов, реализующая или поддерживающая этот прецедент в ИС. Это положение решает первую

задачу паттерна – упрощение прямой трассировки между артефактами потоков работ. Действительно, для каждого прецедента явно выделяется по одному контроллеру и несколько классов, с которыми выделенный контроллер состоит в отношении ассоциации [4]. Для того чтобы решить вторую задачу, упростить процесс внесения изменений в систему, необходимо разобраться с характером изменений программных артефактов, наступающих в ответ на изменения требований. Изменения программных артефактов могут заключаться:

- в изменении набора состояний контроллеров, правил перехода из состояния в состояние, а также в изменении логики обработки сообщений. К таким изменениям может привести, например, изменение последовательности действий внутри некоторого бизнес-процесса;

- в изменении внешнего представления бизнес-объектов, т.е. в изменении граничных классов. К таким изменениям могут привести изменение восприятия системы со стороны заинтересованных лиц или изменение структуры самих бизнес-объектов;

- в изменении структуры и поведения бизнес-объектов, обусловленных, например, изменением внешней среды (изменение законодательства, реинжиниринг и т.д.).

Для устойчивости к такого рода изменениям необходимо:

- вынести из контроллеров описание состояний прецедентов и правила перехода из состояния в состояние, в некоторое внешнее по отношению к классам системы метаописание. На контроллеры должна возлагаться ответственность лишь за интерпретацию этого метаописания (рис. 5);

- вынести из контроллеров поведение, отвечающее за изменение состояния бизнес-модели. Это действие необходимо для того, чтобы мы могли легко вносить в систему изменения, касающиеся динамических аспектов моделируемого бизнеса;

- унифицировать интерфейсы контроллеров;

- по возможности, вынести из кода системы информацию об именах конкретных классов. Объекты должны создаваться с использованием рефлексии, а не через явный вызов конструкторов.

Накладываемые на архитектуру ограничения приводят к тому, что контроллеры перестают отличаться друг от друга. Следовательно, отпадает необходимость определять свой класс контроллера для каждого прецедента. Достаточно будет одного класса контроллера, объекты которого будут параметризоваться таблицей состояний и переходов и именами конкретных граничных классов и бизнес-объектов.

## Структура

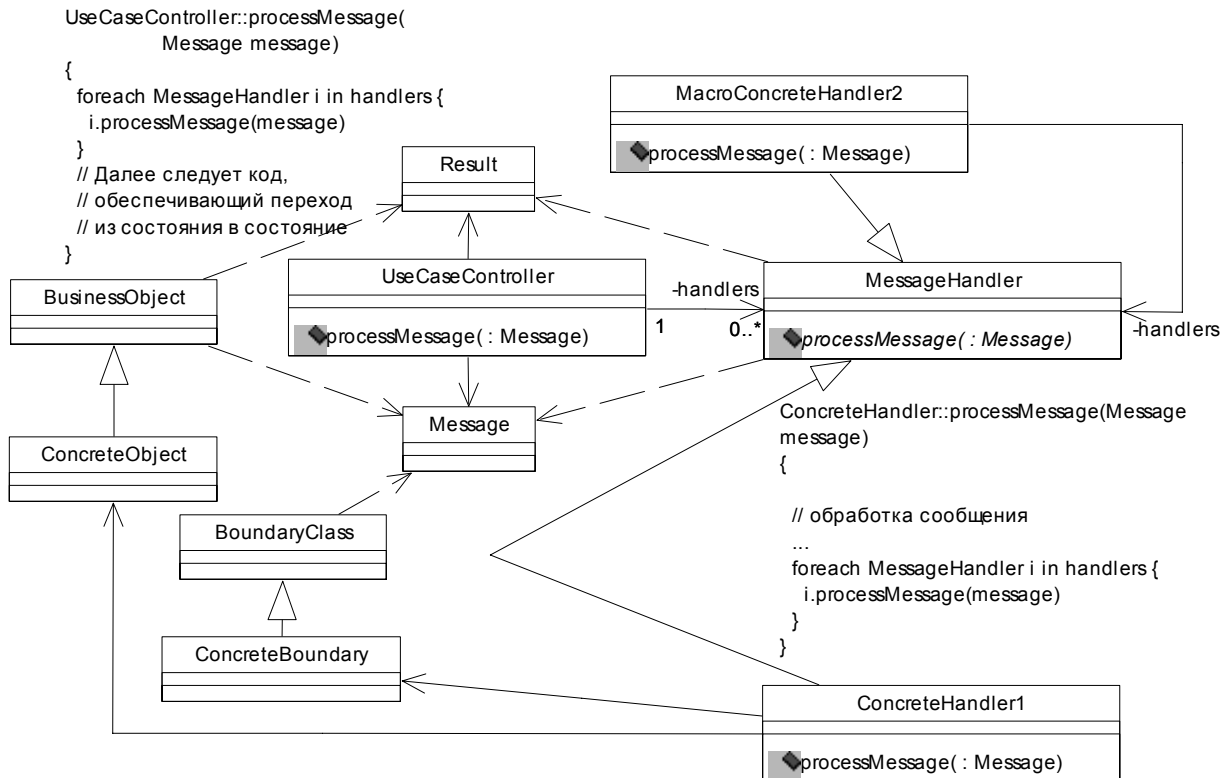


Рис. 5. Структура паттерна Бизнес-процесс

### Участники

**BusinessObject.** Базовый класс бизнес-объектов деловой среды.

**UseCaseController.** Контроллер прецедента. Отвечает за координацию объектов в контексте прецедента. В простейшем случае содержит один-единственный метод processMessage(). Метод предназначен для обработки посылаемых контроллеру сообщений. Реально данный метод большинство сообщений не обрабатывает, а делегирует их обработку объектам класса MessageHandler. Исключением являются сообщения, инициирующие переход контроллера из состояния в состояние.

**BoundaryClass.** Базовый класс граничных классов. Подклассы граничного класса отвечают за организацию взаимодействия системы с внешней средой.

**Message.** Представляет собой абстракцию сообщения. Так, если некоторый объект (клиент) системы хочет инициировать поведение другого объекта (сервера), он создает объект Message и посылает его серверу.

**Result.** Представляет собой абстракцию результата обработки сообщения. Объекты данного класса предназначены для того, чтобы сервер мог вернуть результат своей работы клиенту. **MessageHandler.** Базовый класс обработчиков сообщений.

**MessageHandler.** Обработчик сообщений. В подклассах данного класса содержится практически вся бизнес-логика работы системы. Именно объекты MessageHandler заставляют другие объекты сохра-

няться, изменяться, удаляться, что-либо рассчитывать или показывать и т.д. Конкретные подклассы MessageHandler могут иметь связи с конкретными бизнес-объектами и граничными классами, необходимыми им для обеспечения своей ответственности.

**ConcreteObject.** Конкретный подкласс BusinessObject, например Факультет или Документ.

**ConcreteBoundary1.** Конкретный граничный класс, например форма для внесения личных дел сотрудников.

**ConcreteHandler1, MacroConcreteHandler2.** Конкретные обработчики сообщений.

### Взаимодействие

При старте системы для каждого прецедента создается контроллер (объект класса UseCaseController). Контроллер загружает свою таблицу состояний и переходов, создает необходимые для работы объекты (обработчики сообщений, возможно, бизнес-объекты и объекты граничных классов) и переходит в некоторое начальное состояние. Далее, контроллеру посредством вызова processMessage() может быть послано некоторое сообщение. Контроллер, получив сообщение, пересылает его всем ассоциированным с ним обработчикам (объекты класса MessageHandler) и, дождавшись обработки, возможно, переходит в новое состояние. Обработчики, получив от контроллера сообщение, либо игнорируют его, либо осуществляют какие-то действия по обработке, например сохраняют что-то в базе данных.

## Схема данных

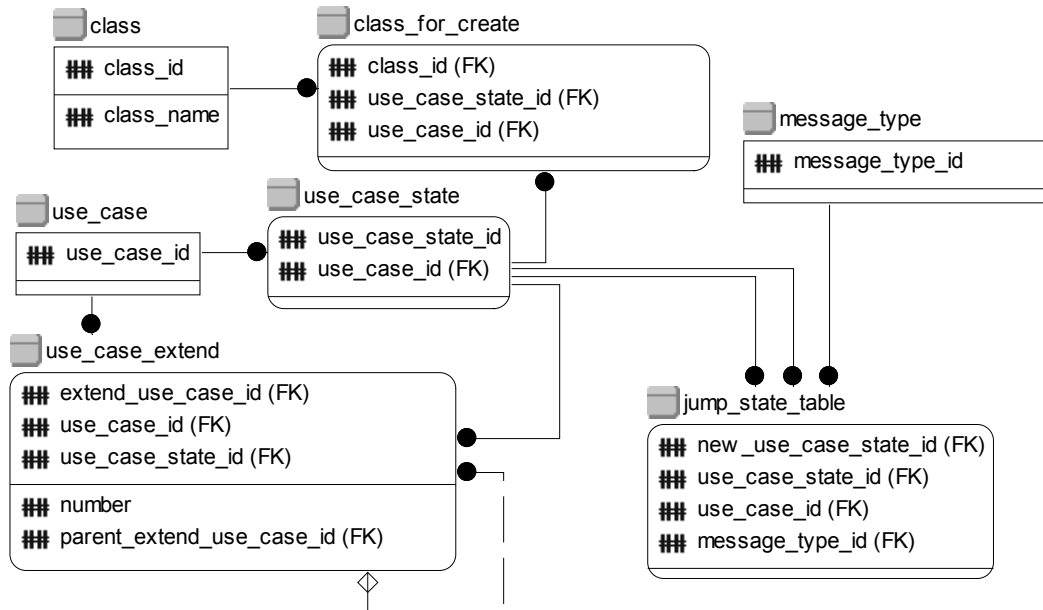


Рис. 5. Структура паттерна Бизнес-процесс

class. Хранит информацию о классах системы.

class\_for\_create. Хранит информацию о том, какие классы должны создаваться при переходе контроллера прецедента в некоторое состояние.

use\_case. Хранит информацию о прецедентах, присутствующих в системе прецедентах.

use\_case\_state. Хранит информацию о возможных состояниях контроллеров.

message. Хранит информацию о сообщениях.

use\_case\_extend. Хранит информацию о расширениях (extent) прецедентов [4], позволяя тем самым задавать условия вида [Если прецедент  $UC_i$  находится в состоянии  $S_k$ , необходимо запустить выполнение прецедента  $UC_j$ ].

jump\_state table. Таблица состояний и переходов. Хранит информацию о том, в какое состояние должен

перейти контроллер прецедента, если он находится в состоянии  $S_i$  и получил сообщение  $M_j$ .

## Результаты

Паттерн Бизнес-процесс фактически представляет собой архитектурное ядро ИС. Архитектура, построенная на базе паттерна Бизнес-процесс, обладает следующими характеристиками:

- Классы ядра обладают низкой степенью связности (см. описание паттерна Low Coupling [2]), что позитивно сказывается на возможности повторного использования.
- Позволяет легко наращивать функциональность системы.
- Не зависит от физической архитектуры системы.

## ЛИТЕРАТУРА

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.
2. Ларман К. Применение UML и шаблонов проектирования. Вильямс, 2002.
3. Dean Leffingwell, Don Widrig. Managing Software Requirements. Addison-Wesley, 2000.
4. Rational Unified Process. Versions 2001-2003. Rational Software Corporation. <http://www.rational.com/>

Статья представлена в научную редакцию 15 мая 2003 г.