

Программистам от программистов

PHP5 для профессионалов

Язык реализации серверной части приложений PHP является идеальным средством быстрой разработки сложных, взаимодействующих с базами данных Web-узлов, Web-приложений и корпоративных сетей. В данном практическом руководстве продемонстрирована вся мощь и гибкость языка PHP и даны полезные советы программистам. В этой книге показано, как построить масштабируемую и высокопроизводительную инфраструктуру на языке PHP5, подробно описан набор инструментов, который можно использовать многократно, и приведены многочисленные практические примеры.

В книге рассматриваются концепции объектно-ориентированного подхода и их реализация на языке PHP5, описываются методологии управления проектами, а также уделяется внимание обсуждению многих других вопросов. Вся эта информация позволит вам научиться разрабатывать качественное программное обеспечение в более сжатые сроки.

Из книги вы узнаете, как:

- применять базовые и расширенные принципы объектно-ориентированной разработки при создании программ на языке PHP;
- описывать и документировать программный код на языке UML;
- применять преимущества повторного использования кода на основе шаблонов;
- использовать в качестве строительных блоков программ коллекции, итераторы, объекты взаимодействия с базами данных и шаблоны;
- использовать шаблон "модель-вид-контроллер" в приложениях на языке PHP;
- применять на практике все преимущества модульного тестирования;
- правильно применять методологии управления проектами, планирования и построения архитектуры в процессе разработки более качественного программного обеспечения.

Книги данной серии написаны профессиональными программистами и отражают реальные требования разработчиков и профессионалов в области информационных технологий. В них освещены вопросы, с которыми ежедневно приходится сталкиваться профессионалам. В книгах приводятся практические примеры, готовые решения и рекомендации специалистов, которые помогут программистам повысить уровень знаний и качество работы.

Э. Леки-Томпсон,
Х. Айде-Гудман,
А. Кав,
С. Новицки



Программистам от программистов

Руководство по PHP5

для
профессионалов

для профессионалов

Для кого предназначена эта книга

Эта книга предназначена для разработчиков программ на языке PHP, которые для написания приложений в настоящее время используют процедурный подход, но хотят перейти к применению объектно-ориентированных принципов, воспользоваться современными концепциями построения архитектуры программных систем и повысить таким образом свои профессиональные навыки.

Категория:	языки программирования
Предмет рассмотрения:	PHP
Уровень:	для пользователей средней и высокой квалификации



Издательство "Диалектика"
www.dialektika.com

p2p.wrox.com

Информационный центр
для программистов
www.wrox.com

ISBN 5-8459-1066-8



Эд Леки-Томпсон, Хью Айде-Гудман, Алек Кав, Стивен Д. Новицки



Обновления, исходный код и техническая поддержка на сайте
www.wrox.com

www.dialektika.com

PHP5

для профессионалов

Professional

PHP5

Ed Lecky-Thompson
Heow Eide-Goodman
Steven D. Nowicki
Alec Cove



Wiley Publishing, Inc.

PHP5

для профессионалов

Эд Леки-Томпсон
Хъяо Айде-Гудман
Алек Кав
Стивен Д. Новицки



“Диалектика”
Москва • Санкт-Петербург • Киев
2006

ББК 32.973.26-018.2.75

Л43

УДК 681.3,07

Компьютерное издательство “Диалектика”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук. *А.Ю. Шелестова*

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:

info@dialektika.com, <http://www.dialektika.com>
115419, Москва, а/я 783; 031150, Киев, а/я 152

Леки-Томпсон, Эд, Коув, Алек, Новицки, Стивен, Айде-Гудман, Хью

Л43 PHP 5 для профессионалов. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2006. — 608 с. : ил. — Парал. тит. англ.

ISBN 5-8459-1066-8 (рус.)

В данном практическом руководстве продемонстрирована вся мощь и гибкость языка PHP и даны полезные советы программистам. В этой книге показано, как построить масштабируемую и высокопроизводительную инфраструктуру на языке PHP5, подробно описан набор инструментов, который можно многократно использовать, и приведены многочисленные практические примеры.

В книге рассматриваются концепции объектно-ориентированного подхода и их реализация на языке PHP5, описываются методологии управления проектами, а также уделяется внимание обсуждению многих других вопросов. Вся эта информация позволит вам научиться разрабатывать качественное программное обеспечение в более сжатые сроки.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства JOHN WILEY&Sons, Inc.

Copyright © 2006 by Dialektika Computer Publishing.

Original English language edition Copyright © 2005 by Wiley Publishing, Inc., Indianapolis, Indiana.

All rights reserved including the right of reproduction in whole or in part in any form. This translation published by arrangement with Wiley Publishing, Inc.

Wiley, the Wiley Publishing logo, Wrox, the Wrox logo, and Programmer to Programmer are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

ISBN 5-8459-1066-8 (рус.)

© Компьютерное изд-во “Диалектика”, 2006
перевод, оформление, макетирование

ISBN 0-7645-7282-2 (англ.)

© by Wiley Publishing, Inc., 2005

Оглавление

Введение	17
Часть I. Основы разработки объектно-ориентированного программного обеспечения	23
Глава 1. Введение в объектно-ориентированное программирование	24
Глава 2. Унифицированный язык моделирования UML	52
Глава 3. Объектный подход в действии	71
Глава 4. Шаблоны проектирования	94
Часть II. Разработка повторно используемого набора объектов: простые служебные классы и интерфейсы	119
Глава 5. Класс Collection	120
Глава 6. Класс CollectionIterator	141
Глава 7. Класс GenericObject	149
Глава 8. Уровни абстракции базы данных	172
Глава 9. Интерфейс Factory	193
Глава 10. Управление событиями	200
Глава 11. Регистрация событий и отладка	212
Глава 12. Протокол SOAP	231
Часть III. Разработка повторно используемого набора объектов: сложные (но не слишком) служебные классы	247
Глава 13. Модель, вид, контроллер	248
Глава 14. Общение с пользователями	279
Глава 15. Сеансы и аутентификация	301
Глава 16. Каркас для модульного тестирования	328
Глава 17. Конечные автоматы и файлы конфигурации	342
Часть IV. Учебный пример: автоматизация работы торгового предприятия	357
Глава 18. Знакомство с проектом	358
Глава 19. Методологии управления проектами	366
Глава 20. Проектирование системы	385
Глава 21. Архитектура системы	398
Глава 22. Разработка средства автоматизации торговли	407
Глава 23. Обеспечение качества	493
Глава 24. Развёртывание	509
Глава 25. Разработка надежной системы генерации отчетов	519
Глава 26. Что дальше	536
Часть V. Приложения	539
Приложение А. Зачем использовать контроль версий	540
Приложение Б. Интегрированные среды разработки для языка PHP	554
Приложение В. Настройка производительности PHP	571
Приложение Г. Практические советы по установке PHP	583
Предметный указатель	595

Содержание

Введение	17
Часть I. Основы разработки объектно-ориентированного программного обеспечения	23
Глава 1. Введение в объектно-ориентированное программирование	24
Что такое объектно-ориентированное программирование	24
Преимущества объектно-ориентированного подхода	25
Конкретный пример	26
Основные принципы и понятия ООП	27
Классы	27
Объекты	28
Наследование	37
Интерфейсы	47
Инкапсуляция	49
Изменения ООП в PHP 5	50
Резюме	51
Глава 2. Унифицированный язык моделирования UML	52
Определение требований	52
Интервьюирование клиента	53
Диаграммы прецедентов	54
Диаграммы классов	56
Моделирование предметной области	56
Отношения между классами	58
Реализация	60
Диаграммы видов деятельности	63
Диаграммы последовательностей	64
Диаграммы состояний	67
Диаграммы компонентов и развертывания	68
Резюме	70
Глава 3. Объектный подход в действии	71
Создание менеджера контактов	71
Диаграммы UML для адресной книги	72
Класс PropertyObject	77
Типы контактной информации	79
Класс DataManager	82
Классы Entity, Individual и Organization	84
Использование системы	91
Резюме	92

Глава 4. Шаблоны проектирования	94
Шаблон Composite	95
Реализация	96
Обсуждение	99
Шаблон Observer	100
Элементы управления	101
Обсуждение	106
Шаблон Decorator	106
Реализация	109
Использование шаблона Decorator	110
Обсуждение	111
Шаблон Facade	112
Шаблон Builder	113
Реализация	114
Обсуждение	117
Резюме	117
Часть II. Разработка повторно используемого набора объектов: простые служебные классы и интерфейсы	119
Глава 5. Класс Collection	120
Назначение класса Collection	121
Проектирование класса Collection	122
Основы класса Collection	122
Метод addItem()	123
Методы getItem() и removeItem()	124
Другие методы	124
Использование класса Collection	125
Реализация позднего инстанцирования	126
Обратные вызовы	127
Метод setLoadCallback() класса Collection	131
Использование класса Collection	135
Усовершенствование класса Collection	140
Резюме	140
Глава 6. Класс CollectionIterator	141
Интерфейс Iterator	142
Класс CollectionIterator	143
Интерфейс IteratorAggregate	145
Захист содержимого объекта Iterator с помощью оператора clone	146
Резюме	148
Глава 7. Класс GenericObject	149
Класс GenericObject	150
Когда нужно использовать класс GenericObject	150
Что позволяет делать класс GenericObject	150
Преимущества использования	151
Типичная реализация GenericObject	153

Встреча с предком	155
Взаимодействие класса GenericObject с базой данных	157
Методы и свойства класса GenericObject	159
Преимущества использования класса GenericObject	162
Класс GenericObjectCollection	162
Традиционная реализация	163
Когда традиционная реализация оказывается неудачной	164
Принципы, положенные в основу класса GenericObjectCollection	164
Исходный код класса GenericObjectCollection	165
Типичное использование класса GenericObjectCollection	167
Тестирование класса UserHome	168
Как это работает	168
Класс GenericObjectCollection: подведение итогов	171
Резюме	171
Глава 8. Уровни абстракции базы данных	172
Что такое уровень абстракции базы данных	173
Простая реализация	173
Конфигурационный файл	174
Установка соединения	174
Выборка данных	175
Изменение информации	175
Использование класса Database	177
Уровень абстракции PEAR DB	179
Подключение к базе данных с помощью класса DB	180
Извлечение информации	181
Другие полезные функции	183
Получение дополнительной информации	185
Завершенный уровень абстракции базы данных	185
Поддержка транзакций	189
Шаблон проектирования Singleton	190
Резюме	192
Глава 9. Интерфейс Factory	193
Шаблон проектирования Factory	193
Пример интерфейса Factory	194
Старый школьный подход	194
Применение интерфейса Factory	195
Использование интерфейса Factory на уровне абстракции базы данных	196
Многократное применение шаблона Factory	198
Усовершенствование существующих классов	199
Резюме	199
Глава 10. Управление событиями	200
Что такое события	201
Использование ООП для управления событиями	202
Проектное решение по управлению событиями	202
Реализация решения	204
Обеспечение безопасности	208

Остановитесь и подумайте	210
Резюме	211
Глава 11. Регистрация событий и отладка	212
Создание механизма регистрации событий	212
Запись в файл	212
Пример записи в файл	213
Класс Logger	214
Расширение класса Logger	218
Создание механизма отладки	227
Резюме	230
Глава 12. Протокол SOAP	231
SOAP и PHP 5	232
Расширение SOAP PHP 5	232
Создание SOAP-клиента	235
За кулисами	238
Обработка исключений в клиенте SOAP	242
Создание сервера SOAP	243
Резюме	245
Часть III. Разработка повторно используемого набора объектов: сложные (но не слишком) служебные классы	247
Глава 13. Модель, вид, контроллер	248
Знакомство с архитектурой MVC	249
Модель	250
Вид	250
Контроллер	250
Инфраструктура	250
MVC в Web-приложениях	250
MVC в языке PHP	251
Минимальный набор классов для реализации MVC	253
Знакомство с набором	253
Использование предложенного набора	263
Применение разработанного набора	268
Реальные шаблоны	269
Реализация шаблонов средствами PHP	269
Реализация шаблонов на основе пакета Smarty	270
Установка пакета Smarty	270
Использование пакета Smarty	271
Расширенные возможности пакета Smarty	275
Когда лучше использовать пакет Smarty, а не традиционные шаблоны	277
Резюме	277
Глава 14. Общение с пользователями	279
Для чего общаться	279
Причины для общения с пользователями	280
За пределами Web-браузера	282

Типы связи	282
Общие свойства взаимодействия	282
Свойства конкретных типов сообщений	283
Информация о получателе	283
Представление взаимодействия в виде иерархии классов	284
Класс Recipient: быстрая проверка объектного мышления	284
Класс Communication	287
Переписка с пользователями по электронной почте	289
Создание тестовой версии	289
Получение сообщения	292
Применение шаблонов	298
Использование MIME	299
Другие подклассы Communication	300
Передача SMS-сообщений	300
Факс	300
Резюме	300
Глава 15. Сеансы и аутентификация	301
Знакомство с сеансами	302
Краткий экскурс в историю протокола HTTP	302
Определение сеанса	304
Сохранение сеанса	304
Безопасность сеанса	306
Реализация сеансов в PHP	312
Сеансы в PHP	312
Ограничения базовых сеансов PHP	315
Создание класса аутентификации	315
Подключение механизма управления сеансами к базе данных	315
Знакомство с классом UserSession	316
Схема базы данных	317
Код класса UserSession.phpm	318
Код модульного теста для класса UserSession	321
Как работает класс UserSession	324
Использование класса UserSession	326
Резюме	326
Глава 16. Каркас для модульного тестирования	328
Методология и терминология	328
Разработка интерфейса класса	329
Создание пакета тестирования для класса	330
Реализация класса	331
Повторный запуск	331
Знакомство с PHPUnit	331
Установка PHPUnit	332
Использование PHPUnit	332
Тестовые классы	332
Тестовый пакет	334
Зачем беспокоиться?	335
Возвратное тестирование	335

Удобство использования каркаса	336
Гарантия качества	336
Упрощение функционального тестирования	336
Пример из жизни	337
Резюме	341
Глава 17. Конечные автоматы и файлы конфигурации	342
Знакомство с конечными автоматами	343
Простой конечный автомат: калькулятор для обратной польской записи	343
Теоретические реализации конечных автоматов	345
Реализация конечных автоматов на PHP	345
Пример калькулятора для обратной польской записи	347
Реальные примеры конечных автоматов	350
Пользовательские конфигурационные файлы	351
Использование PHP	351
Использование XML	352
Использование INI-файлов	352
Класс Config пакета PEAR	354
Рекомендации для работы с конфигурационными файлами	355
Резюме	356
Часть IV. Учебный пример: автоматизация работы торгового предприятия	357
Глава 18. Знакомство с проектом	358
Компания Widget World	358
Компания Widget World изнутри	359
Технический уровень	360
Финансовый уровень	360
Политический уровень	361
Разработчик	361
Действительно ли дело в технологии	361
Подход к разработке	361
Что это означает	363
Технология	365
Резюме	365
Глава 19. Методологии управления проектами	366
Выполните домашнее задание	367
Почему возникает новый проект	367
Для кого проект предназначен	367
Какую предысторию имеет проект	369
Каковы исходные условия	369
Разработка описания проекта	370
Формулировка бизнес-требований	370
Определение границ	372
График выполнения работ	373
Бюджет	374
Коммерческие условия	375
Планы на будущее	376

“Внешний облик” приложения	376
Технология	377
Поддержка	377
Что делать дальше	377
Написание предложения	378
Предложение или счет	378
Предложение или спецификация	378
Кто должен участвовать в написании предложения	379
Когда следует двигаться дальше	379
Когда следует сказать “нет”	380
Структурирование предложения	380
Выбор персонала	382
Менеджер проекта	382
Исполнительный директор	382
Главный архитектор	382
Разработчики и кодировщики	383
Разработчики клиентской части приложения	383
Главный дизайнер	383
Художники	383
Совмещение ролей	384
Организация работ	384
Роль заказчика	384
Резюме	384
Глава 20. Проектирование системы	385
Выбор процесса	385
Каскадный процесс	385
Спиральный процесс	386
Принятие решений	388
Общие рекомендации	388
Фаза разработки спецификации	388
Фаза проектирования	390
Карта страниц	391
Фаза реализации	391
Фаза тестирования	392
Передача проекта	392
Методология программирования	393
Разработка на основе тестирования	393
Экстремальное программирование	393
Управление изменениями	395
Модификация требований	396
Изменение требований после подписания спецификации	396
Конфликт из-за разницы толкования	396
Дефекты, обнаруженные клиентом	396
Резюме	397
Глава 21. Архитектура системы	398
Что такое системная архитектура	398
Почему это важно	398
Что нужно сделать	399

Эффективная реализация требований	399
Хостинг, соединения, серверы и сеть	400
Надежность и избыточность	400
Поддержка	401
Безопасность	401
Проектирование среды	401
Хостинг и соединения	401
Вычисление параметров канала	401
Серверы	403
Сеть	404
Дополнительная память	405
Поддержка	405
Безопасность	405
Резюме	406
Глава 22. Разработка средства автоматизации торговли	407
Начало проекта: понедельник	407
Слушайте внимательно	408
Оценка трудоемкости реализации сценариев	410
Планирование процесса разработки	416
Начало работы	417
Детали реализации сценария 9	418
Создание тестов	418
PHPUnit	419
Создание страницы регистрации	425
Следующий сценарий	427
Повторная оценка проекта	437
Чистка кода	438
Рефакторинг кода	438
Завершение итерации	444
Сценарий 14 – “Сохранение данных при изменении недели”	444
Сценарий 15 – “Еженедельное добавление данных в отчет о контактах с покупателями”	445
Отчет о командировочных расходах	453
Элемент командировочных расходов	453
Вычисление командировочных расходов за неделю	457
Последние штрихи	460
Дополнительные тесты для отчета о еженедельных расходах	461
Реализация класса еженедельного отчета о командировочных расходах	463
Окончательный отчет по командировочным расходам	474
Объекты-имитаторы	488
Резюме	492
Глава 23. Обеспечение качества	493
Основы анализа качества	493
Почему нужно ставить высокие цели	494
Что такое качество	495
Мера качества	495

Тестирование	497
Модульное тестирование	497
Функциональное тестирование	498
Тестирование нагрузки	499
Тестирование удобства использования	500
Отслеживание ошибок	500
Отслеживание ошибок с помощью системы Mantis	501
Несколько заключительных слов о системе Mantis	507
Резюме	508
Глава 24. Разворачивание	509
Организация среды разработки	509
Сервер разработки	509
Поэтапная разработка	510
Среда поэтапного развёртывания	511
Рабочая среда	512
Разработка баз данных	512
Процесс развёртывания	513
Автоматическое извлечение данных из хранилища контроля версий	514
Утилита rsync	516
Синхронизация серверов с помощью утилиты rsync	518
Резюме	518
Глава 25. Разработка надежной системы генерации отчетов	519
Рабочие данные	519
Понимание потребностей заказчика	520
Управление запросами заказчика	520
Данные отчета	521
Разработка отчета	521
Архитектура генерации отчетов	524
Генерация отчетов в фоновом режиме	525
Страница Reports	526
Страница newreport	527
Сценарий обработки отчета	530
Обработка	531
Сценарии обработчиков	532
Страница Мои отчеты	533
Сценарии преобразования отчетов	533
Пример использования генератора отчетов	534
Визуализация	535
Резюме	535
Глава 26. Что дальше	536
Мотивация	536
Ваша карьера как разработчика	537
За пределами Web-приложений	537
Жизненный опыт	537
Академические навыки	537
Жизнь в социуме	538
Резюме	538

Часть V. Приложения	539
Приложение А. Зачем использовать контроль версий	540
Принципы контроля версий	540
Параллельное и исключающее управление версиями	541
Топология контроля версий	546
Программное обеспечение контроля версий	548
Microsoft Visual SourceSafe	548
CVS	550
RCS	551
Другие системы контроля версий	552
Дополнительные приемы контроля версий	552
Ветвление	552
Тегирование	553
Комментарии	553
Двоичные файлы	553
Резюме	553
Приложение Б. Интегрированные среды разработки для языка PHP	554
Выбор IDE	555
Zend Studio Client	555
Komodo	565
Другие IDE и редакторы	569
Резюме	570
Приложение В. Настройка производительности PHP	571
Проблемы производительности	571
Типы узких мест, связанных с производительностью	572
Причины недостаточной производительности	574
Поиск узких мест	574
Повышение производительности	576
Устранение временных задержек при работе с базой данных	576
Устранение узких мест в коде	577
Тестирование	579
Предупреждение неприятностей	580
Рекомендации по реализации высокопроизводительной архитектуры	580
Тестирование нагрузки	581
Резюме	582
Приложение Г. Практические советы по установке PHP	583
Введение в установку PHP	583
Выбор платформы	584
Лучший Web-сервер	585
Лучшая база данных	586
Инсталляция	586
Загрузка и установка СУБД PostgreSQL	587
Установка дополнительных библиотек	589
Установка PHP и Apache	590
Тестирование рабочего окружения	591
Когда использовать систему Windows	592

Модификация путей	593
Странные различия	593
Внешние библиотеки	593
Использование пакета PEAR	593
Резюме	594
Предметный указатель	595

Введение

Книга *PHP 5 для профессионалов* написана для разработчиков, постоянно стремящихся к совершенствованию и расширению своих знаний и навыков в области языка PHP. С ее помощью эту технологию также смогут освоить профессионалы по C++ и Java. Неопытным разработчикам настоятельно рекомендуется начать с книги *PHP 5 для начинающих*.

О чём эта книга

Эта книга посвящена не только синтаксису и принципам разработки программных систем на языке PHP. Она поможет научиться быстро создавать качественное программное обеспечение. Большая часть материала относится не только к языку PHP. Многие приемы разработки напрямую можно перенести на другие традиционные языки высокого уровня. Поэтому полученные при изучении книги знания в любом случае помогут росту вашей карьеры.

PHP 5 — динамично развивающийся язык, предназначенный не только для разработки Web-приложений. В книге вы узнаете, как язык PHP 5 использовать для написания сервисов (служб), приложений общего назначения и сценариев командной строки.

Для кого предназначена эта книга

Эта книга охватывает серьезные технологии разработки на языке PHP, поэтому читатель должен владеть основами создания приложений. При этом он необязательно должен быть знаком с PHP 5. Если вы до сих пор работали с PHP 4, то этого вполне достаточно. В книге четко описаны различия между этими версиями.

Возможно, вы работаете разработчиком Web-приложений в большой компании и хотите перейти от использования технологии ASP, Java и других платформ к PHP. В этом случае книга поможет вам определить преимущества данной технологии.

Возможно, вы являетесь единственным Web-профессионалом в непрофильной компании, перед которым стоит задача реализации большого проекта. В этом случае вам придется определиться с выбором платформы разработки и курировать сам процесс.

А может быть, вы учитесь в университете и хотите освоить технологии разработки и управления проектами, которые помогут вам сделать карьеру.

Предполагается, что у читателя есть определенный опыт разработки на C++ или Java, тогда данная книга поможет вам глубже освоить принципы объектно-ориентированного проектирования и стать профессионалом по разработке Web-приложений на PHP.

Кем бы вы ни были, если вы знаете основы PHP и хотите получить новые знания не просто от профессионалов, а от людей, влюбленных в эту технологию, то эта книга для вас.

Что необходимо для работы с этой книгой

Приведем краткий перечень требований, которые необходимо удовлетворить для плодотворного изучения книги.

- Рабочая станция с операционной системой Windows или Linux и любым текстовым редактором или средой разработки, а также Web-браузером для демонстрации примеров.
- Сервер разработки с интерпретатором PHP, сконфигурированный в соответствии с рекомендациями приложения Г.
- Сетевое соединение между этими компьютерами.

В идеале желательно иметь соединение с Интернет, поскольку в данной книге приводится множество адресов полезных Web-ресурсов.

Как структурирована эта книга

В этом разделе рассматривается общая структура книги. Более детальное содержание приводится в оглавлении. Если вы хотите ознакомиться с основами объектно-ориентированной разработки и ее реализацией на PHP 5, вам следует начать с части I. Остальные главы книги базируются на этом материале, поэтому для их успешного освоения необходимо познакомиться с основами объектно-ориентированного подхода. Книга состоит из 26 глав и 4 приложений.

Часть I. Основы разработки объектно-ориентированного программного обеспечения

Часть I книги посвящена основам объектно-ориентированной разработки. В PHP 5 эта технология реализована на более серьезном уровне. Вы ознакомитесь с основными понятиями объектно-ориентированного подхода, научитесь проектировать и документировать программные системы на языке UML, а также получите общее представление о приемах разработки, обеспечивающих повторное использование кода. После завершения изучения этой части вы будете готовы к разработке приложений на PHP 5 . Для того чтобы опробовать на практике полученные навыки, следует перейти к изучению частей II и III.

Часть II. Разработка повторно используемого набора объектов: простые служебные классы и интерфейсы

В части II приводится не просто набор случайных фрагментов кода.

Вы узнаете о высокоуровневых абстракциях и способах их использования. Коллекции, итераторы, шаблоны используются для создания строительных блоков, которые можно применять при разработке различных приложений.

Здесь же рассказывается о некоторых технологиях разработки, в частности о разработке на основе событий и применении протокола SOAP.

Часть III. Разработка повторно используемого набора объектов: сложные (но не слишком) служебные классы

Эта часть посвящена усовершенствованным возможностям современной разработки на PHP 5 . В ней речь пойдет о реализации шаблона “Модель–Вид–Контроллер”, специфике модульного тестирования и реализации абстракции конечного автомата.

Кроме того, в этой части обсуждаются вопросы аутентификации и обработки сеансов.

Часть IV. Учебный пример: автоматизация работы торгового предприятия

Высокие технологии — это хорошо, но разработчику приходится работать в команде и создавать реальные программы. При выполнении таких проектов вам понадобятся методологии управления проектами, планирования и разработки системной архитектуры. Именно эти вопросы и рассматриваются в данной части.

Здесь же приводится пример разработки системы с использованием множества описанных выше средств, методов и навыков.

Приложения

В приложениях (А, Б, В, Г) рассмотрены некоторые важные вопросы, которые не вошли в основной материал книги.

Здесь вы узнаете о средствах контроля версий и способах их использования в больших проектах, познакомитесь с различными средами разработки на языке PHP, узнаете, как настраивать серверное программное обеспечение, в том числе Apache и Linux.

Общая картина

Авторы надеются, что читатель внимательно протестирует приведенный в книге код, а также программы и проекты, на которые ссылается эта книга.

Ни один разработчик не работает в изоляции, поэтому вместе мы постараемся сделать этот мир немного лучше.

Соглашения

Для облегчения восприятия материала в книге приняты некоторые соглашения и условные обозначения.

В подобных рамках содержится чрезвычайно важная информация, напрямую связанная с излагаемым материалом.

Советы, подсказки, лирические отступления выделяются курсивом.

Кроме того, в тексте книги использованы следующие стили.

- Новые термины выделяются *курсивом*.
- Комбинации клавиш задаются следующим образом: <Ctrl+A>.
- Имена файлов, адреса URL и фрагменты кода внутри текста выделяются так: `persistence.properties`.

- Отдельные фрагменты кода выделяются двумя способами:

Так выделяются новые и важные фрагменты кода.

Менее важные фрагменты или приведенные ранее имеют такой вид.

Исходный код

В процессе изучения этой книги вы можете вводить код вручную либо воспользоваться файлами с исходным кодом, которые можно найти по адресу www.wrox.com. Найдите на этом Web-узле данную книгу и щелкните на ссылке загрузки кода.

Поскольку названия многих книг очень схожи, для поиска лучше воспользоваться номером английского ISBN, для данной книги это 0-7645-7282-2.

Загрузив код, разархивируйте его. Затем можно приступить к его практическому использованию. Исходный код можно также загрузить по адресу www.wrox.com/dynamic/books/download.aspx.

Опечатки

Авторы приложили максимум усилий по устраниению опечаток, обнаруженных в тексте или коде книги. Однако никто не застрахован от ошибок.

Если вы обнаружите ошибку, просим сообщить о ней авторам. При этом вы сэкономите время и усилия других читателей, которые смогут получить более качественную информацию.

Web-ресурс p2p.wrox.com

По адресу p2p.wrox.com вы найдете форумы, на которых книгу можно обсудить с авторами.

По этому адресу находится множество форумов, которые не только помогут вам лучше понять материал этой книги, но и окажутся полезными при разработке собственных приложений. Для участия в форумах выполните следующие действия.

1. Перейдите на страницу p2p.wrox.com и щелкните на ссылке регистрации.
2. Ознакомьтесь с представленными соглашениями и подтвердите свое согласие с ними.
3. Введите нужную информацию.
4. Вы получите электронное сообщение с информацией о том, как верифицировать свою учетную запись и завершить процесс соединения с форумом.

Теперь вы можете отправлять сообщения и отвечать на сообщения других участников форума.

Более подробная информация об этих форумах содержится в разделе часто задаваемых вопросов.

Об авторах

Эд Леки-Томпсон

Эд Леки-Томпсон — основатель профессиональной консалтинговой компании Ashridge New Media, расположенной в пригороде Лондона. Ему приходится решать как бухгалтерские вопросы, так и вопросы разработки программных систем. Он считает себя ярым приверженцем языка PHP, и именно его использует практически во всех своих проектах. Эд имеет более чем шестилетний опыт разработки коммерческого программного обеспечения и архитектуры корпоративных систем на различных платформах, в том числе на основе открытых технологий PHP и Perl в операционных системах Linux и FreeBSD.

Свободное время Эд посвящает быстрой езде по окрестностям Лондона.

Хъяо Эйд-Гудман

Хъяо Эйд-Гудман — член сообществ NYPHP и LispNYC. Он использует язык PHP для повседневной разработки Web-узлов и служб, а также обеспечивает взаимодействие корпоративных приложений с базами данных SQL Server, InterBase/Firebird и MySQL.

Стивен Д. Новицки

Стивен Д. Новицки — директор департамента разработки программных систем в консалтинговой компании штата Калифорния. В настоящее время он занимается планированием ресурсов больших предприятий и системой управления контактами, насчитывающей более 300 тысяч строк объектно-ориентированного кода на языке PHP. Стивен имеет десятилетний опыт разработки крупномасштабных систем на большинстве известных в настоящее время платформ.

Алек Коув

Алек Коув имеет более чем десятилетний опыт разработки Web-приложений. Прежде чем открыть частную консалтинговую фирму, Алек работал главным архитектором в компании по разработке программного обеспечения, где занимался проектированием корпоративных систем и Web-служб. Область его интересов — объектно-ориентированное проектирование, шаблоны и генетическое программирование. В свободное от работы время Алек любит играть на гитаре, сочинять музыку и колесить по Нью-Йорку на велосипеде. Более подробную информацию о нем можно найти по адресу www.cove.org.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Адреса для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

Часть I

Основы разработки объектно-ориентированного программного обеспечения

В ЭТОЙ ЧАСТИ...

**Глава 1. Введение в объектно-ориентированное
программирование**

Глава 2. Унифицированный язык моделирования UML

Глава 3. Объектный подход в действии

Глава 4. Шаблоны проектирования

1

Введение в объектно-ориентированное программирование

Разработка объектно-ориентированного программного обеспечения поначалу может сбить с толку тех разработчиков, которые привыкли использовать структурный (или процедурный) подход. Однако это только на первых порах. В этой главе вы познакомитесь с основами объектно-ориентированного подхода (ООП) и сложной терминологией, которая при этом используется. Здесь также содержится информация о преимуществах и важности ООП, а также о том, как с помощью этого подхода можно ускорить разработку сложных программных систем и существенно упростить их модификацию.

В последующих нескольких главах описанные основные идеи будут рассмотрены более подробно, а кроме того, внимание будет удалено и изучению более сложных вопросов. Если вы уже имели опыт объектно-ориентированной разработки на других языках программирования (а не на PHP 5), то несколько первых глав книги можно и пропустить. Однако в них содержится хороший обзорный материал, поэтому авторы все же рекомендуют с ним познакомиться.

Что такое объектно-ориентированное программирование

При разработке объектно-ориентированных приложений требуется особый способ мышления. Объекты позволяют более полно и точно моделировать те понятия и процессы из реального мира, которыми и должно оперировать разрабатываемое приложение.

Вместо того чтобы рассматривать приложение как механизм управления, который обеспечивает передачу порций данных от одной функции к другой, объектно-ориентированный подход позволяет моделировать приложение как совокупность взаимодействующих объектов, каждый из которых выполняет возложенные на него задачи.

Рассмотрим следующую аналогию. При строительстве дома сантехники имеют дело с трубами, а инженеры-электрики — с проводами. При этом сантехников абсолютно не интересует, на 10 или 20 ампер рассчитана проводка в спальне. Они заняты исключительно своими делами. При этом главному подрядчику требуется, чтобы каждый специалист качественно выполнил свою работу, однако он далеко не всегда вникает в связанные с этим технические детали. То же самое наблюдается и при использовании объектно-ориентированного подхода. Каждый объект “скрывает” детали своей реализации от других объектов и тем самым обеспечивает свою независимость от других компонентов системы. Имеет значение лишь набор сервисов или “услуг”, которые объектом предоставляются.

Классы и объекты, а также способы их использования для разработки программного обеспечения — это основа объектно-ориентированного подхода. В определенном смысле ООП является противоположностью структурному программированию, при использовании которого используются функции и глобальные структуры данных. Как станет видно из последующего материала, по сравнению с процедурным программированием объектно-ориентированный подход предоставляет огромные преимущества, а его новая реализация в языке PHP 5 позволяет создавать крупномасштабные и эффективные приложения.

Преимущества объектно-ориентированного подхода

Одним из основных преимуществ ООП является простота преобразования бизнес-требований предметной области в отдельные программные модули. Поскольку объектно-ориентированный подход позволяет моделировать приложение в терминах объектов реального мира, то всегда можно выявить взаимосвязь людей, предметов и понятий с соответствующими программными классами. Эти классы обладают теми же свойствами и поведением, что и понятия из реального мира, которые они представляют. Поэтому не составит никакого труда определить, как должны быть реализованы и как должны взаимодействовать различные компоненты приложения.

Второе преимущество объектно-ориентированного подхода заключается в возможности повторного использования кода. Зачастую в разных частях приложений нужно использовать одни и те же типы данных. Например, в приложении, предназначенном для управления данными о пациентах в больнице, наверняка потребуется использовать класс *Person* (Пациент). В процессе ухода за больным, как правило, участвует большое количество людей: сам пациент, врачи, медсестры, администраторы, страховые агенты и т.д. При этом при выполнении каких-либо действий необходимо вносить соответствующие записи в историю болезни. (Ее нужно использовать также и при определении возможности выполнения этих действий.) Создав класс с именем *Person*, обладающий свойствами и методами, присущими всем людям, можно добиться повторного использования большой части программного кода. Достичь аналогичного результата при структурном программировании удается далеко не всегда.

Однако, что можно сказать о других приложениях? Во многих ли из них можно воспользоваться уже разработанным классом, предназначенным для обработки информации определенного рода? Таких приложений наверняка достаточно много. Хорошо написанный класс *Person* можно легко перенести из одного проекта в другой

без каких-либо изменений или с минимальной модификацией. Это позволит воспользоваться богатой функциональностью класса, которая была заложена в нем ранее. Именно в возможности повторного использования программного кода в разных приложениях и заключается одно из наиболее существенных преимуществ объектно-ориентированного подхода.

Еще одно преимущество ООП заключается в модульной природе классов. Если в классе `Person` была обнаружена ошибка или в его программный код потребовалось внести изменения, то все изменения достаточно внести в одном месте, поскольку вся функциональность класса определена в одном файле. При этом внесенные изменения коснутся всех компонентов приложения, в которых используется данный класс. Эта возможность позволяет значительно упростить поиск ошибок и добавление новых свойств.

Конкретный пример

В небольших приложениях использование принципа модульности может показаться тривиальной задачей. Однако при работе над крупными проектами преимущества модульной структуры могут оказаться просто неоценимыми. В свое время один из авторов этой книги участвовал в работе над приложением на основе структурного подхода, программный код которого составлял более 200 000 строк кода на языке PHP. При этом около 65% времени, которое тратилось на поиск ошибок и отладку кода, уходило на поиск требуемых функций и определение типов данных, которые использовались для их взаимодействия. Позже код приложения был переписан с использованием объектно-ориентированного подхода, что позволило значительно уменьшить размер программного кода. Если бы приложение с самого начала разрабатывалось с учетом принципов ООП, то можно было бы не только уменьшить время его разработки, но и существенно снизить количество ошибок (чем меньше код — тем меньше проблем) и, следовательно, ускорить сам процесс его отладки.

Поскольку объектно-ориентированный подход заставляет особое внимание уделять структуре кода, то ее изучение новыми разработчиками значительно упрощается. Кроме того, в этом случае гораздо проще определить то место, куда нужно добавить новую функциональность.

Как правило, в работе над крупными проектами обычно участвует группа разработчиков с разным уровнем подготовки и опыта. И в этом случае объектно-ориентированный подход оказывается гораздо полезнее, чем структурный. Так, объекты скрывают детали реализации от своих пользователей. Вместо того чтобы тратить много времени на анализ сложных структур данных и осмысление бизнес-логики, начинающие разработчики команды могут познакомиться с краткой документацией и сразу же приступить к использованию уже разработанных классов.

Однако снова вернемся к рассмотренному выше примеру. Для достаточно глубокого изучения приложения, разработанного с использованием структурного подхода, новым членам группы разработчиков требовалось около двух месяцев. Однако после его преобразования с учетом принципов ООП для этого вполне достаточно было и нескольких дней. После краткого знакомства с используемыми объектами и классами разработчики-новички могли в полную силу включаться в процесс разработки. Они быстро научились оперировать сложными объектами, поскольку теперь уже не было никакой необходимости знакомиться с нюансами их внутренней реализации.

Теперь, после краткого знакомства с особенностями и преимуществами объектно-ориентированного подхода, можно перейти к изучению следующих нескольких разделов,

в которых рассматриваются фундаментальные понятия и принципы, лежащие в его основе. После внимательного изучения двух последующих глав все преимущества ООП станут еще более очевидными.

Основные принципы и понятия ООП

В этом разделе речь пойдет об основных принципах и терминах объектно-ориентированного программирования, а также об основных способах их совместного использования. В главе 3 “Объектный подход в действии” вы узнаете, как принципы ООП реализованы в языке PHP 5. В частности, будут рассмотрены следующие вопросы.

- *Классы* (class) представляют собой “каркас” для создания объектов. В классах содержится реальный программный код, который определяет атрибуты и методы, обеспечивающие функционирование приложения.
- *Объекты* (object) являются экземплярами классов и содержат в себе все необходимые данные и информацию о состоянии, которые требуются для функционирования приложения.
- *Наследование* (inheritance) — это механизм определения новых классов определенного типа, которые являются подтипов (разновидностью) другого класса. (Точно так же, как квадрат является разновидностью прямоугольника.)
- *Полиморфизм* (polymorphism) позволяет определять классы, относящиеся к классам нескольких категорий. (Как, например, автомобиль можно рассматривать как “некоторый предмет с двигателем” или как “нечто с колесами”).
- *Интерфейсы* (interface) предоставляют возможность определения соглашения о том, что в объекте может быть реализован какой-либо метод, без конкретного описания его реализации.
- *Инкапсуляция* (encapsulation) обеспечивает защищенность внутренних данных объектов.

Если какие-либо из перечисленных терминов на первый взгляд покажутся трудными для понимания, не расстраивайтесь. В последующих разделах все они рассматриваются достаточно подробно. Более того, вполне возможно, что полученные знания позволят кардинально изменить подход к разработке программного обеспечения, который используется вами в настоящее время.

Классы

В реальном мире все объекты обладают свойствами и поведением. Так, каждый автомобиль имеет свой цвет, вес, производителя, а также бак для хранения бензина определенной емкости. Все эти характеристики являются свойствами автомобиля. В то же время автомобиль может набирать скорость, останавливаться, указывать направления поворотов и использовать звуковой сигнал. Это определяет его поведение. При этом следует заметить, что приведенные выше свойства и аспекты поведения являются общими для всех автомобилей. Однако в то же время разные автомобили могут быть покрашены в различный цвет. С помощью конструкции класса объектно-ориентированный подход позволяет перейти к понятию “автомобиль”, которое обладает всеми описанными характеристиками. В общем случае класс представляет собой фрагмент кода, содержащий описание переменных и функций. Это описание

определяет свойства и поведение, присущие всем элементам некоторого множества. В данном случае класс с именем `Car` (Автомобиль) будет описывать свойства и поведение, которыми обладают все автомобили.

В терминах ООП свойства класса описываются посредством *свойств* или *атрибутов* (property). Каждый атрибут имеет имя и значение, которое может либо изменяться, либо оставаться неизменным. Например, класс `Car` наверняка будет содержать такие атрибуты, как `color` (цвет) и `weight` (вес). Несмотря на то, что автомобиль можно перекрасить в новый цвет, его вес (без пассажиров и багажа) должен всегда оставаться постоянным.

Часть свойств описывает состояние объекта. Состояние объекта определяется теми свойствами, значения которых изменяются из-за наступления определенных событий, а не вследствие их непосредственной модификации. Например, в приложении для моделирования автомобиля класс `Car` наверняка будет содержать атрибут `velocity` (скорость). При этом абсолютно очевидно, что скорость автомобиля не может меняться сама по себе. Ее значение зависит от количества горючего, которое было подано в двигатель, от параметров самого двигателя, а также от характера местности, по которой движется автомобиль.

Поведение класса описывается с помощью *методов*. Методы класса синтаксически эквивалентны традиционным функциям. Подобно обычным функциям, методы могут получать любое количество параметров, значение каждого из которых должно соответствовать заданному типу данных. При этом в качестве параметров могут использоваться как внешние данные, так и атрибуты самого класса. Свойства класса полезно использовать для уведомления о выполнении тех или иных действий другими методами (например, перед выполнением требуемых действий метод `accelerate()` (увеличить скорость) должен проверить наличие горючего в баке) или для изменения состояния объекта путем модификации значения соответствующего атрибута (например, `velocity`).

Объекты

Класс можно рассматривать как каркас для создания объектов. Точно также как один чертеж можно использовать для строительства многих зданий, класс можно использовать для создания нескольких его экземпляров — объектов. Однако вполне очевидно, что на чертеже не определяются такие детали, как цвет стен или тип напольного покрытия. На нем должны лишь присутствовать все элементы, которые имеются в каждом здании. Классы функционируют аналогичным образом. В классе определяются свойства и поведение, которыми будут обладать все объекты, но не их значения. Объект представляет собой конкретную сущность, которая создана на основе описания класса. Например, `дом` (как понятие) является классом. А конкретный дом, в котором вы живете (конкретная реализация понятия “дом”), — объектом.

Имея в своем распоряжении чертеж и все необходимые строительные материалы, можно построить дом. Точно также для создания объектов используется класс. Этот процесс называется *инстанцированием* (instantiation). Для инстанцирования объекта необходимо следующее.

- ❑ Определение адреса в оперативной памяти, по которому объект будет загружен. Эта операция автоматически выполняется интерпретатором PHP.
- ❑ Наличие данных, которые будут использованы для инициализации значений атрибутов. Требуемую информацию можно извлечь из базы данных, текстового файла, другого объекта или источника.

Класс не может содержать значения атрибутов или иметь состояние. Ими могут обладать только объекты. Прежде чем приступать к поклейке обоев, на основе чертежа нужно построить сам дом. То же самое происходит и при использовании объектно-ориентированного подхода. Перед тем как использовать свойства объекта или вызывать его методы, нужно его инстанцировать на основе класса. Работа с классами выполняется в процессе проектирования приложения. Именно в это момент в их методы и атрибуты можно вносить изменения. Объекты же используются во время выполнения приложения, когда для всех их атрибутов заданы требуемые значения. Неподготовленным разработчикам иногда трудно определить различие между *классом* и *объектом*. Эти термины зачастую вводят новичков в заблуждение.

После инстанцирования объекта его можно использовать для реализации бизнес-логики приложения. Рассмотрим, как это можно осуществить с использованием языка PHP.

Создание класса

Рассмотрим простой пример. Создадим файл `class.Demo.php` со следующим фрагментом кода.

```
<?php
class Demo {
}
?>
```

Вот и создан класс `Demo`. Не очень впечатляет, не так ли? Однако в приведенном примере представлен базовый синтаксис объявления нового класса на языке PHP. В коде используется ключевое слово `class`, после которого указано имя класса и пара фигурных скобок (`{ }`), которые определяют начало и конец описания класса.

Для организации и именования файлов с исходным кодом очень важно использовать универсальные соглашения. Одно из хороших правил заключается в размещении каждого класса в отдельном файле с именем вида `class.[ИмяКласса].php`.

Инстанцировать объект класса `Demo` можно следующим образом.

```
<?php
require_once('class.Demo.php');
$objDemo = new Demo();
?>
```

Перед созданием объекта убедитесь, что интерпретатору PHP известно расположение файла с описанием класса (в данном случае `class.Demo.php`). Затем воспользуйтесь оператором `new` и укажите имя класса в круглых скобках. При этом возвращаемое значение будет присвоено новой переменной `$objDemo`. Теперь можно использовать созданный объект, вызывать его методы, а также проверять или модифицировать значения его атрибутов (если, конечно, они существуют).

Хотя созданный класс `Demo` практически ничего не умеет, с синтаксической точки зрения он создан абсолютно правильно.

Добавление метода

С практической точки зрения класс `Demo` является бесполезным. Поэтому рассмотрим, как в PHP можно добавлять методы. Помните, что метод класса представляет собой ни что иное, как простую функцию. Добавив функцию внутрь круглых скобок, мы фактически добавляем новый метод класса. Рассмотрим пример.

```
<?php

class Demo {

    function sayHello($name) {
        print "Здравствуй $name!";
    }
}

?>
```

Теперь объект, созданный на основе класса `Demo`, может выдавать сообщение с приветствием всем тем, кто вызывает метод `sayHello()`. Для доступа к только что созданному методу необходимо использовать специальный оператор `->`. Например:

```
<?php

require_once('class.Demo.php');

$objDemo = new Demo();

$objDemo->sayHello('Стив');

?>
```

Теперь объекты класса `Demo` могут выводить на экран сообщение с приветствием. Для доступа к методам и атрибутам всех объектов используется оператор `->`.

Те, кто применял объектно-ориентированный подход в других языках программирования, могут обратить внимание, что оператор `->` всегда используется для доступа к методам и атрибутам объектов. Реализация ООП в языке PHP не предусматривает использования оператора “точка” (.) .

Добавление атрибутов

Добавление атрибутов класса выполняется столь же просто, как и методов. Для этого следует объявить переменную внутри класса, которая будет содержать значения атрибутов. Так, если в структурном подходе необходимо сохранить значение, оно присваивается переменной. Точно также в ООП для хранения значений атрибутов используются переменные. Они объявляются внутри фигурных скобок (обычно вначале), в которых заключен код класса. При этом имя переменной соответствует имени атрибута. Например, для атрибута `color` (цвет) будет использована переменная с именем `$color`.

Откройте файл с именем `class.Demo.php` и добавьте выделенные строки кода.

```
<?php

class Demo {
```

```
    public $name;
```

```

function sayHello() {
    print "Здравствуй $this->name!";
}

?>

```

Новая переменная с именем `$name` — это все, что нужно для добавления в класс `Demo` атрибута с именем `name` (имя). Для доступа к атрибуту используется оператор `->` рядом с именем атрибута. В модифицированном методе `sayHello()` осуществляется доступ к значению атрибута `name`.

Создайте файл с именем `testdemo.php` и добавьте следующие строки кода.

```

<?php

require_once('class.Demo.php');

$objDemo = new Demo();
$objDemo->name = 'Стив';

$objAnotherDemo = new Demo();
$objAnotherDemo->name = 'Эд';

$objDemo->sayHello();
$objAnotherDemo->sayHello();

?>

```

Сохраните файл и запустите его в Web-браузере. На экране будут отображены две строки: “Здравствуй Стив!” и “Здравствуй Эд!”.

Ключевое слово `public` (открытый) используется для обеспечения доступа к переменной класса извне. Дело в том, что в общем случае некоторые переменные-члены класса могут существовать только для внутреннего использования и не должны быть доступны из внешнего кода приложения. В данном примере нужно обеспечить возможность присвоения и получения значения атрибута `name`. Обратите внимание на изменения, внесенные в описание метода `sayHello()`. Теперь вместо внешней информации (т.е. аргументов) этот метод использует данные, извлеченные из атрибута.

Для этого применяется переменная `$this`, которая позволяет объекту получать информацию о самом себе. Действительно, можно создать множество экземпляров одного и того же класса. Но поскольку имя объекта заранее неизвестно, переменная `$this` указывает на текущую реализацию класса.

В предыдущем примере при первом вызове метода `sayHello()` приветствие относится к Стиву, а при втором — к Эду. Это происходит вследствие того, что переменная `$this` позволяет любому объекту получить доступ к атрибутам и методам без знания имени переменной-объекта во внешнем приложении. Ранее уже указывалось, что значения атрибутов могут влиять на функциональность методов. Действительно, в примере с автомобилем работа метода с именем `accelerate()` класса `Car` зависит от количества оставшегося в баке бензина. Поэтому при реализации данного метода наверняка потребуется доступ к соответствующему атрибуту `$this->amountOfFuel`.

При доступе к атрибутам класса следует использовать один символ доллара \$, т.е. \$объект->имя_атрибута, а не \$объект->\$имя_атрибута. Это правило обычно вводит в заблуждение новичков PHP. Итак, доступ к открытому атрибуту объекта осуществляется посредством \$объект->имя_атрибута.

Кроме переменных, в которых хранятся значения атрибутов класса, можно объявлять вспомогательные переменные, которые будут использоваться для внутренних операций. Оба вида переменных относят к *внутренним переменным-членам* класса (internal member variables). Некоторые из них доступны извне в виде атрибутов, а некоторые строго предназначены для внутреннего использования. Например, если по каким-то причинам в классе `Car` необходимо извлекать информацию из базы данных, будет использована соответствующая переменная-член класса. Однако при этом очевидно, что она не будет являться атрибутом автомобиля, а нужна только для совершения классом определенных действий.

Защищенный доступ к переменным-членам класса

Как видно из предыдущего примера, атрибуту с именем `name` можно присваивать любые значения, включая объекты, массивы чисел, идентификаторы файлов, а также любые бессмысленные значения. При этом нет никакой возможности осуществить проверку данных или обновить значения других переменных при изменении значения данного атрибута.

Для решения этой проблемы атрибуты классов следует изменять или использовать с помощью функций с именами `get [имя_атрибута] ()` и `set [имя_атрибута] ()`. Эти функции называются *методами доступа* (accessor method). Рассмотрим пример их использования.

Внесите выделенные изменения в файл с именем `class.Demo.php`.

```
<?php
class Demo {

    private $_name;

    public function sayHello() {
        print "Здравствуй {$this->getName()}!";
    }

    public function getName() {
        return $this->_name;
    }

    public function setName($name) {
        if(!is_string($name) || strlen($name) == 0) {
            throw new Exception("Неверное значение имени");
        }
        $this->_name = $name;
    }
}
?>
```

Измените файл с именем `testdemo.php` следующим образом.

```
<?php
require_once('new_class.Demo.php');
$objDemo = new Demo();

$objDemo->setName('Стив');
$objDemo->sayHello();

$objDemo->setName(37); //будет выдана ошибка
?>
```

Как видно из примера, область доступа атрибута `name` изменена с открытой (`public`) на закрытую (`private`). При этом к имени соответствующей переменной добавлен символ подчеркивания (`_`). Такое правило рекомендуется использовать для именования закрытых переменных и функций класса. (Это всего лишь соглашение, и использовать его в PHP не обязательно.) Ключевое слово `private` (закрытый) защищает переменные объекта от модификаций, т.е. доступ к ним извне класса запрещен. Таким образом, для работы с переменной необходимо использовать методы доступа `setName()` и `getName()`. При этом в классе должна осуществляться проверка значений, которые присваиваются переменным. В данном примере срабатывает исключение, когда атрибуту с именем `name` присваивается некорректное значение. Кроме того, к имени функций был добавлен спецификатор `public`. Если для переменных и функций класса явно не определен уровень доступа, по умолчанию используется именно открытый. Однако стоит при этом заметить, что явное задание уровня доступа к переменным и функциям-членам класса считается хорошим стилем программирования.

В общем случае существует три уровня (или области) доступа к атрибутам и методам класса: открытый (`public`), закрытый (`private`) и защищенный (`protected`). Открытые члены класса доступны в любом месте программы. В свою очередь закрытые члены доступны только в рамках самого класса. Обычно они предназначены для внутреннего использования и содержат информацию, например, о подключении к базе данных или разнообразных настройках. Защищенные члены класса доступны как в рамках самого класса, так и классам-наследникам. (Механизм наследования более подробно рассмотрен далее в этой главе.)

В общем случае использование методов доступа для всех атрибутов класса позволяет существенно упростить проверку данных, добавление новой бизнес-логики приложения, а также внесение других изменений в объекты в будущем. Даже если текущие требования не предусматривают проверку данных, атрибуты классов все равно следует реализовывать на основе функций `set` и `get`.

Всегда используйте методы доступа для всех атрибутов класса. Тогда последующие изменения в бизнес-логике приложения или требованиях по проверке данных будет реализовать намного легче.

Инициализация объектов

При создании объекта некоторого класса обычно необходимо задать специальные начальные значения. К ним могут относиться, например, получение информации из базы данных либо же инициализация значений соответствующих атрибутов. Для выполнения всех действий, необходимых для корректного создания объекта, используется специальный метод, называемый *конструктором* (`constructor`). В языке PHP конструктор реализуется посредством функции с именем `__construct()`, которая автоматически вызывается интерпретатором PHP при создании объекта.

В качестве примера изменим класс `Demo` следующим образом.

```
<?php

class Demo {

    private $name;

    public function __construct($name) {
        $this->name = $name;
    }

    public function getName() {
        return $this->name;
    }
}
```

```

    $this->name = $name;
}

function sayHello() {
    print "Здравствуй $this->name!";
}
?>

```

Функция с именем `__construct()` будет автоматически вызвана при создании экземпляра класса `Demo`.

Для пользователей PHP 4. В версии PHP 4 конструкторы имеют те же имена, что и сам класс. В версии PHP 5 внесены изменения, направленные на унификацию использования конструкторов. Для обеспечения обратной совместимости при создании объекта интерпретатор PHP сперва ищет функцию с именем `__construct()`. Если она не найдена, выполняется поиск функции с именем, аналогичным имени класса (public function `Demo()` в предыдущем примере).

Если при разработке класса не требуется выполнять инициализацию, тогда нет необходимости создавать конструктор. Как видно из реализации первой версии класса `Demo`, интерпретатор PHP автоматически выполняет все необходимые действия для создания объекта. Создавайте конструкторы только в случае необходимости.

Уничтожение объектов

Созданные переменные объектов удаляются из памяти при выполнении одного из следующих условий: при завершении выполнения запрашиваемой страницы приложения, при выходе за пределы области видимости переменной или при явном удалении. В языке PHP 5 можно проследить за процессом удаления объекта и выполнить при этом определенные действия. Для этого необходимо в классе создать специальную функцию с именем `__destruct()` без параметров. Эта функция автоматически вызывается интерпретатором PHP непосредственно перед удалением объекта (если, конечно, она была создана).

Использование функции `__destruct()` позволяет выполнить последние действия над объектом, в частности по освобождению памяти (“уборке мусора”), закрытию используемых файлов или закрытию ненужных подключений к базе данных.

В последующем примере при удалении объекта информация о времени его жизни помещается в соответствующий файл аудита. Применяемый метод может оказаться полезным при исследовании производительности системы (особенно при использовании объектов, требующих существенных ресурсов памяти и процессора) и определении путей ее повышения.

В большинстве примеров, связанных со взаимодействием с базой данных, будет использоваться СУБД PostgreSQL. Авторы полагают, что улучшенные свойства, поддержка транзакций, а также робастные механизмы хранения данных дают ей существенные преимущества по сравнению с базой данных MySQL и другими открытыми СУБД при разработке масштабных программных приложений. Если у вас не оказалось в наличии СУБД PostgreSQL, сделать соответствующие изменения в используемое программное обеспечение не составит труда.

Итак, создайте таблицу с именем `widget`, используя следующий SQL-запрос:

```
CREATE TABLE "widget" (
    "widgetid" SERIAL PRIMARY KEY NOT NULL,
    "name" varchar(255) NOT NULL,
    "description" text
);
```

Внесите в нее некоторую информацию.

```
INSERT INTO "widget" ("name", "description")
VALUES ('Foo', 'This is a footacular widget!');
```

Теперь создайте файл с именем class.Widget.php и следующими строками кода.

```
<?php

class Widget {

    private $id;
    private $name;
    private $description;  private $hDB;
    private $needsUpdating = false;

    public function __construct($widgetID) {
        //Параметр widgetID является первичным ключом для
        //записей базы данных с именем widget

        //Создание идентификатора подключения к базе данных и ее сохранение
        //в закрытой переменной-члене класса
        $this->hDB = pg_connect('dbname=parts user=postgres');
        if(! is_resource($this->hDB)) {
            throw new Exception('Невозможно подключиться к базе данных.');
        }

        $sql = "SELECT \"name\", \"description\" FROM widget WHERE
                widgetid = $widgetID";
        $rs = pg_query($this->hDB, $sql);
        if(! is_resource($rs)) {
            throw new Exception("Произошла ошибка при выборе базы данных.");
        }

        if(! pg_num_rows($rs)) {
            throw new Exception('Указанная запись в базе данных отсутствует!');
        }

        $data = pg_fetch_array($rs);
        $this->id = $widgetID;
        $this->name = $data['name'];
        $this->description = $data['description'];
    }

    public function getName() {
        return $this->name;
    }

    public function getDescription() {
        return $this->description;
    }

    public function setName($name) {
        $this->name = $name;
        $this->needsUpdating = true;
    }
}
```

```

public function setDescription($description) {
    $this->description = $description;
    $this->needsUpdating = true;
}

public function __destruct() {
    if(! $this->needsUpdating) {
        return;
    }

    $sql = 'UPDATE "widget" SET ';
    $sql .= "\"name\" = '" . pg_escape_string($this->name) . "', ";
    $sql .= "\"description\" = '" . pg_escape_string($this->description) . "'";
    $sql .= "WHERE widgetID = " . $this->id;

    $rs = pg_query($this->hDB, $sql);
    if(! is_resource($rs)) {
        throw new Exception('Произошла ошибка при обновлении базы данных.');
    }

    //Все действия по работе с базой данных завершены. Разрыв подключения.
    pg_close($this->hDB);
}
?>

```

В конструкторе класса `Widget` устанавливается подключение к базе данных с именем `parts`, используя учетную запись привилегированного пользователя `postgres` (учетная запись по умолчанию). Затем идентификатор подключения сохраняется в закрытой переменной-члене класса `$hDB` для последующего использования. Значение идентификатора записи `$widgetID`, которое передается конструктору в качестве аргумента, используется для построения запроса SQL с целью получения необходимой информации из базы данных. Эта информация также сохраняется в закрытых переменных-членах класса (`$name` и `$description`), доступ к которым осуществляется посредством методов `get` и `set`. Заметьте, что если по каким-либо причинам возникает ошибка, конструктор класса генерирует соответствующее исключение. Поэтому при создании объекта класса `Widget` помещайте его внутри блока `try...catch`.

Два метода доступа с именами `getName()` и `getDescription()` позволяют получить значения закрытых переменных-членов класса. В свою очередь, методы `setName()` и `setDescription()` используются для присваивания им конкретных значений. Заметьте, что в этом случае переменной с именем `$needsUpdating` присваивается значение `true` (истина), т.е. необходимо выполнить обновление базы данных. Если же никаких изменений не произошло, обновление базы данных производить не требуется.

Чтобы проверить работу созданного класса, создайте файл с именем `testWidget.php` и следующими строками кода.

```

<?php

require_once('class.Widget.php');

try {
    $objWidget = new Widget(1);

    print "Имя элемента: " . $objWidget->getName() . "<br>\n";
    print "Описание элемента: " . $objWidget->getDescription() . "<br>\n";
}

```

```

    $objWidget->setName('Bar');
    $objWidget->setDescription('Это элемент Bar!');
} catch (Exception $e) {
    die("Произошла ошибка: " . $e->getMessage());
}
?>

```

Запустите этот файл в Web-браузере. При первом запуске результат будет таким.

Имя элемента: Foo

Описание элемента: Это элемент Foo!

При любом последующем запуске на экране будут отображены следующие строки.

Имя элемента: Bar

Описание элемента: Это элемент Bar!

Заметьте, насколько мощной является применяемая технология. Используя всего лишь несколько строк кода, можно получить информацию об объекте из базы данных, изменить его свойства и автоматически внести все изменения в базу данных. Если же никаких изменений не произошло, нет необходимости обращаться к базе данных. Тем самым уменьшается нагрузка на сервер базы данных и повышается эффективность работы приложения.

Стоит при этом заметить, что пользователям класса не обязательно знать детали его реализации. Например, если разработчиками класса `Widget` являются более опытные программисты команды разработчиков, младшие члены команды смогут без проблем пользоваться им. При этом они могут не знать язык запросов SQL и не понимать, каким образом осуществляется взаимодействие с базой данных. Более того, можно запросто изменить источник данных, скажем, с используемой СУБД PostgreSQL на MySQL или файлы в формате XML. В этом случае пользователям класса не придется изменять ни одной строки кода.

Следующий шаг в изучении рассматриваемой концепции будет сделан в главе 7. В ней вы познакомитесь с универсальной версией класса `Widget` — классом `GenericObject`, который можно использовать без изменений в качестве базового практически в любом проекте.

Наследование

Допустим необходимо разработать приложение для автоматизации процесса продажи автомобилей. Тогда вам наверняка понадобиться создать классы с именами `Sedan` (автомобиль типа “седан”), `PickupTruck` (“пикап”) и `Minivan` (“минивэн”), которые соответствуют реальным типам автомобилей. При этом приложение должно выдавать информацию не только о количестве автомобилей тех или иных типов, имеющихся в наличии, но и подробные данные об их характеристиках. Это позволит продавцам подать полную информацию заинтересованным покупателям.

Так, например, автомобиль типа “седан” имеет четыре двери, и важной для него является информация о наличии свободного пространства сзади и вместимости багажника. В свою очередь, в “пикапе” не предусмотрено место для багажа, однако при этом имеется место для груза определенного объема. Кроме того, для “пикапа” определен максимальный вес транспортных средств, которые могут быть отбуксированы с его помощью. Что касается автомобилей типа “минивэн”, то для них существенной является информация о количестве раздвижных дверей (одна либо две), а также о количестве мест внутри.

Однако каждый из приведенных типов транспортных средств является не чем иным, как одним из видов автомобилей. Поэтому очевидно, что все они будут обладать общими характеристиками, такими как цвет, производитель, модель, год выпуска, номер регистрации и т.д. Для того чтобы соответствующие классы имели одни и те же атрибуты, можно просто скопировать одинаковые строки кода в файлы, содержащие описание разных классов. Однако, как уже упоминалось ранее в данной главе, одним из преимуществ объектно-ориентированного подхода является повторное использование кода. Поэтому, безусловно, вам не придется копировать одинаковые участки кода в разные файлы. Вместо этого можно воспользоваться механизмом *наследования* (*inheritance*), который позволяет повторно использовать атрибуты и методы одного класса в другом.

Используя наследование, можно определить базовый (родительский) класс — в рассматриваемом примере класс *Automobile* (автомобиль), — а все остальные классы (разные виды автомобилей) будут являться его наследниками. При этом они будут обладать всеми атрибутами и методами, содержащимися в основном классе. Например, класс *Sedan* будет являться подтипов (наследником) класса *Automobile*, автоматически наследуя его функциональность. При этом нет необходимости копировать одинаковые участки кода. Все, что нужно сделать, — это определить свойства и методы, характерные только для класса *Sedan* (т.е. автомобилей типа “седан”). Таким образом, при использовании механизма наследования необходимо определить только разницу между родительским классом и классами-наследниками. В свою очередь, сходство между классами будет автоматически реализовано с помощью механизма наследования.

Возможность повторного использования кода является не единственным преимуществом наследования. Есть еще одно. Допустим в разрабатываемом приложении имеется класс с именем *Customer* (покупатель), для которого определен метод *buyAutomobile()* (купить автомобиль). Очевидно, что в качестве его аргумента будет использован объект класса *Automobile*. Одна из задач метода *buyAutomobile()* — документирование информации о покупке автомобиля, а также уменьшение на единицу количества автомобилей, имеющихся в наличии. Поскольку объекты классов *Sedan*, *PickupTruck* и *Minivan* являются подтипами класса *Automobile*, их также можно передавать в качестве аргументов метода *buyAutomobile()*. Действительно, приведенные классы наследуют функциональность одного и того же родительского класса. Поэтому они будут обладать одинаковым базовым набором атрибутов и методов. Поскольку для метода *buyAutomobile()* важно наличие свойств, характерных для класса *Automobile*, в качестве его аргументов можно передавать объекты любых классов, которые являются наследниками класса *Automobile*.

Приведем другой пример из животного мира, связанный с видом кошачьих. Все кошки обладают общими свойствами, такими как вес, цвет шерсти, длина усов и скорость бега. Кроме того, все они питаются, спят, мурлыкают и охотятся. Однако львы, например, имеют гриву определенной длины (по крайней мере, самцы) и к тому же рычат. У гепарда на шерсти есть пятна. В свою очередь, домашние кошки не обладают этими свойствами, хотя все эти животные относятся к одному и тому же виду — кошачьим.

Для того чтобы указать, что один класс является подтипов другого, в языке PHP существует ключевое слово *extends*. Оно указывает, что объявляемый класс будет наследовать все атрибуты и методы родительского класса. Все, что остается сделать, — так это реализовать дополнительную функциональность, характерную только для разрабатываемого класса.

Допустим необходимо разработать приложение, моделирующее работу зоопарка. Тогда наверняка в нем будут присутствовать классы с именами *Cat* (кот), *Lion* (лев)

и Cheetah (гепард). Прежде чем перейти к непосредственному кодированию, следует разработать иерархию классов приложения в виде диаграмм UML. Их использование поможет упростить последующее написание кода и создание документации для разрабатываемого приложения. (Более подробно язык унифицированного моделирования UML рассматривается в главе 2. Так что не переживайте, если какие-то из обозначений будут непонятны.) Для рассматриваемого примера диаграмма классов будет содержать родительский класс Cat и два класса-наследника: Lion и Cheetah, как показано на рис. 1.1.

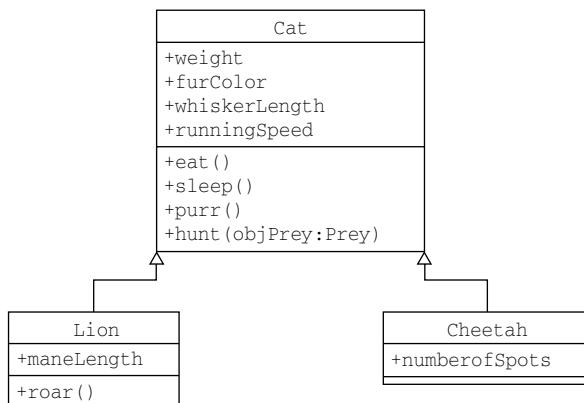


Рис. 1.1.

При этом стоит заметить, что в классе Lion дополнительно определены атрибут maneLength (длина гривы) и метод roar() (рычать), а в классе Cheetah — атрибут numberOfSpots (количество пятен).

Реализация класса Cat на PHP будет иметь следующий вид (файл с именем `class.Cat.php`).

```

<?php
class Cat {
    public $weight;           //в кг
    public $furColor;
    public $whiskerLength;
    public $maxSpeed;         //в км/ч

    public function eat() {
        //описание того, как кошка питается...
    }

    public function sleep() {
        //описание того, как кошка спит...
    }

    public function hunt(Prey $objPrey) {
        //описание того, как кошка охотится за объектом типа Prey
        //класс Prey здесь не определен...
    }

    public function purr() {
        print "муурр..." . "\n";
    }
}
?>
  
```

В этом классе определены атрибуты и методы, характерные для всех животных вида кошачьих. Для того чтобы создать классы Lion и Cheetah, можно было бы просто скопировать соответствующие участки кода из класса Cat. Однако в таком случае возникает две проблемы. Первая состоит в следующем: если в классе Cat допущена ошибка, ее придется исправлять не только в этом классе, но и в классах Lion и Cheetah. Соответственно, придется затратить больше времени на отладку всего приложения (в то время, как уменьшение времени на разработку приложения является одним из основных преимуществ ООП).

Вторая проблема заключается в следующем. Предположим в некотором классе определен метод наподобие следующего.

```
public function petTheKitty(Cat $objCat) {
    $objCat->purr();
}
```

Возможно, идея приласкать льва или гепарда не является такой уж безопасной, поскольку они явно начнут рычать, если вы приблизитесь к ним на достаточно близкое расстояние. Очевидно, что в качестве аргумента данной функции придется передавать объекты как класса Lion, так и класса Cheetah, т.е. если создавать эти классы путем копирования кода, то метод petTheKitty() придется определять отдельно для каждого из этих классов.

Таким образом, необходимо применять другой подход для разработки классов Lion и Cheetah. Наиболее эффективный способ заключается в использовании механизма наследования. Используя ключевое слово extends и указав имя родительского класса, можно создать новый класс, который будет обладать теми же свойствами, что и базовый класс (в рассматриваемом примере класс Cat). Единственное, что остается сделать — это добавить дополнительные элементы. Например:

```
<?php
require_once('class.Cat.php');

class Lion extends Cat {
    public $maneLength; //в см

    public function roar() {
        print "Р-р!";
    }
}
?>
```

Вот и все! Теперь, используя класс Lion, можно выполнять следующие действия.

```
<?php
include('class.Lion.php');

$objLion = new Lion();
$objLion->weight = 200;      //кг = ~450 фунтов
$objLion->furColor = 'коричневый';
$objLion->maneLength = 36;   //см = ~14 дюймов
$objLion->eat();
$objLion->roar();
$objLion->sleep();
?>
```

Из приведенного фрагмента кода можно увидеть, что объекты класса Lion имеют доступ к атрибутам и методам родительского класса Cat (без какого-либо копирования строк кода). Это происходит благодаря использованию ключевого слова extends. Оно

позволяет PHP автоматически поддерживать функциональность класса `Cat`, наряду с атрибутами и методами, характерными только для класса `Lion`. Кроме того, в данном случае объекты класса `Lion` будут также являться объектами класса `Cat`. Поэтому функцию `petTheKitty()` можно вызывать для объектов класса `Lion`, несмотря на то, что она объявлена для переменных типа `Cat`.

```
<?php
    include('class.Lion.php');
    $objLion = new Lion();

    $objPetter = new Cat();
    $objPetter->petTheKitty($objLion);
?>
```

При использовании описанного подхода для создания классов любые изменения, внесенные в класс `Cat`, будут автоматически отображены в классе-наследнике `Lion`, т.е. отладка кода, изменения в описании функций, добавление новых атрибутов и методов — все это будет унаследовано подклассом родительского класса. Очевидно, что в таком случае в грамотно спроектированной иерархии объектов приложения выполнение отладки кода и добавление новых компонентов становится достаточно простой задачей. При этом небольшие изменения в родительском классе могут значительным образом повлиять на функциональность всего приложения.

В последующем примере будет показано, как с помощью обычного конструктора можно расширить функциональность класса.

Для этого создайте новый файл с именем `class.Cheetah.php` и следующим содержимым.

```
<?php
    require_once('class.Cat.php');

    class Cheetah extends Cat {
        public $numberOfSpots;

        public function __construct() {
            $this->maxSpeed = 100;
        }
    }
?>
```

Ведите в файл `testcats.php` следующие строки кода.

```
<?php
require_once('class.Cheetah.php');

function petTheKitty(Cat $objCat) {
    if($objCat->maxSpeed < 5) {
        $objCat->purr();
    } else {
        print "Не могу погладить кошку - она движется со
              скоростью ".$objCat->maxSpeed." км/ч!";
    }
}

$objCheetah = new Cheetah();
petTheKitty($objCheetah);

$objCat = new Cat();
petTheKitty($objCat);
?>
```

В класс `Cheetah` добавлена открытая переменная-член с именем `$numberOfSpots` и конструктор, который прежде не был определен в родительском классе `Cat`. Теперь при создании объекта класса `Cheetah` атрибуту `maxSpeed` (который наследуется из класса `Cat`) присваивается начальное значение 100 км/ч (около 60 м/ч), что приблизительно соответствует средней скорости гепарда на коротких дистанциях. При этом заметьте, что в родительском классе `Cat` значение по умолчанию этого атрибута не указано. Поэтому в функции `petTheKitty()` оно принимается равным 0. Действительно, те, кто когда-либо имел домашних кошек, знают, что кошки любят много спать, т.е. их средняя скорость равна нулю!

Расширить функциональность классов-наследников родительского класса можно путем добавления новых функций, атрибутов или даже конструкторов и деструкторов. Это позволяет добавить новые свойства и возможности приложения с наименьшим количеством кода.

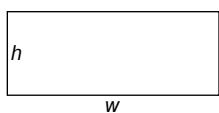
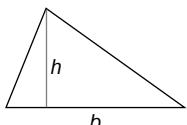
Если можно сказать, что один класс является подтипов другого, то в этом случае следует применять механизм наследования с целью обеспечения возможности повторного использования кода и увеличения гибкости разрабатываемого приложения.

Перегрузка методов

Если некоторый класс является наследником другого, то он неизбежно должен использовать реализацию функций родительского класса. Пусть, например, необходимо разработать приложение, предназначенное для вычисления площади различных геометрических фигур. Очевидно, что в нем будут присутствовать классы с именами `Rectangle` (прямоугольник) и `Triangle` (треугольник). Оба вида фигур относятся к многоугольникам, поэтому соответствующие классы будут наследовать родительский класс с именем `Polygon` (многоугольник).

Для возможности вычисления площади класс `Polygon` будет содержать атрибут с именем `numberOfSides` (количество сторон) и метод `getArea()` (вычислить площадь). Следует заметить, что площадь можно вычислить для любого многоугольника. Однако при этом для разных видов многоугольников можно использовать разные методы. (В общем случае существует универсальная формула для вычисления площади любого многоугольника. Однако с вычислительной точки зрения она менее эффективна, нежели специализированные формулы для конкретных видов многоугольников. Именно они и будут использоваться в рассматриваемом примере.) Например, площадь прямоугольника равна $w \times h$, где w — его ширина, а h — высота. В свою очередь формула для вычисления площади треугольника имеет вид $0,5 \times h \times b$, где h — высота, опущенная на сторону b . На рис. 1.2 приведены разные формулы для вычисления площадей разных видов многоугольников.

$$\text{Площадь треугольника} = (1/2 \times h) \times b$$



$$\text{Площадь прямоугольника} = w \times h$$

Рис. 1.2.

Очевидно, что для каждого класса-наследника класса `Polygon` (т.е. соответствующего вида многоугольника) для вычисления площади эффективнее использовать специализированную формулу, а не формулу базового класса, т.е. фактически необходимо переопределить метод родительского класса и реализовать новый в наследуемом классе.

Так, для класса `Rectangle` следует определить два атрибута с именами `height` (высота) и `width` (ширина) и перегрузить метод `getArea()` родительского класса `Polygon`. Аналогично, для класса `Triangle` необходимо добавить новые атрибуты, которые будут содержать информацию о трех углах, высоте и длине соответствующего основания, а также переопределить метод `getArea()`. Таким образом, используя механизм наследования и перегрузку методов при создании классов-наследников, можно переопределять и уточнять в нем методы базового класса.

Так, функция, которая принимает в качестве аргумента объект типа `Polygon` и используется для вывода площади многоугольника, будет автоматически вызывать метод `getArea()` соответствующего подкласса (т.е. `Rectangle` или `Triangle`). Свойство ООП, позволяющее во время работы приложения автоматически определять, какой именно метод с именем `getArea()` следует вызывать, называется *полиморфизмом* (polymorphism). Полиморфизм позволяет выполнять различные действия в зависимости от текущего (действующего) объекта. В приведенном примере это касалось вызова “разных” методов `getArea()` для вычисления площади геометрических фигур.

Перегружать методы в подклассе следует каждый раз, когда их реализация в базовом классе отлична от требуемой. Такой подход позволяет конкретизировать (специализировать) работу класса-наследника.

Иногда необходимо не только сохранить реализацию некоторого метода родительского класса, но и добавить дополнительную функциональность. Например, если разрабатывается приложение для управления волонтерской организацией, то в нем будет присутствовать класс `Volunteer` (доброволец) с методом `signUp()`. Этот метод будет использоваться для записи пользователей, желающих принять участие в тех или иных проектах. Однако на некоторых пользователей могут распространяться ограничения (например, на участников с криминальным прошлым). Им будет запрещено подписываться на определенные проекты. В этом случае можно создать класс `RestrictedUser` (пользователь с ограничениями) и, используя полиморфизм, перегрузить метод `signUp()` следующим образом: сперва проверять наличие ограничений для учетной записи пользователя на участие в том или ином проекте. Если такие имеются, пользователю будет запрещено подписываться на этот проект. В противном случае следует просто вызвать метод `signUp()` родительского класса для завершения регистрации на проект.

При перегрузке метода родительского класса не обязательно полностью переписывать код данного метода. Можно просто использовать его реализацию и добавить новую, которая характерна для класса-наследника. Такой подход позволяет повторно использовать код и таким образом обеспечивать гибкость приложения, определяемую правилами бизнес-логики.

Возможность одного класса наследовать методы и свойства другого является одним из неотъемлемых свойств объектно-ориентированных систем. Наследование позволяет достичь высокого уровня эффективности и гибкости разрабатываемых приложений.

Итак, вернемся к примеру с геометрическими фигурами. Нам необходимо создать два класса `Rectangle` (прямоугольник) и `Square` (квадрат). Квадрат является специальным видом прямоугольника. Все, что можно делать с прямоугольником, можно выполнять и с квадратом. Однако, поскольку прямоугольник имеет две стороны разной длины, а квадрат всего одну, некоторые действия следует осуществлять по-разному.

Создайте файл с именем `class.Rectangle.php` и следующими строками кода.

```
<?php

class Rectangle {
    public $height;
    public $width;

    public function __construct($width, $height) {
        $this->width = $width;
        $this->height = $height;
    }

    public function getArea() {
        return $this->height * $this->width;
    }
}

?>
```

Это достаточно прямолинейный подход к реализации класса для моделирования прямоугольника. Конструктору передается два параметра (ширина и высота прямоугольника), а метод `getArea()` предназначен для вычисления его площади.

Рассмотрим теперь файл с именем `class.Square.php`.

```
<?php
require_once('class.Rectangle.php');

class Square extends Rectangle {
    public function __construct($size) {
        $this->height = $size;
        $this->width = $size;
    }

    public function getArea() {
        return pow($this->height, 2);
    }
}

?>
```

В данном классе осуществляется перегрузка как конструктора, так и метода `getArea()`. Если прямоугольник является квадратом, его ширина и высота равны между собой. Поэтому конструктору класса `Square` необходимо передавать только один параметр. При этом, если передать несколько параметров, лишние будут проигнорированы.

Интерпретатор PHP не выдает ошибку, если количество аргументов, переданных функции, больше количества аргументов, указанных в ее определении. В некоторых случаях такое поведение приложения является предпочтительным. Если вы хотите узнать больше о работе функций, обратитесь к описанию функции `func_get_args()` в документации по PHP.

Метод `getArea()` класса `Square` также перегружен, несмотря на то, что при использовании реализации `getArea()` родительского класса `Rectangle` для объекта класса `Square` будет получен корректный результат. Это сделано с целью улучшения производительности приложения (хотя в данном случае совсем незначительного). Действительно, при вычислении площади квадрата интерпретатор PHP быстрее возведет в квадрат одну переменную, чем перемножит два передаваемых параметра.

Перегружая конструкторы, деструкторы и методы классов, можно изменять поведение подклассов.

Сохранение функциональности родительского класса

Иногда необходимо сохранить функциональность, предоставляемую родительским классом, т.е. нет необходимости полностью переписывать некоторую функцию — необходимо добавить новые строки кода. Безусловно, можно было бы просто скопировать одинаковый код из родительского класса в класс-наследник. Однако, как уже неоднократно упоминалось, ООП предусматривает иной способ.

Для того чтобы вызвать в подклассе метод родительского класса, следует воспользоваться следующим синтаксисом: `parent::имя_метода`, т.е. если необходимо обеспечить новую функциональность некоторого метода, сперва следует вызвать `parent::имя_метода`, а затем добавить новые строки кода. Расширяя таким образом функциональность класса, метод родительского класса всегда необходимо вызывать первым. Это позволит избежать ошибок.

Вызов метода родительского класса может требовать наличия объекта в определенном состоянии, может изменять его, может изменять значения атрибутов, а также манипулировать внутренней структурой объекта. Поэтому при его перегрузке в классе-наследнике его всегда следует вызывать перед новыми (добавленными) строками кода.

Рассмотрим следующий пример. Пусть имеется два класса `Customer` (покупатель) и `SweepstakesCustomer` (покупатель, участвующий в акции). Предположим в супермаркете установлено приложение, которое в зависимости от проводимой акции использует один из указанных классов. Каждый посетитель магазина имеет свой идентификатор (получаемый из базы данных), а также номер, который указывает, сколько покупателей посетили супермаркет к данному моменту. При этом миллионный покупатель получает приз.

Создайте следующий файл с именем `class.Customer.php`.

```
<?php

class Customer {
    public $id;
    public $customerNumber;
    public $name;

    public function __construct($customerID) {
        //получение информации о покупателе из базы
        //данных

        //В данном примере соответствующие строки кода
        //опущены. Однако в реальном приложении данные
        //будут поступать из базы данных
        $data = array();
        $data['customerNumber'] = 1000000;
        $data['name'] = 'Джэйн Джонсон';

        //Присваивание значений из базы данных атрибутам
        //объекта
        $this->id = $customerID;
        $this->name = $data['name'];
        $this->customerNumber = $data['customerNumber'];
    }
}?
?>
```

Создайте теперь файл с именем `class.SweepstakesCustomer.php`.

```
<?php
require_once('class.Customer.php');

class SweepstakesCustomer extends Customer {
    public function __construct($customerID) {
        parent::__construct($customerID);

        if($this->customerNumber == 1000000) {
            print "Поздравляем $this->name! Вы наш миллионный покупатель! ";
            "Вы выиграли годовую поставку мороженых рыбных палочек!";
        }
    }
}

?>
```

Как функционирует наследование

На основе идентификатора покупателя и информации из базы данных осуществляется инициализация атрибутов класса `Customer`. При этом идентификатор покупателя можно получить, используя номер дисконтной карточки, доступной во многих сетях крупных супермаркетов. Тогда, имея в наличии идентификатор, можно обращаться к базе данных с целью получения всей необходимой информации о покупателе. Кроме того, каждому покупателю присваивается его номер в общем списке посетителей. После этого все данные сохраняются в открытых переменных-членах класса `Customer`.

В свою очередь функциональность конструктора класса `SweepstakesCustomer` была расширена. Сперва в нем вызывается конструктор родительского класса `parent::__construct ($customerID)`, которому в качестве аргумента передается идентификатор покупателя, а затем проверяется значение атрибута `customerNumber`. Если покупатель является миллионным, он выигрывает приз.

Рассмотрим на примере работу класса `SweepstakesCustomer`. Создайте файл с именем `testCustomer.php` и следующим содержимым.

```
<?php

require_once('class.SweepstakesCustomer.php');
//Поскольку этот файл уже включает в себя файл с
//именем class.Customer.php, нет необходимости
//подключать его дополнительно.

function greetCustomer(Customer $objCust) {
    print "Добро пожаловать обратно в магазин $objCust->name!";
}

//Измените значение этой переменной с тем, чтобы
//использовать один из классов Customer или
//SweepstakesCustomer
$promotionCurrentlyRunning = true;

if ($promotionCurrentlyRunning) {
    $objCust = new SweepstakesCustomer(12345);
} else {
    $objCust = new Customer(12345);
}
```

```

greetCustomer($objCust);
?>

```

Запустите в браузере файл с именем `testCustomer.php`, сперва установив значение переменной `$promotionCurrentlyRunning` равное `TRUE`, а затем `FALSE`. В первом случае появится сообщение о выигрыше приза.

Интерфейсы

Иногда в приложениях встречаются классы, которые не связаны отношением наследования. В общем случае это могут быть абсолютно различные классы, однако в тоже время обладающие некоторыми общими характеристиками. Например, сосуд и дверь можно открывать и закрывать. Но на этом сходства и заканчиваются. Не важно, какой природы сосуд или дверь. Важно лишь то, что они могут совершать одинаковые действия. И все.

Что позволяют делать интерфейсы

Интерфейс (`interface`) позволяет указать, что объект может совершать определенные действия. Однако при этом он не указывает, каким именно образом. Интерфейс является своего рода связующим звеном между некоторыми (в общем случае несвязанными друг с другом) объектами и общими для них действиями. Объект, реализующий интерфейс, гарантирует, что его пользователи смогут выполнять определенный набор действий. Так, велосипед и футбольная экипировка представляют собой абсолютно разные вещи. Однако соответствующие объекты системы реестра товаров спортивного магазина должны уметь одинаково взаимодействовать с ней.

Объявляя интерфейс и реализуя его в объектах, можно свести абсолютно разные классы к общим функциям. В следующем примере показана аналогия “сосуд-дверь”.

Создайте файл с именем `interface.Opener.php` и следующими строками кода.

```

<?php
interface Openable {
    abstract function open();
    abstract function close();
}
?>

```

Поскольку мы условились называть файлы, содержащие описание классов, именами `class.имя_класса.php`, это правило следует применять и для интерфейсов: `interface.имя_интерфейса.php`.

Синтаксис для объявления интерфейсов подобен тому, который используется для классов. Однако вместо ключевого слова `class` используется `interface`. Кроме того, в интерфейсах отсутствуют переменные-члены, а методы не реализуются.

По этой причине функции в интерфейсах объявляются абстрактными. Для этого предназначено ключевое слово `abstract`, Т.е. любой класс, реализующий данный интерфейс, должен обеспечить реализацию *всех* его абстрактных методов. В противном случае интерпретатор PHP выдаст ошибку.

Как работают интерфейсы

Интерфейс `Openable` является связующим звеном между разными частями приложения. Он указывает, что любой класс, реализующий этот интерфейс, будет предоставлять два метода с именами `open()` и `close()` (без аргументов). После определения

набора общих методов самые разные объекты приложения могут использовать одни и те же функции. При этом наличие между ними отношения наследования является не обязательным.

Создадим два файла с именами `class.Door.php` и `class.Jar.php`.

Файл `class.Door.php`.

```
<?php

require_once('interface.Openable.php');

class Door implements Openable {

    private $_locked = false;

    public function open() {
        if($this->_locked) {
            print "Невозможно открыть дверь. Она закрыта.";
        } else {
            print "скрип при открытии...<br>";
        }
    }

    public function close() {
        print "Хлоп!!<br>";
    }

    public function lockDoor() {
        $this->_locked = true;
    }

    public function unlockDoor() {
        $this->_locked = false;
    }
}

?>
```

Файл `class.Jar.php`.

```
<?
require_once('interface.Openable.php');

class Jar implements Openable {
    private $contents;

    public function __construct($contents) {
        $this->contents = $contents;
    }

    public function open() {
        print "Теперь сосуд открыт<br>";
    }

    public function close() {
        print "Теперь сосуд закрыт<br>";
    }
?

?>
```

Для того чтобы продемонстрировать использование этих двух классов, создадим файл с именем `testOpenable.php` в том же каталоге.

```
<?php
require_once('class.Door.php');
require_once('class.Jar.php');

function openSomething(Openable $obj) {
    $obj->open();
}

$objDoor = new Door();
$objJar = new Jar("желе");

openSomething($objDoor);
openSomething($objJar);
?>
```

Поскольку оба класса `Door` и `Jar` реализуют интерфейс `Openable`, объекты этих классов можно передавать в качестве аргументов функции `openSomething()`. Так как она принимает в качестве параметров объекты, реализующие интерфейс `Openable`, методы `open()` и `close()` могут быть вызваны в ее теле без проблем. При этом доступ, например, к атрибуту `$contents` класса `Jar` или методам `lock()` или `unlock()` класса `Door` при описании функции `openSomething()` запрещен. Действительно, все эти атрибуты и методы не являются частью интерфейса `Openable`. В нем содержатся только объявления методов `open()` и `close()` и больше ничего.

Использование интерфейсов при разработке приложений позволяет связать абсолютно разные объекты посредством функций, специфицированных в интерфейсе.

Инкапсуляция

Как уже упоминалось ранее в данной главе, объекты позволяют скрывать детали реализации от пользователей. Им нет необходимости знать, где хранится информация, например, о классе `Volunteer`: в базе данных, в файле, документе XML или используется какой-либо другой механизм при вызове функции `signUp()`. Точно также, пользователям не важно, как информация об участниках хранится внутри объекта: с использованием переменных, массивов или других объектов. Эта возможность объектов скрывать детали своей реализации называется *инкапсуляцией* (*encapsulation*). В общем случае инкапсуляция воплощает в себе две концепции: защита внутренних данных класса от доступа извне и скрытие деталей реализации.

Буквально слово *инкапсулировать* означает поместить в капсулу или внешний контейнер. Хорошо спроектированный класс обеспечивает внешнюю оболочку, надстроенную над внутренней структурой класса. При этом для доступа к классу извне он предоставляет специальный интерфейс. Такой подход для создания классов обладает двумя преимуществами: можно легко вносить изменения во внутреннюю структуру класса без каких-либо последствий для его пользователей; поскольку извне класса невозможно изменить состояние или значения атрибутов объекта, создатель класса может быть уверен в их корректности и использовать без каких-либо проблем.

Переменные-члены и методы класса имеют соответствующий уровень видимости (доступа). Под *видимостью* (*visibility*) понимают возможность доступа к классу извне. *Закрытые* (*private*) члены класса не доступны извне и используются для реализации его внутренней структуры. *Защищенные* (*protected*) переменные-члены и методы класса доступны только классам-наследникам данного класса. И наконец, *открытые* (*public*) члены класса доступны из внешней части приложения.

В общем случае все внутренние переменные-члены класса должны быть объявлены как `private`. Доступ к ним следует осуществлять посредством специальных методов доступа (accessor method). Незачем кому-либо позволять кормить себя вслепую. Прежде чем что-либо съесть, следует это проверить. Точно также и с объектами — если кто-то извне хочет изменить состояние объекта или значения атрибутов, использование инкапсулированного доступа к ним посредством открытых функций позволяет проверить эти изменения на корректность и впоследствии принять либо же отклонить их.

В качестве примера рассмотрим приложение, предназначенное для обработки информации о состоянии счета клиентов банка. Для этого необходимо создать класс `Account` (счет) с атрибутом `totalBalance` (общий баланс) и методами `makeDeposit()` (сделать вклад) и `makeWithdrawal()` (снять деньги). Очевидно при этом, что значение переменной `$totalBalance` можно только считывать. Единственный возможный способ для ее изменения — сделать вклад или снять деньги. Если же этот атрибут объявить как открытый (т.е. `public`), тогда извне можно было бы легко увеличивать это значение без реального вклада денег. Очевидно, что такой подход абсолютно не устроит банк. Вместо этого следует объявить атрибут `totalBalance` как `private`, а доступ к нему осуществить посредством открытого метода `getTotalBalance()`. Именно он будет возвращать остаток на счету клиента. В этом случае прямого доступа к переменной `$totalBalance` уже не будет. И если клиенту банка захочется увеличить сумму на своем счету, ему придется сделать соответствующий вклад.

Возможность ООП скрывать детали реализации класса и защищать доступ к его внутренним данным позволяет придать разрабатываемому приложению гибкость и устойчивость.

Инкапсуляция позволяет защищать и контролировать доступ к внутренним данным объектно-ориентированных систем и скрывать детали реализации.

Изменения ООП в PHP 5

Поддержка объектов в PHP была включена начиная еще с версии PHP3. Однако тогда у разработчиков PHP не было намерений реализовывать идею классов и объектов в полном виде. По словам Зива Сураски (Zeev Suraski), одного из основных разработчиков ядра Zend, тогда поддержка объектов представляла собой “синтаксический сахар” для ассоциативных массивов. Объекты были удобны для группировки данных и функций. При этом была реализована лишь небольшая группа свойств, традиционно связанных с объектно-ориентированными языками. Однако с ростом популярности PHP разработка масштабных распределенных приложений не могла обходиться без использования ООП, но плохая реализация в PHP принципов ООП накладывала свои ограничения.

Например, абсолютно отсутствовала поддержка “настоящей” инкапсуляции. Переменные-члены и методы класса нельзя было объявить как закрытые или защищенные. Все они были открытые, что в большинстве случаев создает громадные проблемы.

Кроме того, отсутствовала поддержка абстрактных интерфейсов и методов. Переменные-члены и методы нельзя было объявлять как статические. Отсутствовали деструкторы. Всем, кто когда-либо имел опыт работы с объектно-ориентированными языками, эти понятия знакомы. Отсутствие большинства этих свойств в объектной модели PHP делало переход от одного языка, скажем Java (который, к слову, поддерживает все эти свойства), очень непростым. Для тех, кто имел опыт работы с PHP 4, приводим перечень изменений, внесенных в объектную модель PHP 5.

Новое свойство	Преимущество
Поддержка закрытых и защищенных переменных-членов и методов классов	В PHP стали возможны инкапсуляция и защита данных
Улучшенная поддержка разыменовывания ссылок	Стала возможной поддержка выражений вида <code>\$obj->getSomething()->doSomething()</code>
Поддержка статических методов и переменных-членов класса	Теперь можно явно объявлять статические методы. Использование статических переменных позволяет контролировать пространство глобальных имен
Наличие универсального конструктора	Конструктор любого класса имеет имя <code>__construct()</code> . Такой подход позволяет облегчить инкапсуляцию перегруженных конструкторов подклассов, а также вносить изменения, когда несколько классов наследуют друг друга
Наличие деструкторов	Метод класса с именем <code>__destruct()</code> называется деструктором. Его следует вызывать, когда необходимо выполнить некоторые действия перед удалением объекта
Поддержка абстрактных классов и интерфейсов	Необходимые методы можно объявить в родительском классе, а реализовать — в другом (подклассе). Абстрактные классы не могут быть инстанцированы. Инстанцировать можно только их неабстрактные подклассы
Задание типов параметров функций	В качестве аргумента функции можно объявлять класс. В этом случае при вызове функции необходимо передавать объект соответствующего класса. Выражение <code>function foo(Bar \$objBar)</code> позволяет однозначно определить тип данных аргумента функции

Резюме

В данной главе вы познакомились с концепцией объектно-ориентированного программирования. В ней класс рассматривается как каркас для создания объектов. В свою очередь объекты являются экземплярами классов. Объекты обладают свойствами, называемыми *атрибутами* (*property*), и поведением, определяемым *методами* (*methods*). Атрибуты можно рассматривать в терминах переменных, а методы — в терминах функций.

Некоторые классы имеют “родственные” отношения. Например, квадрат является прямоугольником. Если один класс является подклассом другого, он наследует все его методы и атрибуты. При этом есть возможность модифицировать наследуемые методы. Можно реализовать метод заново либо же воспользоваться реализацией, предоставляемой родительским классом, и добавить функциональность, характерную только для этого конкретного подкласса (или вообще не перегружать метод).

Инкапсуляция представляет собой важную концепцию ООП. Она позволяет закрыть доступ к внутренним переменным-членам класса и скрывать детали его реализации от внешних пользователей. Методы и переменные-члены класса имеют уровни видимости (доступа): закрытый, защищенный и открытый. Доступ к закрытым членам класса можно осуществлять только из самого класса. Защищенные члены доступны только классу и его наследникам. Открытые члены класса доступны извне.

Поддержка ООП в PHP появилась в версии PHP 5 с разработкой нового ядра Zend Engine 2. Новые свойства и значительные улучшения позволяют отнести PHP к классу объектно-ориентированных языков программирования.

2

Унифицированный язык моделирования UML

Если вы являетесь единственным разработчиком небольшого приложения, то при его разработке достаточно нарисовать схему на бумаге и хранить все детали реализации в уме. В таком случае разработчик сможет сам успешно выполнить проект.

Предположим, однако, что над разработкой приложения работают два человека. Например, один из них отвечает за проектирование системы, а второй — за написание кода. Тогда возникает следующий вопрос: каким образом следует передавать информацию о проекте друг другу и заинтересованным лицам? Безусловно, одним из вариантов является словесное описание классов, особенностей реализации приложения и принципов его функционирования. Однако уже через небольшой промежуток времени придется смириться с громадным количеством текста. А если представить себе еще больший проект, то становится очевидным, что способ описания деталей его реализации простыми словами нельзя назвать эффективным и практичным. В то же время можно было бы разработать свои схемы и диаграммы, упрощающие этот процесс. Однако в этом случае придется потратить много времени на объяснение своих собственных обозначений другим членам команды разработчиков.

Существует корректное решение данной проблемы — унифицированный язык моделирования UML (Unified Modeling Language). Прежде всего, UML — это описание стандартизованных диаграмм, каждая из которых используется на определенном этапе разработки программного обеспечения. При этом язык UML представляет единый способ описания свойств системы и мощный инструментарий для визуализации процесса разработки.

Определение требований

В настоящей главе в качестве примера будет рассматриваться процесс разработки гипотетической системы BandSpy, предназначеннной для управления информацией о музыкальных группах. В рамках этого процесса будут рассмотрены основные диаграммы UML и будет описано, как их следует использовать.

Предположим, с вами связался клиент из музыкальной записывающей компании на предмет разработки Web-приложения, предназначенного для учета информации о музыкальных группах, которые он представляет. Во время первого телефонного разговора клиент пояснил, что система BandSpy должна позволять пользователям просматривать информацию о группах через Web, а также предоставлять информацию о предстоящих концертах. Вам удалось узнать, что некоторые сотрудники компании должны иметь возможность добавлять в систему информацию о новых группах и заказывать концерты. Несмотря на то, что ваш разговор с клиентом является неформальным, он инициирует важную фазу проектирования программного обеспечения — *определение требований* (requirements-gathering phase). Во время данной фазы необходимо определить, чего именно хотят пользователи от разрабатываемой системы.

Интервьюирование клиента

Предположим, во время телефонного разговора вы договорились о встрече с клиентом. После знакомства во время встречи со специалистом в области разработки программных систем обсуждаются вопросы, связанные с системой BandSpy. На встрече присутствуют Билл, владелец компании, Джейн, менеджер, и Том, “компьютерщик” компании.

На этапе интервью очень важно прочувствовать роли людей. Очевидно, что каждый из специалистов музыкальной компании имеет свое представление о том, как должна работать система. Хотя в общем случае это видение не всегда может быть определяющим для конечной версии программного продукта.

Во время фазы определения требований важно выделить человека, который имеет глубокие знания о *предметной области* (domain), т.е. области, которую будет моделировать разрабатываемое приложение. Это может быть человек, который выполнял работу вручную, а теперь она будет автоматизирована. Или же это может быть человек, хорошо знающий детали производства и функции создаваемого программного обеспечения. В любом случае данного человека принято называть *экспертом по предметной области* (domain expert).

Во время интервью больше всего говорит Джейн. Она описывает, что должна будет выполнять система и как с этим справляется компания сейчас. Нетрудно предположить, что в нашем случае именно Джейн является экспертом по предметной области. Поэтому в последующих интервью будет полезным общаться с ней напрямую. Билл большую часть времени молчит. У него, возможно, и других дел хватает. А Том отметил, что наверняка будет отвечать за информационное наполнение системы. Поэтому ему придется объяснить работу средств администрирования.

В результате проведенного интервью можно выписать следующий перечень требований, которым должна удовлетворять система.

1. Пользователи должны иметь возможность посещать Web-узел BandSpy и загружать информацию о музыкальных группах. К ней относится информация о типе группы, музыкантах, играющих в ней, и музыкальных инструментах.
2. Пользователи должны иметь возможность просматривать информацию о предстоящих концертах, в которых могут принимать участие одна и несколько групп.
3. Администратор Web-узла должен иметь возможность добавлять информацию о новых группах.

4. Администратор Web-узла должен иметь возможность редактировать информацию о существующих группах.
5. Администратор Web-узла должен иметь возможность добавлять информацию о предстоящих концертах. Этот процесс включает резервирование места проведения концерта, а также формирование и печать билетов. Фактически этим будут заниматься сторонние компании. Предполагается, что они имеют в наличии соответствующие системы резервирования, с которыми должна взаимодействовать система BandSpy.

Итак, определив требования, которым должна удовлетворять система BandSpy, можно перейти к диаграммам UML.

Диаграммы прецедентов

Диаграммы прецедентов (use case diagram) описывают систему с точки зрения выполнения пользователем определенных действий (задач), т.е. каждый прецедент отражает действие, которое пользователь выполняет в системе. На рис. 2.1 показана диаграмма прецедентов для пользователей, посещающих Web-узел BandSpy и просматривающих информацию о музыкальных группах.

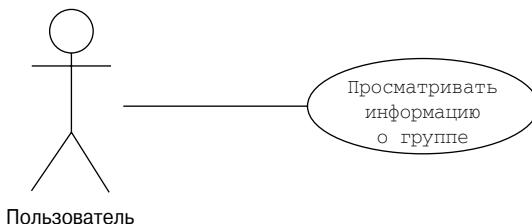


Рис. 2.1.

Исполнитель (actor) изображается в виде фигурки человека, а прецедент — эллипса. Линия указывает, что исполнитель может выполнять определенный прецедент. На диаграммах прецедентов в качестве исполнителя может выступать как человек, так и внешняя система.

Заметим, что прецедент Просматривать информацию о группе является достаточно общим. В данном случае высокий уровень детализации не желателен. Необходимо просто описать все прецеденты системы. При этом их можно разбивать на отдельные *сценарии* (scenarios). Сценарий представляет собой последовательность шагов, составляющих прецедент. Например, предыдущий прецедент Просматривать информацию о группе можно разбить на следующие сценарии:

- Пользователь заходит на Web-узел BandSpy.
- Пользователь просматривает содержимое Web-узла, используя меню.
- Пользователь просматривает информацию о группе/музыкантах/концертах.

В свою очередь диаграмма прецедентов для администратора имеет вид, показанный на рис. 2.2.

Диаграмма прецедентов для администратора состоит из набора действий, которые он может выполнять при работе с системой BandSpy. При этом клиент упоминал

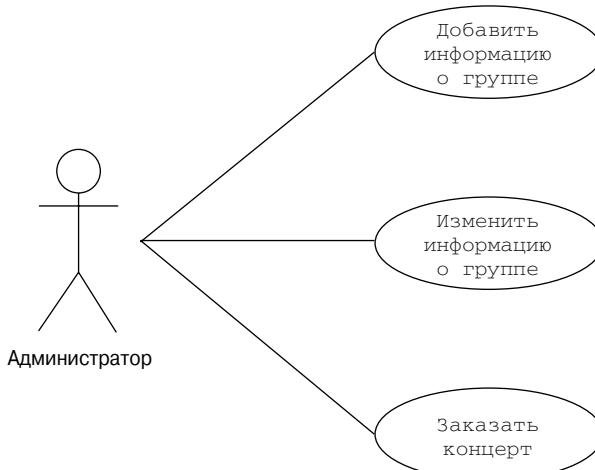


Рис. 2.2.

о необходимости наличия секции администратора, для входа в которую нужна соответствующая учетная запись с паролем. Поскольку вход в систему можно рассматривать как выполнение задачи, с помощью ключевого слова “включает” можно указать, что этот прецедент является частью трех предыдущих. На рис. 2.3 прецедент Вход в систему соединяется с тремя другими посредством штриховой линий.

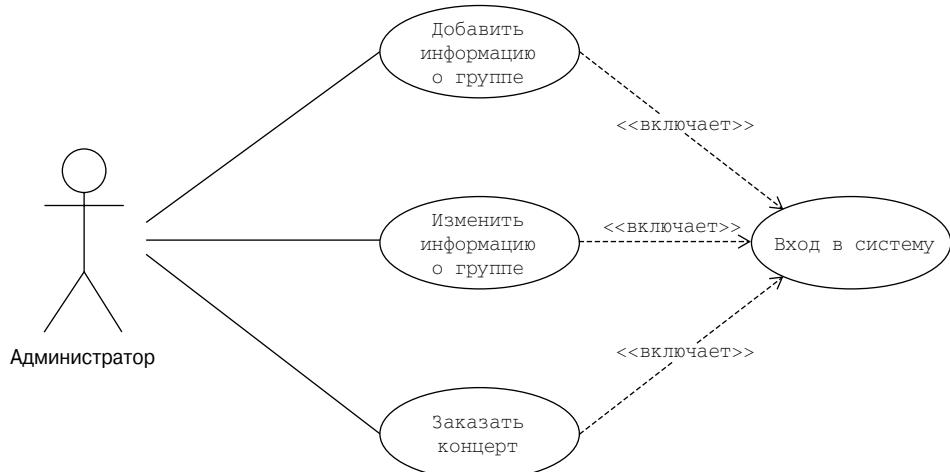


Рис. 2.3.

Очевидно, что в общем случае диаграмма прецедентов может включать несколько исполнителей. При этом разные исполнители могут выполнять один и тот же прецедент. На рис. 2.4 приведена полная диаграмма прецедентов для системы BandSpy. Поскольку администратор и обычный пользователь могут просматривать информацию о музыкальных группах, они могут выполнять один тот же прецедент. Кроме того, на

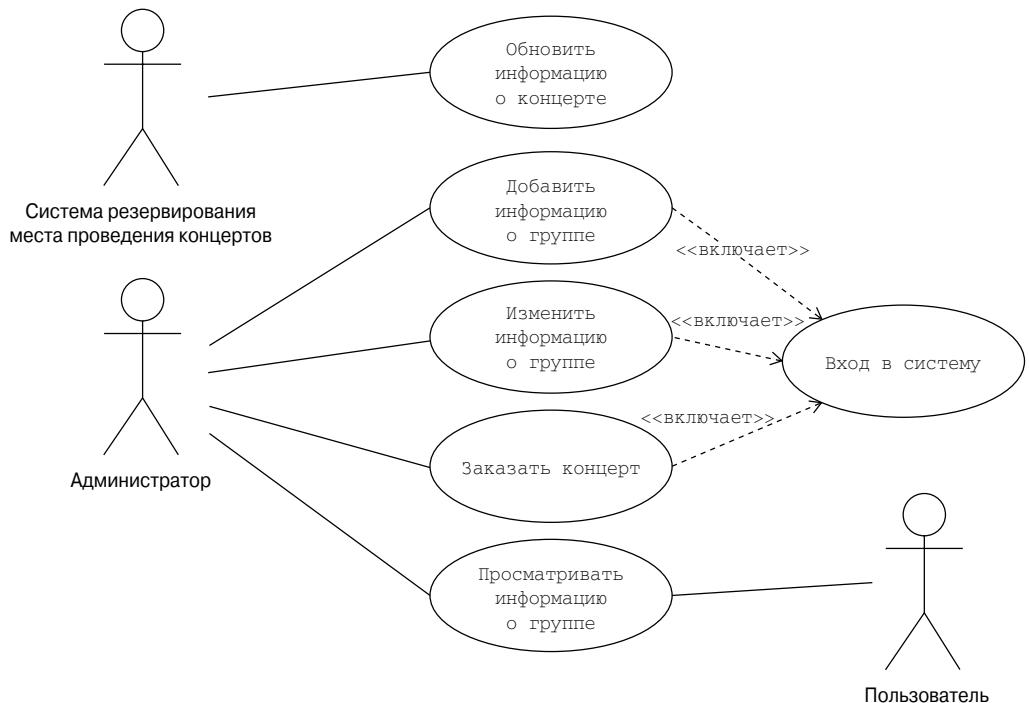


Рис. 2.4.

этой диаграмме приведен исполнитель, который никак не связан с человеком: система резервирования места проведения концертов. Она выполняет функции по обновлению информации о предстоящих концертах групп.

Диаграммы классов

Несмотря на то, что диаграммы прецедентов являются удобным средством для описания возможностей системы по отношению к разным пользователям (ролям), на них никак не отражены детали функционирования самой системы. Для отражения подробностей работы системы следует воспользоваться знаниями об объектах и соответственно другими диаграммами UML — диаграммами классов.

Моделирование предметной области

После интервьюирования клиента и определения прецедентов системы можно переходить к следующему этапу проектирования программного обеспечения — *моделированию предметной области* (domain modeling). При этом стоит отметить, что разрабатываемые классы должны соотноситься с предметами реального мира. Например, в нашем приложении BandSpy пользователи могут просматривать информацию о музыкантах. Соответственно, неплохо бы было иметь класс *Musician* или еще лучше интерфейс *Musician*, который могут реализовывать различные типы музыкантов.

Диаграммы классов, по-видимому, представляют собой один из важнейших видов диаграмм для описания и понимания принципов работы разрабатываемого программного обеспечения. При этом они позволяют обеспечить хороший уровень детализации и гибкость. Действительно, диаграммы классов предоставляют описание моделируемой предметной области. И в то же время они позволяют варьировать уровень детализации описания работы приложения, например, от самого абстрактного до определения конкретных методов и атрибутов классов. Кроме того, диаграммы классов являются очень полезными при описании таких объектно-ориентированных понятий, как шаблоны проектирования, которым посвящена глава 4.

Основной элемент диаграммы классов (прямоугольник с именем класса, методами и атрибутами) изображен на рис. 2.5. В верхней секции прямоугольника указывается имя класса. В свою очередь в средней секции размещены *атрибуты* (attribute) класса, а в нижней — *операции* (operation). В языке PHP термины “атрибуты” и “операции” можно рассматривать как переменные-члены и методы класса, соответственно.

Как видно из приведенного рисунка, типы атрибутов стоят после символа двоеточия (:). В нашем простом примере все атрибуты имеют строковый тип. Знак “минус” (-) указывает, что атрибуты являются закрытыми членами класса. В свою очередь знак “плюс” (+), стоящий перед операциями, показывает, что все они являются открытыми. В случае, если операция возвращает некоторое значение, его тип указывается после символа двоеточия. В приведенном классе все они также являются строковыми.

Ниже приведен фрагмент кода на PHP, соответствующий классу Musician, диаграмма которого приведена на рис. 2.5.

```
class Musician {
    private $last;
    private $first;
    private $bandName;
    private $type;

    function __construct($last, $first, $musicianType) {
        $this->last = $last;
        $this->first = $first;
        $this->mtype = $musicianType;
    }

    public function getName() {
        echo $this->first . $this->last;
    }

    public function getBand() {
        echo $this->bandName;
    }

    public function getMusicianType() {
        echo $this->type;
    }

    public function setName($first, $last) {
        $this->first = $first;
        $this->last = $last;
    }
}
```

Musician
-type:String
-firstName:String
-lastName:String
-bandName:String
+getBand():String
+getName():String
+getType():String
+setBand()
+setType()
+setName()

Рис. 2.5.

```

    }

    public function setBand($bandName) {
        $this->bandName = $bandName;
    }

    public function setMusicianType($musicianType) {
        $this->type = $musicianType;
    }
}

```

Отношения между классами

Если бы разрабатываемое приложение содержало один класс, то очевидно, что от диаграмм классов не было бы никакой пользы. Поскольку рассматриваемое в данной главе программное обеспечение BandSpy содержит несколько классов, необходимо определить, как соответствующие объекты будут взаимодействовать друг с другом. В предыдущем примере все переменные-члены класса Musician были строковыми. В общем случае так поступать не стоит. Действительно, если музыкант является участником некоторой музыкальной группы, то имеет смысл выводить информацию не только о ее имени, но и об остальных участниках коллектива, об используемом ими стиле музыки и т.д. Все эти вопросы должны рассматриваться на этапе проектирования приложения. Действительно, в таком случае некоторые переменные-члены класса будут представлять собой более сложные структуры данных. И, соответственно, для них необходимо создавать дополнительные классы.

Ассоциации

Как видно из рис. 2.6, атрибут с именем bandName (имя группы) был перемещен в другой класс с именем Band. При этом на диаграмме не приводятся абсолютно все атрибуты и методы классов.

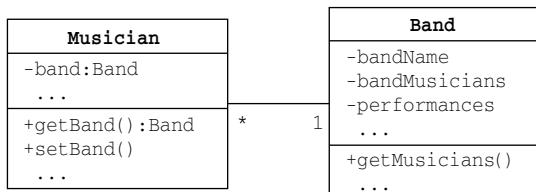


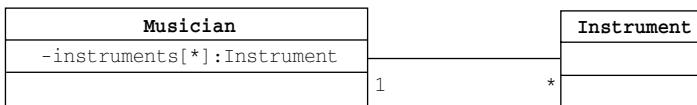
Рис. 2.6.

При использовании диаграмм классов следует приводить только наиболее важные для данной диаграммы элементы. Поскольку для доступа к закрытым переменным-членам класса по умолчанию подразумевается использование простых методов, здесь они опущены. Если диаграмма классов является неполной, можно воспользоваться символом троеточия (...) с тем, чтобы показать, что некоторая информация была специально опущена.

Прямая линия, соединяющая два прямоугольника, показывает структурную связь между классами, называемую *ассоциацией* (association). В данном случае ассоциация имеет кратность *один ко многим*, что определяется наличием символов `1` и `*`, т.е. один объект типа **Band** (музыкальная группа) может содержать несколько объектов типа

Musician (музыкант), и наоборот — несколько музыкантов могут участвовать в одной музыкальной группе.

Кроме того, ассоциация может иметь *направление* (navigability). В рассмотренном на-
ми примере предполагается, что ассоциация *дву направленна* (bi-directional navigability)
(линия без стрелки). Это значит, что можно перемещаться от одного класса к другому
и наоборот. Иначе говоря, оба класса знают друг о друге. В свою очередь, если ассоциа-
ция является *одно направленной* (unidirectional navigability), перемещаться можно толь-
ко в одном направлении. На рис. 2.7 приведен пример такой связи между классами
Musician (музыкант) и *Instrument* (музыкальный инструмент). Очевидно, что объект
типа *Musician* имеет доступ ко всем своим музыкальным инструментам. В свою оче-
редь объекту класса *Instrument* не известно, какой именно музыкант владеет им.

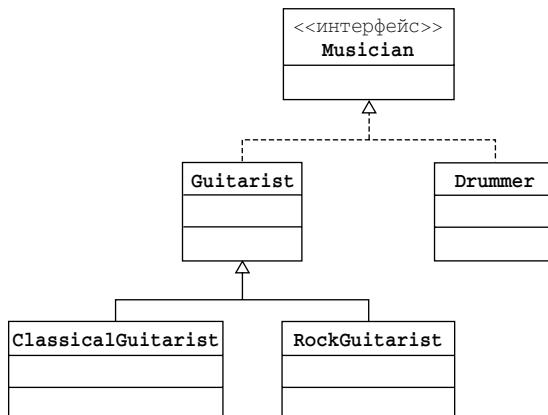


Puc, 2,7,

Реализация и обобщение

Ранее уже упоминалось, что создание интерфейса `Musician`, который будут реализовывать другие классы (соответствующие разным типам музыкантов), является достаточно эффективным подходом. Благодаря ему можно разрабатывать такие классы, как `Guitarist` (гитарист) или `Drummer` (барабанщик), поведение которых будет подчиняться правилам, определенным для интерфейса `Musician`.

Если необходимо указать интерфейс и класс, который его реализует, используется отношение *реализации* (realization). Для соединения соответствующих прямоугольников используется штриховая линия с полым треугольником на конце. Треугольник указывает на интерфейс. При этом в UML-диagramмах интерфейсы обозначаются точно так же, как и классы, однако дополнительно используется слово <<интерфейс>> над именем интерфейса. Пример отношения реализации приведен на рис. 2.8 (классы *Guitarist* и *Drummer*, реализуют интерфейс *Musician*).



Puc. 2.8.

Если необходимо указать, что один класс наследует другой, более общий, используется отношение *общества* (generalization). Его отличие от отношения реализации заключается в использовании сплошной линии. На диаграмме классов, приведенной на рис. 2.8, классы RockGuitarist (рок-гитарист) и ClassicalGuitarist (классический гитарист) являются подклассами класса Guitarist.

Композиция

Очень часто отношения между классами основаны не на ассоциации или наследовании, а на том, как они группируются. Рассмотрим пример с набором барабанов. В нашем приложении на основе интерфейса *Instrument* (музыкальный инструмент) реализованы такие классы, как *Guitar* (гитара) или *Piano* (пианино), что показано на рис. 2.9. Кроме того, этот интерфейс также реализуют классы *DrumSet* (набор барабанов), *Drum* (барабан) и *Cymbal* (тарелки). При этом на приведенной диаграмме введено новое обозначение — сплошная линия с заштрихованным ромбом на конце. Она показывает, что класс-целое *DrumSet* агрегирует (содержит) классы-части *Drum* и *Cymbal*, т.е. любой экземпляр класса *DrumSet* будет также содержать экземпляры классов *Drum* и *Cymbal*. В общем случае отношение композиции характеризуется следующим. Во-первых, части композиции не могут использоваться разными объектами. Другими словами, объект класса *Drum*, принадлежащий некоторому объекту *DrumSet*, не может быть использован в другом объекте. Во-вторых, при удалении объекта класса *DrumSet* необходимо удалить и все его части. Более общий вид композиции называется *агрегацией* (aggregate). Для агрегации используется то же обозначение, что и для композиции, только ромб является незаштрихованным. Отношение агрегации позволяет разным экземплярам класса-целого использовать одни и те же экземпляры классов-частей. Таким образом, при удалении объектов не обязательно удалять его части.

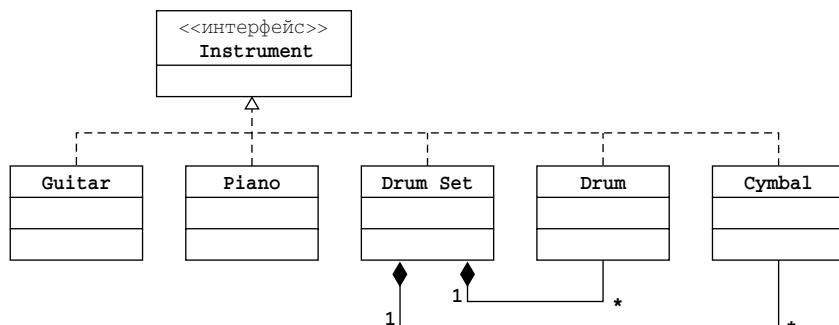


Рис. 2.9.

Реализация

Описав основные принципы использования диаграмм классов, приведем пример фрагмента кода, который реализует концепции, изложенные в предыдущих разделах.

```

<?php

interface Band {
    public function getName();
    public function getGenre();
}
  
```

```
public function addMusician();
public function getMusicians();
}

interface Musician {
    public function addInstrument();
    public function getInstruments();

    public function assignToBand();
    public function getMusicianType();
}

interface Instrument {
    public function getName();
    public function getCategory();
}

class Guitarist implements Musician {

    private $last;
    private $first;
    private $musicianType;

    private $instruments;
    private $bandReference;

    function __construct($first, $last) {
        $this->last = $last;
        $this->first = $first;
        $this->instruments = array();
        $this->musicianType = "гитарист";
    }

    public function getName() {
        return $this->first . " " . $this->last;
    }

    public function addInstrument(Instrument $instrument) {
        array_push($this->instruments, $instrument);
    }

    public function getInstruments() {
        return $this->instruments;
    }

    public function getBand() {
        return $this->$bandReference;
    }

    public function assignToBand(Band $band) {
        $this->$bandReference = $band;
    }

    public function getMusicianType() {
        return $this->musicianType;
    }

    public function setMusicianType($musicianType) {
        $this->musicianType = $musicianType;
    }
}
```

```
class LeadGuitarist extends Guitarist {
    function __construct($last, $first) {
        parent::__construct($last, $first);
        $this->setMusicianType("ведущий гитарист");
    }
}

class RockBand implements Band {

    private $bandName;
    private $bandGenre;
    private $musicians;

    function __construct($bandName) {
        $this->bandName = $bandName;
        $this->musicians = array();
        $this->bandGenre = "рок";
    }

    public function getName() {
        return $this->bandName;
    }

    public function getGenre(){
        return $this->bandGenre;
    }

    public function addMusician(Musician $musician){
        array_push($this->musicians, $musician);
        $musician->assignToBand($this);
    }

    public function getMusicians() {
        return $this->musicians;
    }
}

class Guitar implements Instrument {

    private $name;
    private $category;

    function __construct($name) {
        $this->name = $name;
        $this->category = "гитара";
    }

    public function getName() {
        return $this->name;
    }

    public function getCategory() {
        return $this->category;
    }
}

// Создание тестовых объектов
$band = new RockBand("Изменчивые");
$bandMemberA = new Guitarist("Джек", "Флот");
$bandMemberB = new LeadGuitarist("Джим", "Интелджер");
$bandMemberA->addInstrument(new Guitar("Гибсон Ле Поль"));
$bandMemberA->addInstrument(new Drum("Драм-кит"));
```

```

$bandMemberB->addInstrument(new Guitar("Фендер Стратокастер"));
$bandMemberB->addInstrument(new Guitar("Хондо Н-77"));

$band->addMusician($bandMemberA);
$band->addMusician($bandMemberB);

foreach($band->getMusicians() as $musician) {
    echo "Музыкант ".$musician->getName() . "<br>";
    echo "является " . $musician->getMusicianType() . "<br>";
    echo "в " . $musician->getBand()->getGenre() .
        " и играет в музыкальной группе <br>";
    echo "с названием " . $musician->getBand()->getName() . "<br>";

    foreach($musician->getInstruments() as $instrument) {
        echo "И играет на музыкальном инструменте " . $instrument->getName() .
    " ";
        echo $instrument->getCategory() . "<br>";
    }
    echo "<p>";
}
?>

```

Несмотря на то, что приведенный пример является достаточно простым, он иллюстрирует один из важнейших аспектов объектно-ориентированного программирования. Действительно, при создании этих объектов отсутствуют какие-либо условные переходы, т.е. нет необходимости проверять, какой именно объект класса, Musician или Instrument, используется при вызове определенных методов. Поскольку соблюдены все правила реализации интерфейсов, можно быть полностью уверенным, что соответствующие объекты будут вести себя корректно согласно их реализации. Как уже отмечалось в предыдущей главе, это свойство ООП называется полиморфизмом. Если необходимо добавить новый тип музыканта, можно просто создать новый класс, определяя его поведение согласно интерфейсу Musician.

Диаграммы видов деятельности

Ранее Джейн упоминала, что при добавлении информации о новом концерте система BandSpy должна уметь взаимодействовать с другими внешними системами — системой формирования и печати билетов, а также системой резервирования места проведения концертов. Для того чтобы понять порядок взаимодействия всех составляющих данного процесса, можно воспользоваться *диаграммами видов деятельности* (activity diagram). Данный тип диаграмм является особенно полезным, когда необходимо отобразить порядок видов деятельности в рамках прецедента. В качестве примера рассмотрим прецедент добавления администратором информации о новом концерте в систему BandSpy.

Последующие обсуждения с Джейн позволили узнать, что приложение BandSpy должно не только отправлять сообщения системе резервирования места проведения концертов, но также должно получать информацию от обеих внешних систем. Так, система резервирования будет отправлять сообщения с указанием возможности проведения концерта в том или ином месте. В свою очередь система формирования и печати билетов будет предоставлять BandSpy информацию о билетах, например о ценах на них. Таким образом, если администратор попытается забронировать концерт на дату, которая недоступна или занята, соответствующее сообщение будет получено от системы резервирования. В этом случае администратору придется выбрать другое время для проведения концерта.

Диаграмма видов деятельности для описанного выше процесса изображена на рис. 2.10. Она начинается с черного круга в верхней части, который означает *начальное состояние* (starting point). Далее поток событий следует согласно изображенными линиям со стрелками, которые называются *переходами* (transition), соединяющими эллизы, изображающие *действия* (activities).

Точка ветвления (decision point) на диаграммах видов деятельности обозначается ромбом. Это то место, начиная с которого поток событий разветвляется согласно некоторому решению или условию. В рассматриваемом примере решение зависит от доступности места проведения концерта на указанную дату.

После принятия решения следует жирная черная линия, которая называется *полосой синхронизации* (fork). Она используется для распараллеливания потока событий на два независимых потока. Так, система резервирования места проведения концертов генерирует подтверждающее сообщение и отправит его системе BandSpy. В то же время система формирования и печати билетов начинает работу по производству билетов на концерт и также отправляет соответствующую информацию приложению BandSpy. Впоследствии оба потока (от внешних систем) встречаются снова на жирной черной линии, в данном случае называемой *объединением* (join). Объединение показывает, что оба внешних сообщения должны быть направлены системе BandSpy синхронно, чтобы выполнить последнее действие: сохранить и обновить информацию о новом концерте.

Необходимо заметить, что три масштабных прямоугольника, которые используются для разделения видов деятельности каждой системы, называются *дорожками* (swimlane). Несмотря на то, что они не являются обязательными, они позволяют сделать диаграмму более понятной, особенно в случае использования большого количества систем.

Диаграммы последовательностей

Нетрудно видеть, что приложение BandSpy содержит иерархию классов. Например, класс *Band* (группа) состоит из музыкантов (класс *Musician*). Те, в свою очередь, содержат класс *Instrument* (инструмент). Кроме того, инструменты могут включать другие музыкальные инструменты. Во время использования приложения различные типы объектов обмениваются разными сообщениями, т.е. вызов метода можно рассматривать как обмен сообщениями между объектами. Например, если объекту типа *Band* необходимо узнать, на каких музыкальных инструментах (*Instrument*) играют участники данной группы (*Musician*), он отправит сообщение с именем *getInstruments()* всем музыкантам. Другими словами, объект типа *Band* вызывает метод с именем *getInstruments()* для всех объектов класса *Musician*.

Удобно визуализировать процесс отправки сообщений между объектами. Для этого в UML предназначена специальная диаграмма — *диаграмма последовательностей* (sequence diagram).

Диаграмму последовательностей можно использовать для конкретного прецедента. Так, в предыдущих подразделах рассматривался прецедент Просматривать информацию о группе. Чтобы построить диаграмму последовательностей для данного прецедента, разобьем его на сценарии. На рис. 2.11 показан прецедент Просматривать информацию о группе для исполнителя Пользователь.

Рассмотрим прецедент Просмотр музыкальных инструментов по группам и построим для него соответствующую диаграмму последовательностей (рис. 2.12).

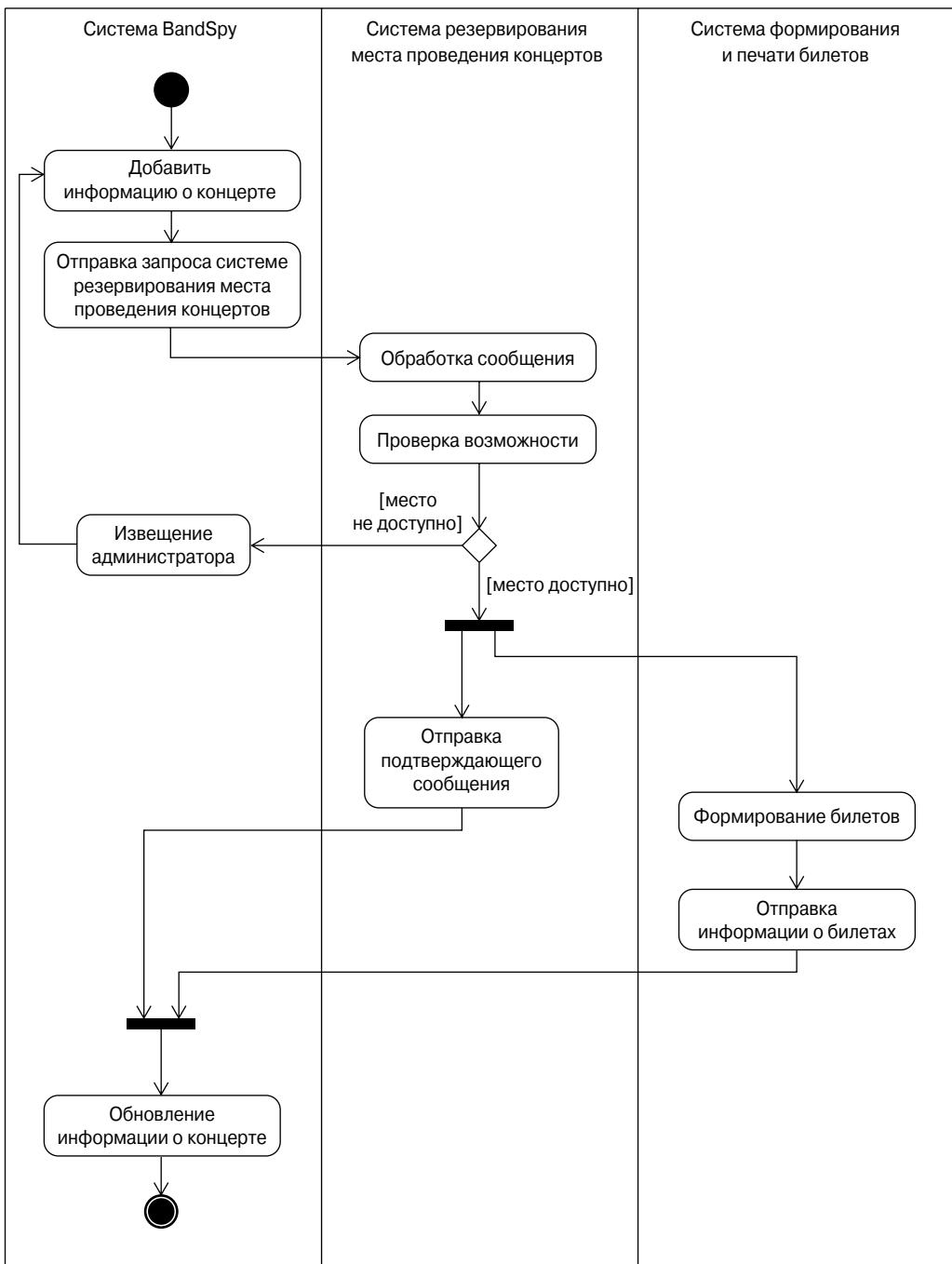


Рис. 2.10.

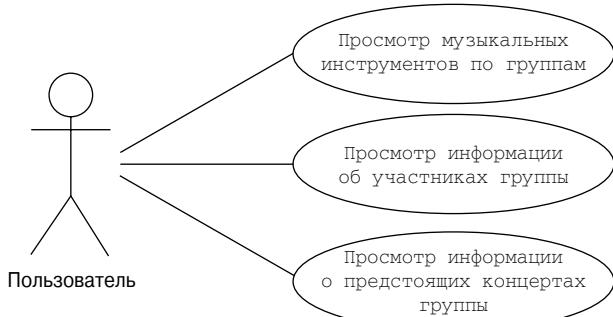


Рис. 2.11.

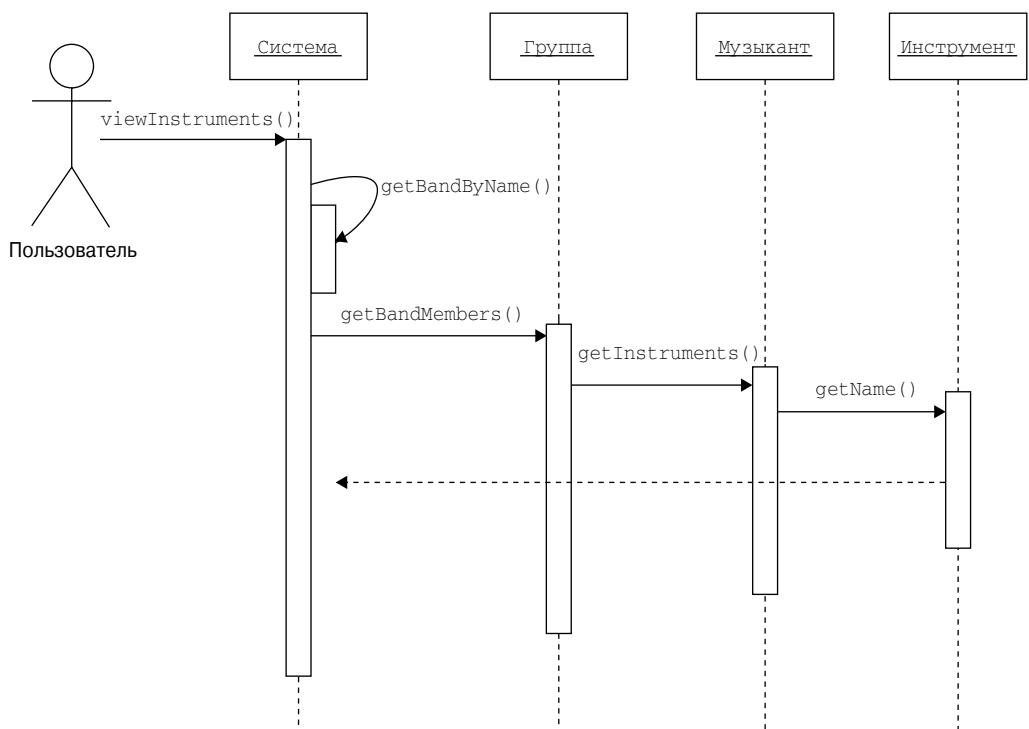


Рис. 2.12.

Объекты, которые участвуют в прецеденте, находятся в верхней части диаграммы и изображаются в виде прямоугольников. В общем случае прямоугольник с подчеркнутым именем обозначает экземпляры данного класса. Пунктирная вертикальная линия, исходящая из объектов, изображает линию жизни объекта (object's lifeline). В данном случае она не содержит специальных символов “создания” и “уничтожения” объектов. Предполагается, что они были созданы перед началом рассматриваемого прецедента. Однако, если на диаграмме необходимо показать процесс создания объекта, можно воспользоваться сообщением с именем `создать`. Для уничтожения объектов используется символ X, который помещается на конце линии жизни объекта.

Вертикальный узкий прямоугольник изображает блок активации объекта (*object's activation*). Он представляет период времени, в течение которого объект участвует в выполнении определенных операций или действий. Чем длиннее прямоугольник, тем дольше объект вовлечен в действие.

В данную диаграмму последовательностей впервые включен объект Система. В приложении BandSpy данный объект отвечает за получение всех сообщений, отправленных посредством Web-интерфейса, т.е. объект Система не позволяет внешним пользователям напрямую взаимодействовать с объектами системы BandSpy, предоставляя тем самым единую точку входа.

Процесс отправки сообщений изображают с помощью линии со стрелкой. Название данной линии соответствует названию метода. Однако необязательно придерживаться этой формальности — можно применять другие принципы для именования сообщений. Таким образом, для прецедента Просмотр музыкальных инструментов по группам последовательность выполняемых действий имеет следующий вид.

1. Пользователь генерирует запрос на получение информации о музыкальных инструментах музыкальной группы, вводя ее имя.
2. На основе введенного пользователем имени объект Система получает ссылку на группу, выполняя *самовызов* (*self-call*) метода с именем `getBandByName()` (получить ссылку на группу по ее имени).
3. Получив ссылку на группу, объект Система вызывает метод с именем `getBandMembers()` (получить участников группы).
4. Объект типа Band вызывает метод с именем `getInstrument()` для каждого участника данной группы.
5. Объект типа Musician вызывает метод с именем `getName()` для каждого из используемых музыкальных инструментов.
6. Полученная информация о музыкальных инструментах отправляется обратно объекту Система, который отображает ее пользователю.

Диаграммы состояний

Диаграммы состояний (*state diagrams*) полезны в тех случаях, когда необходимо показать изменения состояний конкретного объекта в течение его жизни. Изменения могут состоять как в простой модификации значений атрибутов объекта, так и в инициализации процесса ожидания ответа от внешних ресурсов.

В приложении BandSpy особое внимание следует уделять классу Performance. Построим для него соответствующую диаграмму состояний. При создании экземпляра класса Performance соответствующий объект должен обратиться к системе резервирования места проведения концертов и ожидать результатов обработки своего запроса. При получении положительного ответа информацию о проведении данного концерта следует сохранить. В противном случае объект класса Performance должен отправить соответствующее сообщение объекту Система, а затем должен быть удален. Диаграмма состояний для класса Performance представлена на рис. 2.13.

Черный круг указывает на начальное состояние объекта. Переходы между состояниями обозначаются посредством линий со стрелками. После создания объекта класса Performance и отправки сообщения системе резервирования данный объект переходит в состояние ожидания. Он будет находиться в данном состоянии до тех пор, пока не получит ответное сообщение от системы резервирования.



Рис. 2.13.

На диаграмме состояний, изображенной на рис. 2.13, добавлено новое обозначение — **комментарии** (note). В данном случае комментарии указывают, что объект класса *Performance* будет находиться в состоянии ожидания не более одной минуты. Комментарии можно использовать на любых диаграммах UML, когда необходимо предоставить дополнительную информацию об объекте или процессе.

Диаграммы компонентов и развертывания

Последний вид диаграмм, который будет рассмотрен в данной главе, — **диаграмма компонентов** (component diagram). Диаграмма компонентов представляет высоконивневое, абстрактное описание разрабатываемого программного обеспечения. Для приложения BandSpy диаграмма компонентов фактически будет состоять из одного компонента на стороне Web-сервера. С технической точки зрения такие элементы, как Web-сервер являются составляющими другого вида диаграмм UML — **диаграмм развертывания** (deployment diagrams). На диаграмме развертывания обычно изображают физические элементы инфраструктуры разрабатываемого приложения. К ним, например, относятся разнообразные серверы, на которых запущено приложение.

На рис. 2.14 представлена диаграмма развертывания для приложения BandSpy. В виде кубов изображаются узлы (nodes). Под узлами понимают отдельное аппаратное средство или, в более концептуальном смысле, отдельное программное обеспечение, такое как Web-сервер.

Внутри узлов представлены компоненты, которые изображаются в виде прямоугольников с двумя прямоугольниками меньшего размера, наложенными на его левую сторону. Посредством пунктирной линии со стрелкой отображаются **зависимости** (dependencies)

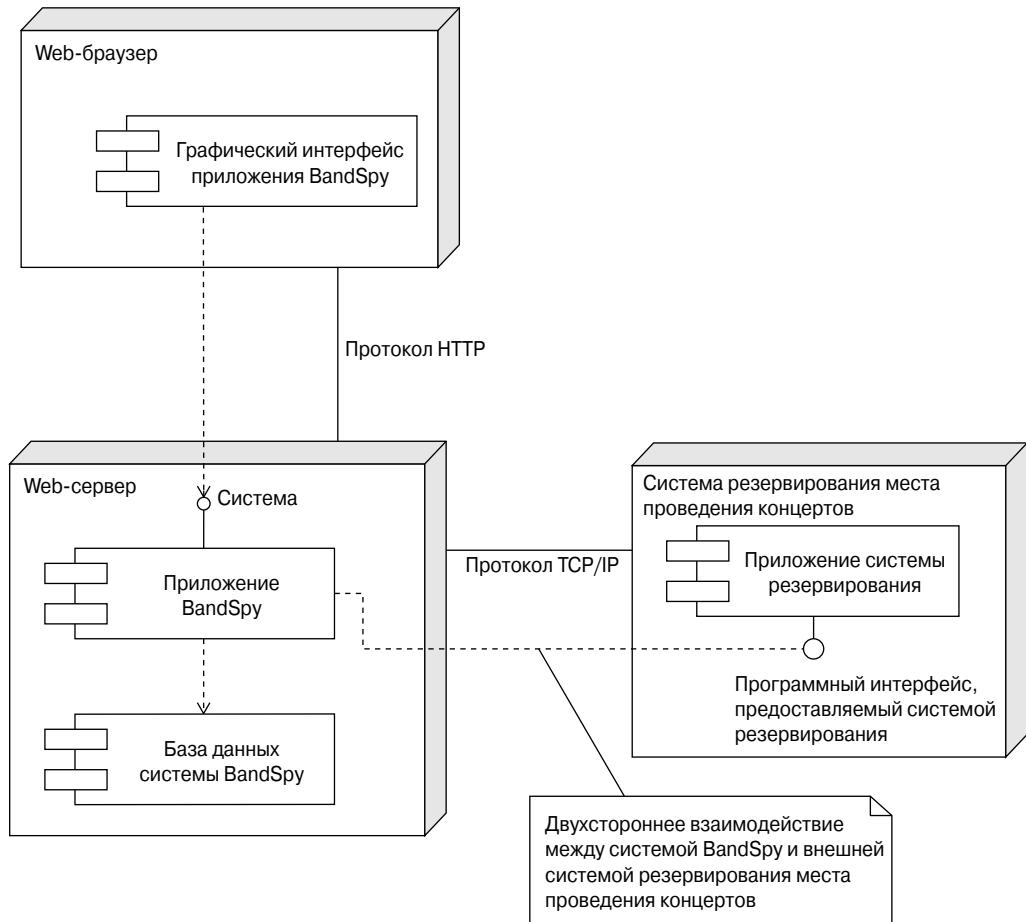


Рис. 2.14.

между компонентами. Например, Web-интерфейс приложения BandSpy связан с Web-сервером. Причем одновременно с ним могут работать несколько клиентов. В свою очередь Web-сервер никак не зависит от Web-интерфейса. Сплошная линия отображает физическое соединение между узлами. Например, Web-сервер взаимодействует с Web-интерфейсом посредством протокола HTTP.

Интерфейс компонента изображают в виде круга, соединенного прямой линией с компонентом. Как указывалось ранее, приложение BandSpy имеет для пользователей единую точку входа в систему — интерфейс Система. Еще один интерфейс на рис. 2.14 показан для узла системы резервирования места проведения концертов. Владельцы данной системы предоставили перечень открытых методов, которые можно использовать для взаимодействия с этой системой. При этом нет необходимости быть в курсе деталей реализации и работы системы резервирования. Необходимо только знать конкретные методы интерфейса API (Application Programming Interface — программный интерфейс приложения), используемые в рамках данной системы.

Резюме

UML — стандартизованный и гибкий язык, предназначенный для проектирования и моделирования разрабатываемого программного обеспечения. В данной главе были рассмотрены следующие виды диаграмм:

- диаграммы прецедентов;
- диаграммы классов;
- диаграммы видов деятельности;
- диаграммы последовательностей;
- диаграммы состояний;
- диаграммы компонентов и развертывания.

В этой главе также были затронуты вопросы проектирования программного обеспечения, связанные с моделированием предметной области и определением требований.

Помните, нет необходимости использовать все виды диаграмм. Так, диаграммы прецедентов и классов всегда полезны. В то же время нет необходимости разрабатывать другие виды диаграмм для каждого класса или прецедента. Например, диаграмму видов деятельности следует использовать в тех случаях, когда необходимо лучше понять некоторый процесс для конкретного прецедента. Диаграмму состояний полезно использовать для тех объектов, для которых изменение состояний представляет сложный процесс в рамках нескольких прецедентов. Диаграммы последовательностей помогут разобраться, как взаимодействуют объекты. Однако при этом не следует отображать абсолютно все сообщения, передаваемые объектами друг другу при работе приложения.

Глава 4 посвящена вопросам, связанным с использованием шаблонов проектирования. Шаблоны проектирования предоставляют механизмы для описания отношений между объектами, которые можно повторно использовать для решения разнообразных задач, возникающих при разработке программного обеспечения. Поскольку для описания шаблонов проектирования используют диаграммы классов, изображаемые с помощью языка UML, у читателя теперь есть необходимые навыки для изучения материала, изложенного в следующей главе.

3

Объектный подход в действии

Ознакомившись в первых двух главах с основами объектно-ориентированного подхода, можно рассмотреть более сложную задачу создания на его основе реального приложения.

В качестве такого приложения рассмотрим адресную книгу или контакт-менеджер, предназначенный для управления информацией о людях (индивидуумах) и организациях, и позволяющий пользователям находить и редактировать информацию о контактах. Материал этой главы познакомит читателя с основными принципами проектирования реальных объектно-ориентированных приложений. Попутно будет показано, как применять основные принципы объектно-ориентированной разработки, такие как повторное использование кода, инкапсуляция, наследование и, конечно, абстракция.

Создание менеджера контактов

Приложение для управления контактной информацией — менеджер контактов — позволяет пользователю отслеживать информацию о людях и организациях (почтовый адрес, адрес электронной почты и номер телефона), а также отношения между ними. По существу, это аналог адресной книги в Microsoft Outlook.

Приведем краткое высокоуровневое описание требований к приложению.

- Приложение будет отслеживать информацию о людях и организациях в базе данных и отображать ее на Web-странице.
- Каждому контакту может соответствовать один и более почтовых и электронных адресов, а также телефонных номеров, а может не соответствовать ни одного.
- Человек может быть связан лишь с одной организацией.
- К каждой организации относится один или несколько сотрудников (а может и ни одного).

Представим эти взаимосвязи в виде диаграммы UML. Однако, прежде чем приступить к ее построению, следует сделать одно замечание. Цель данной главы — продемонстрировать эволюцию видения системы в ходе ее создания, изменение и выявление новых решений в процессе разработки. Поэтому в данной главе описывается эволюционный процесс развития приложения, а не листинг его кода. Рассматриваемый конкретный пример позволит лучше понять, как применять на практике принципы объектно-ориентированного подхода.

Диаграммы UML для адресной книги

Запустите любое приложение, предназначенное для построения диаграмм UML, и создайте новый файл с именем `ContactManager.[расширение]`, где `[расширение]` — это расширение файла, присваиваемое приложением по умолчанию (например, `.dia` для диаграмм, созданных с использованием редактора Dia).

Сначала создайте классы для трех разных видов контактной информации, которая будет предоставлена в приложении. В данном случае это будут классы почтового адреса, электронного адреса и номера телефона. Можно также создать классы для любой другой контактной информации. Свойства указанных классов приведены в следующей таблице.

Класс	Свойства
Address	street1 street2 city state zip (почтовый индекс) type (домашний, рабочий и т.д.)
EmailAddress	email type
PhoneNumber	number extension type

Эти классы предназначены лишь для хранения и отображения информации, поэтому их методы пока определять не нужно, и соответственно третья секция в обозначении каждого класса остается пустой. Все свойства являются открытыми, поэтому в качестве префикса в имени атрибута используется символ “плюс”. На рис. 3.1 показано текущее представление этих классов в виде обозначений UML.

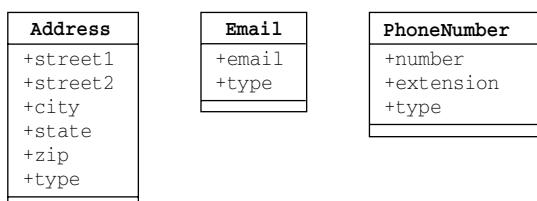


Рис. 3.1.

Далее необходимо разработать классы `Individual` (человек) и `Organization` (организация). Человек имеет имя, фамилию, уникальный идентификатор (поле `id` в базе данных), ему соответствует набор контактных данных (адрес электронной почты, почтовый адрес и телефонный номер), представляемый в виде коллекции, организация и должность. Необходимо также обеспечить возможность добавлять другие виды контактной информации. На рис. 3.2 изображена предыдущая диаграмма UML с добавлением класса `Individual`.

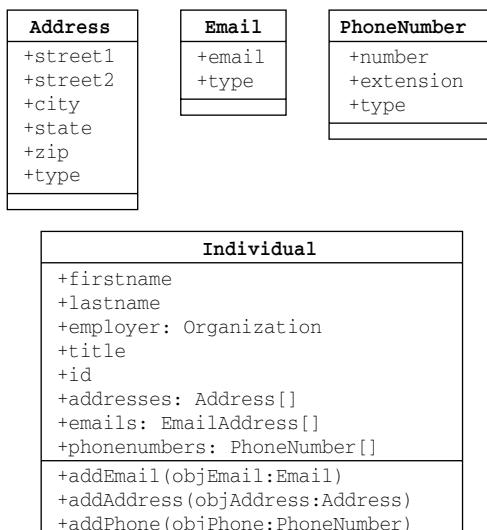


Рис. 3.2.

Организация имеет название, уникальный идентификатор, с ней связан набор контактных данных (таких же, как и у отдельного человека), методы для их добавления, а также “коллекция” индивидуумов (сотрудников). На рис. 3.3 изображена та же диаграмма UML после включения класса `Organization`.

Из диаграммы видно, что классы `Individual` и `Organization` имеют множество одинаковых свойств и методов. По большому счету это свидетельствует о том, что можно улучшить проектное решение с помощью наследования, значительно сэкономив трудозатраты и повысив гибкость приложения. Можно создать другой класс (в данном случае `Entity`), в котором объединить элементы, характерные для классов `Individual` и `Organization`, и дать возможность этим классам совместно использовать общий для них код. На диаграммах UML свойства и методы отображаются только в тех классах, в которых они фактически реализованы. Поэтому все одинаковые свойства и методы классов `Individual` и `Organization` необходимо переместить в символическое представление класса `Entity` (рис. 3.4).

В данном случае свойство `name` (имя) класса `Entity` перекрывается в производном классе `Individual`. Поэтому при извлечении значения свойства `name` будет возвращена строка, содержащая фамилию и имя — `"lastname, firstname"`. Таким образом объекты классов `Organization` и `Individual` можно передавать в общую функцию, которая выводит имя, не прибегая к отдельным функциям для каждого из этих классов.

В UML также определены обозначения для отображения взаимосвязей. В данном примере необходимо показать, что классы `Individual` и `Organization` наследуются

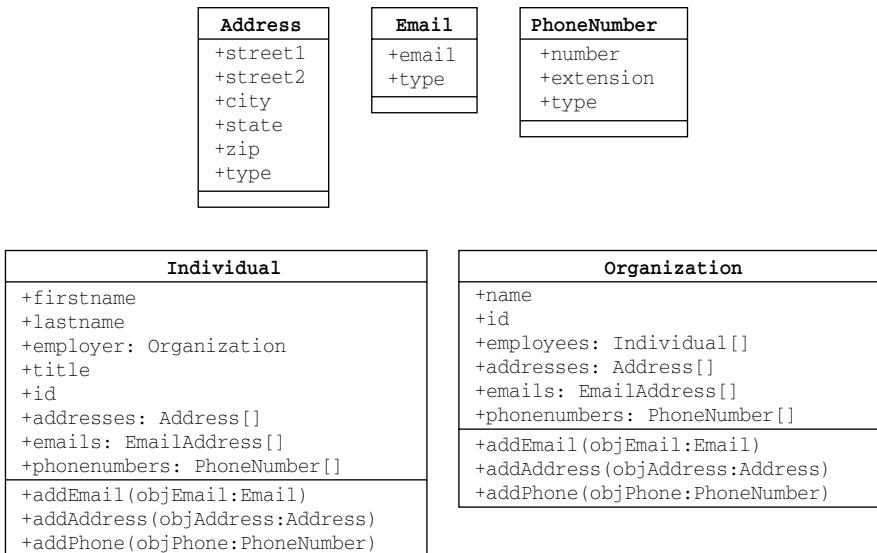


Рис. 3.3.

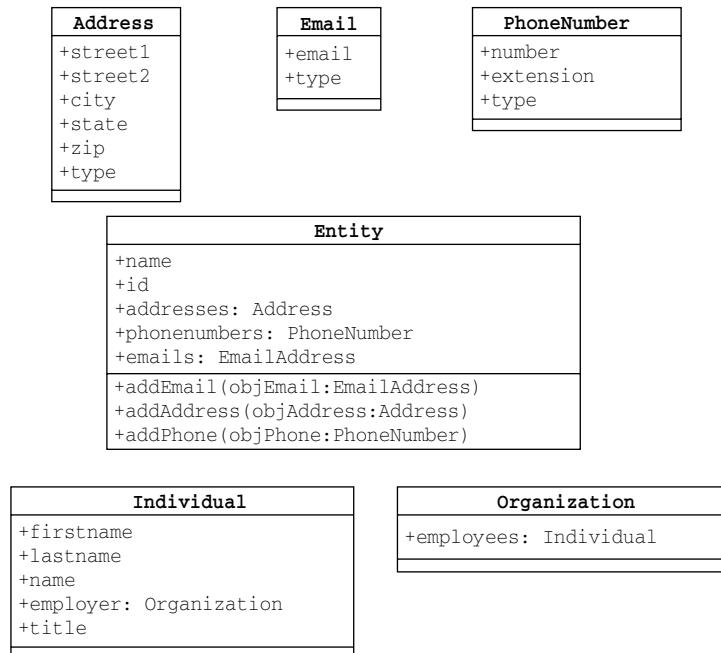


Рис. 3.4.

от класса Entity. В спецификации UML это отношение называется *обобщением* (generalization) и обозначается линией связи с полым треугольником на конце, направленным от дочернего класса к родительскому, как показано на рис. 3.5.

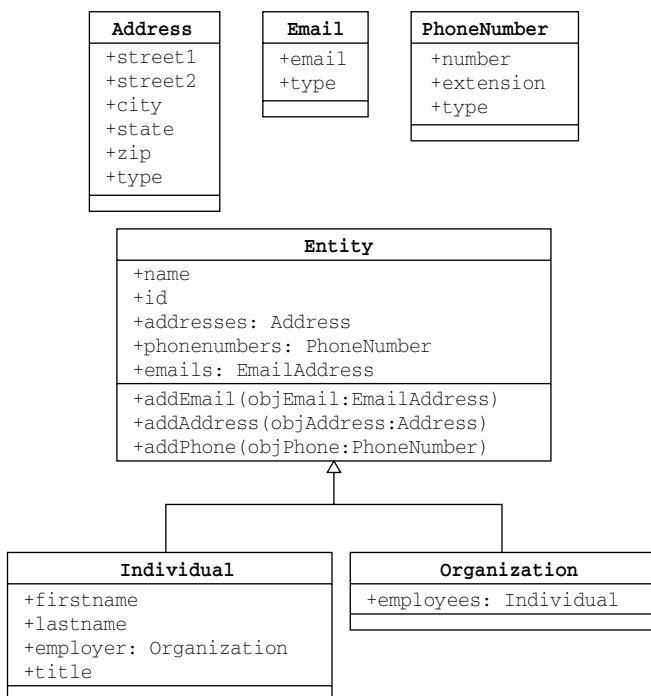


Рис. 3.5.

Теперь видно, что классы Individual и Organization наследуются от класса Entity. Использование линий связи для указания наследования позволяет при беглом взгляде на диаграмму легко определить, в каких отношениях состоят классы друг с другом.

На диаграмме следует отобразить еще один тип связи — использование классом Entity классов Address, Email и PhoneNumber. В спецификации UML это отношение называется *композицией* (composite) и обозначается с помощью линии с закрашенным ромбом на конце. Ромб изображается со стороны класса-агрегата, который является пользователем другого класса (рис. 3.6). Используемые классы обладают свойством *кратности* (multiplicity), которое указывает, какое количество объектов используется. В данном примере с каждой сущностью Entity могут быть связаны ноль, один или более экземпляров контактной информации, поэтому над соединительной линией рядом с используемым классом указывается $0..*$. Это означает, что класс-композит может ссылаться на 0 или более объектов данного типа. Эти обозначения показаны на рис. 3.6. Отобразив это отношение на диаграмме, можно легко понять, какие части приложения будут затронуты при изменении некоторой его части. В данном случае изменение классов Email, Address или PhoneNumber отразится на классах Entity, Individual и Organization.

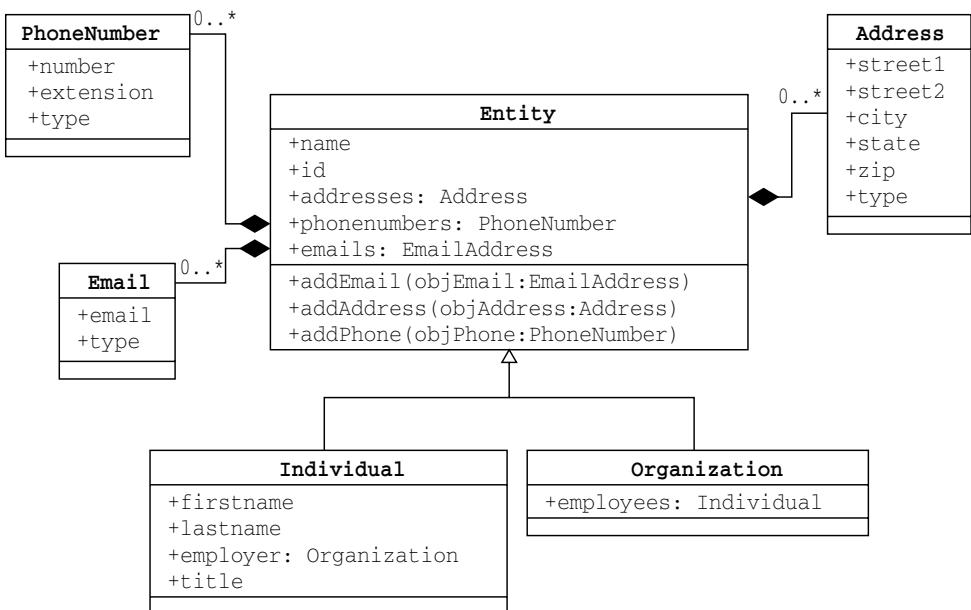


Рис. 3.6.

Класс Entity агрегирует объекты PhoneNumber, Email и Address, и в каждом из трех случаев он может включать один или несколько таких элементов либо вовсе не включать их. Так как классы Individual и Organization унаследованы от Entity, любой из них также может быть связан с одним или несколькими видами контактной информации.

При обсуждении инкапсуляции в главе 1 говорилось о том, что в целях безопасности данные целесообразно хранить в закрытых (private) переменных-членах, а для организации доступа к ним использовать методы доступа. Напомним также, что использование методов `__get` и `__set` несколько упрощает этот процесс. Из приведенной выше диаграммы классов видно, что требования к свойствам всех изображенных классов довольно просты, поэтому выполнить проверку корректности ввода их значений не составит труда. Однако для обеспечения общности кода и возможности его повторного использования эту функциональность целесообразно убрать из класса Entity и перенести в другой класс, добавив еще один уровень абстракции, реализующий общие для всех классов функции. Назовем этот родительский класс `PropertyObject`.

При работе с данными тоже целесообразно добавить еще один уровень абстракции, т.е. создать класс `DataManager`. Принимая во внимание отсутствие унифицированного средства для проверки данных, необходимо разработать интерфейс для проверки корректности ввода. Приведем код такого интерфейса.

```

<?php
interface Validator {
    abstract function validate();
}
?>
  
```

Сохраним этот код в файле `interface.Validator.php` и используем этот интерфейс для работы с новым классом `PropertyObject`, который будет рассмотрен ниже.

Класс PropertyObject

Следующий ниже код позволяет объединить интерфейс Validator с классом PropertyObject. Внесите код в файл под названием class.PropertyObject.php.

```
<?php
require_once('interface.Validator.php');

abstract class PropertyObject implements Validator {

    protected $propertyTable = array();
        //содержит пары "имя-значение",
        //которые связывают свойства с
        //именами полей базы данных
    protected $changedProperties = array();
        //список свойств, которые
        //подвергались модификации
    protected $data;
        // актуальная информация из
        //базы данных

    protected $errors = array();
        //любые ошибки ввода информации,
        //которые могли произойти

    public function __construct($arData) {
        $this->data = $arData;
    }

    function __get($propertyName) {
        if(!array_key_exists($propertyName, $this->propertyTable))
            throw new Exception("Неверное имя свойства\"$propertyName\"!");
        if(method_exists($this, 'get' . $propertyName)) {
            return call_user_func(array($this, 'get' . $propertyName));
        } else {
            return $this->data[$this->propertyTable[$propertyName]];
        }
    }

    function __set($propertyName, $value) {
        if(!array_key_exists($propertyName, $this->propertyTable))
            throw new Exception("Неверное имя свойства\"$propertyName\"!");
        if(method_exists($this, 'set' . $propertyName)) {
            return call_user_func(
                array($this, 'set' . $propertyName),
                $value
            );
        } else {
            //Если значение свойства действительно изменено
            //и оно все еще не находится в массиве
            //,$changedProperties, добавить его
            if($this->propertyTable[$propertyName] != $value &&
               !in_array($propertyName, $this->changedProperties)) {
                $this->changedProperties[] = $propertyName;
            }
        }
        //Теперь устанавливаем новое значение
        $this->data[$this->propertyTable[$propertyName]] = $value;
    }
}
```

```

    }

    function validate() {
    }

}

?>

```

Рассмотрим более подробно, что получилось в результате. Созданы четыре защищенные переменных-члена. Защищенные переменные-члены видны только производным классам; они не видны во внешней части приложения, где используются эти объекты.

Массив `$propertyTable` содержит перечень имен свойств и соответствующих им имен полей в базе данных. Зачастую имена полей в базе данных содержат префикс, указывающий на тип данных поля. Например, имя `entities.sname1` может соответствовать полю в таблице сущностей Entity, имеющему тип `string` и содержащему фамилию. Однако имя `sname1` нельзя назвать подходящим для свойства объекта, поэтому необходимо обеспечить механизм для конвертирования имен полей из базы данных в осмысленные имена свойств.

Массив `$changedProperties` содержит список имен свойств, которые подверглись изменению.

Массив `$data` является ассоциативным массивом имен полей базы данных и их значений. Он формируется в конструкторе, а структура данных возвращается непосредственно функцией `pgsql_fetch_assoc()`. Такой метод, как станет ясно из дальнейшего, существенно упрощает создание нужных объектов напрямую по запросу к базе данных.

Последняя переменная-член `$errors` будет содержать массив имен полей и сообщений о возможных ошибках при выполнении метода `validate()` (объявленного в интерфейсе `Validate`).

Класс объявлен как абстрактный (`abstract`) по двум причинам. Во-первых, класс `PropertyObject` сам по себе не очень полезен. Прежде чем использовать его методы, необходимо реализовать классы, расширяющие `PropertyObject`. Во-вторых, не реализован требуемый метод `validate()`. Поскольку этот метод помечен в классе `PropertyObject` как абстрактный, абстрактным необходимо объявить и сам класс. Тогда указанный метод придется реализовать во всех производных классах. Попытка использовать класс, расширяющий класс `PropertyObject`, не реализовав в нем метод `validate()`, приведет к ошибке во время выполнения программы.

В приведенном примере реализован весьма упрощенный конструктор. Он попросту принимает в качестве параметра ассоциативный массив, который наиболее вероятно будет получен в результате выполнения запроса к базе данных, и записывает его в защищенную переменную-член `$data`. При создании большинства подклассов `PropertyObject` этот конструктор потребуется перекрыть, сделав его более содержательным.

И наконец, обратите внимание на методы доступа `__get()` и `__set()`. Так как данные хранятся в элементе `$data`, необходимо иметь возможность преобразовывать имена свойств в фактические имена полей базы данных. Стока, содержащая код `$this->data[$this->propertyTable[propertyName]]` делает именно это.

Не беспокойтесь, если принципы использования массивов `$data` и `$propertyTable` вам еще не совсем ясны. Все станет ясно, как только мы рассмотрим конкретный пример.

Типы контактной информации

Разработав класс `PropertyObject`, можно начинать его использовать. Ниже приводится описание классов `Address`, `EmailAddress` и `PhoneNumber`.

В коде встречаются ссылки на класс `DataManager`, который является классом-оболочкой для работы с базой данных. Эта оболочка полностью инкапсулирует код взаимодействия с данными. Класс `DataManager` будет рассмотрен несколько позже.

Внесите следующий код (класс `Address`) в файл под названием `class.Address.php`.

```
<?php
require_once('class.PropertyObject.php');

class Address extends PropertyObject {

    function __construct($addressid) {
        $arData = DataManager::getAddressData($addressid);

        parent::__construct($arData);

        $this->propertyTable['addressid'] = 'addressid';
        $this->propertyTable['id'] = 'addressid';
        $this->propertyTable['entityid'] = 'entityid';
        $this->propertyTable['address1'] = 'saddress1';
        $this->propertyTable['address2'] = 'saddress2';
        $this->propertyTable['city'] = 'scity';
        $this->propertyTable['state'] = 'cstate';
        $this->propertyTable['zipcode'] = 'spostalcode';
        $this->propertyTable['type'] = 'stype';
    }

    function validate() {
        if(strlen($this->state) != 2) {
            $this->errors['state'] = 'Пожалуйста, введите правильно штат';
        }

        if(strlen($this->zipcode) != 5 &&
           strlen($this->zipcode) != 10) {
            $this->errors['zipcode'] = 'Пожалуйста, введите пяти- или
девятизначный ZIP-код';
        }

        if(!$this->address1) {
            $this->errors['address1'] = 'Адрес 1 является обязательным полем';
        }

        if(!$this->city) {
            $this->errors['city'] = 'Город является обязательным полем';
        }

        if(sizeof($this->errors)) {
            return false;
        } else {
            return true;
        }
    }

    function __toString() {
        return $this->address1 . ', ' .
               $this->address2 . ', ' .
               $this->city . ', ' .
               $this->state . ' ' . $this->zipcode;
    }
}
```

```

    }
?>

```

Так как большую часть работы выполняет класс `PropertyObject`, в классе `Address` остается реализовать только два метода (причем метод `__toString()` добавлен только для удобства). В конструкторе впервые используется массив `$propertyTable`. Список необходимых свойств этого класса определен на диаграмме UML, созданной на первом этапе разработки приложения (в начале этой главы). Учитывая перечень свойств объекта, можно принять несколько решений, касающихся структуры таблицы базы данных. Вообще говоря, на каждое свойство требуется по одному полю. Однако поскольку этот класс связан с классом `Entity`, надо также хранить ссылку на этот родительский класс. Для создания таблицы `Address` можно использовать следующий SQL-запрос.

```

CREATE TABLE "entityaddress" (
    "addressid" SERIAL PRIMARY KEY NOT NULL,
    "entityid" int,
    "saddress1" varchar(255),
    "saddress2" varchar(255),
    "scity" varchar(255),
    "cstate" char(2),
    "spostalcode" varchar(10),
    "stype" varchar(50),
    CONSTRAINT "fk_entityaddress_entityid"
        FOREIGN KEY ("entityid") REFERENCES "entity"("entityid")
);

```

В имени каждого поля создаваемой таблицы тип содержащихся в нем данных отражается при помощи односимвольного префикса. Это позволяет легко узнать, какой тип данных хранится в поле. Соглашения об именовании важны как с точки зрения проектирования базы данных, так и для кода приложения.

Массив `propertyTable` используется в классе `Address` для поддержки соответствия дружественных для разработчика названий свойств (таких, как `city`, `state` и `zipcode`) и менее дружественных имен полей базы данных (такие как `scity`, `sstate` и `spostalcode`). Заметим, что в массиве `propertyTable` одному полю базы данных может соответствовать несколько имен свойств. Эта таблица соответствий позволяет обращаться к первичному ключу адреса двумя способами: `$objAddress->addressed` либо `$objAddress->id`.

Преобладающая часть кода класса `Address` обеспечивает реализацию бизнес-логики и проверку корректности данных. Здесь практически нет лишнего кода. Единственной обязанностью этого класса является предоставление информации о себе и проверка корректности своих данных. Все остальные функции выполняют классы `DataManager` (который вскоре будет рассмотрен более детально) и `PropertyObject`.

Далее приведен код класса `Email`, который очень похож на класс `Address`. Внесите его в файл под названием `class.EmailAddress.php`.

```

<?php
require_once('class.PropertyObject.php');

class EmailAddress extends PropertyObject {

    function __construct($emailid) {
        $arData = DataManager::getEmailData($emailid);

        parent::__construct($arData);
}

```

```

$this->propertyTable['emailid'] = 'emailid';
$this->propertyTable['id'] = 'emailid';
$this->propertyTable['entityid'] = 'entityid';
$this->propertyTable['email'] = 'semail';
$this->propertyTable['type'] = 'stype';
}

function validate() {
    if (!$this->email) {
        $this->errors['email'] = 'Необходимо указать адрес электронной почты.';
    }

    if (sizeof($this->errors)) {
        return false;
    } else {
        return true;
    }
}

function __toString() {
    return $this->email;
}
}
?>

```

В этом файле содержится очень мало вспомогательного кода. В конструкторе просто делается выборка из базы данных и заполняется массив `propertyTable`. Все остальное — это проверка корректности данных. Свойства класса `Email` и структура соответствующей таблицы базы данных определяются приведенной выше диаграммой UML.

Таблица базы данных для таблицы `entityemail` создается следующим образом.

```

CREATE TABLE "entityemail" (
    "emailid" SERIAL PRIMARY KEY NOT NULL,
    "entityid" int,
    "semail" varchar(255),
    "stype" varchar(50),
    CONSTRAINT "fk_entityemail_entityid"
        FOREIGN KEY ("entityid") REFERENCES "entity"("entityid")
);

```

Класс `PhoneNumber` работает аналогично `Address` и `Email`. Приведем его код, который нужно сохранить в файле `class.PhoneNumber.php`.

```

<?php
require_once('class.PropertyObject.php');

class PhoneNumber extends PropertyObject {

    function __construct($phoneid) {
        $arData = DataManager::getPhoneNumberData($phoneid);

        parent::__construct($arData);

        $this->propertyTable['phoneid'] = 'phoneid';
        $this->propertyTable['id'] = 'phoneid';
        $this->propertyTable['entityid'] = 'entityid';
        $this->propertyTable['number'] = 'snumber';
        $this->propertyTable['extension'] = 'sextension';
        $this->propertyTable['type'] = 'stype';
    }
}

```

```

function validate() {
    if (!$this->number) {
        $this->errors['number'] = 'Нужно указать номер телефона.';
    }

    if(sizeof($this->errors)) {
        return false;
    } else {
        return true;
    }
}

function __toString() {
    return $this->number .
           ($this->extension ? ' x' . $this->extension : '');
}
?>

```

Приведем SQL-запрос для создания таблицы entityphone.

```

CREATE TABLE "entityphone" (
    "phoneid" int SERIAL PRIMARY KEY NOT NULL,
    "entityid" int,
    "snumber" varchar(20),
    "sextension" varchar(20),
    "stype" varchar(50),
    CONSTRAINT "fk_entityemail_entityid"
        FOREIGN KEY ("entityid") REFERENCES "entity"("entityid")
);

```

Класс DataManager

Теперь можно обратиться к классу `DataManager`. В нем и в других примерах кода, связанного с работой базы данных в этой главе, используется база данных PostgreSQL, хотя подобный класс может успешно работать и с другими реляционными СУБД, в том числе Oracle.

Основной обязанностью класса `DataManager` является обеспечение доступа к данным. Сосредоточение всего кода для работы с базой данных в одном классе в дальнейшем существенно упростит изменение вида базы данных или параметров соединения. Все методы класса объявлены как статические, потому что класс не содержит ни одной переменной-члена. Обратите внимание на использование статической переменной в функции `getConnection()`. Это сделано для того, чтобы при запросе каждой страницы было открыто лишь одно соединение с базой данных. При установке соединения с базой данных используется много системных ресурсов, поэтому предотвращение бесполезных соединений помогает улучшить качество функционирования системы. Создайте файл под названием `class.DataManager.php` и введите в него следующий код.

```

<?php
require_once('class.Entity.php'); //Это понадобится позже
require_once('class.Individual.php');
require_once('class.Organization.php');

class DataManager
{
    private static function _getConnection() {
        static $hDB;

```

```

if(isset($hDB)) {
    return $hDB;
}

$hDB = pg_connect("host=localhost port=5432 dbname=sample_db
                    user=phpuser password=phppass");
    or die("Не удается установить соединение с базой данных!");
return $hDB;
}

public static function getAddressData($addressID) {
    $sql = "SELECT * FROM \"entityaddress\" WHERE \"addressid\" = $addressID";
    $res = pg_query(DataManager::__getConnection(), $sql);
    if(! ($res && pg_num_rows($res))) {
        die("Невозможно получить данные адреса для $addressID");
    }
    return pg_fetch_assoc($res);
}

public static function getEmailData($emailID) {
    $sql = "SELECT * FROM \"entityemail\" WHERE \"emailid\" = $emailID";
    $res = pg_query(DataManager::__getConnection(), $sql);

    if(! ($res && pg_num_rows($res))) {
        die("Невозможно получить данные электронного адреса для $emailID");
    }

    return pg_fetch_assoc($res);
}

public static function getPhoneNumberData($phoneID) {
    $sql = "SELECT * FROM \"entityphone\" WHERE \"phoneid\" = $phoneID";
    $res = pg_query(DataManager::__getConnection(), $sql);
    if(! ($res && pg_num_rows($res))) {
        die("Невозможно получить номер телефона для $phoneID");
    }

    return pg_fetch_assoc($res);
}
?>
```

Класс DataManager предоставляет структуры данных, используемые для заполнения элемента \$data подкласса PropertyObject. Каждая функция возвращает данные одного из типов. Позже в этот класс придется добавить несколько новых функций.

Все методы этого класса объявлены как статические. При использовании статических методов все переменные-члены тоже должны быть статическими. Для использования методов статических классов не требуется создавать экземпляров (выполнять инстанцирование). Это имеет смысл в нескольких случаях. Рассмотрим класс Math (математика), который предоставляет методы squareRoot () (извлечь квадратный корень), power () (возвести в степень) и cosine () (вычислить косинус) и имеет свойства, включающие математические константы e и pi. Каждый экземпляр этого класса представляет собой одну и ту же математику. Квадратный корень из 2 не меняется, 4 в кубе всегда будет 64 и две константы, ясное дело, являются постоянными. Нет нужды создавать отдельный экземпляр этого класса, так как его состояние и свойства никогда не меняются. При такой реализации класса Math его функции могут быть объявлены статическими.

Класс `DataManager` во многом аналогичен. Все его функции самодостаточны. Для взаимодействия с функциями не используются нестатические переменные-члены. В этом классе отсутствуют свойства. В результате методы класса можно вызывать, используя оператор вызова статического метода `::`. Поскольку все методы являются статическими, вам никогда не придется инициализировать объект с помощью оператора `$obj=new DataManager()`, а затем вызывать методы, используя оператор `$obj->getEmail()`. Вместо этого можно использовать простой синтаксис `DataManager::getEmail()`.

Классы Entity, Individual и Organization

Создав все вспомогательные классы, можно перейти к основе приложения: классу `Entity` и его подклассам.

Не забывайте обновлять диаграммы UML по ходу изменения иерархии объектов, отслеживая таким образом ход разработки. К настоящему моменту создан класс `PropertyObject` и все его наследники, а также класс `DataManager`, который не наследуется, а попросту представляет набор функций для работы с данными. Класс `PropertyObject` реализует абстрактный интерфейс `Validator`. На рис. 3.7 изображена обновленная диаграмма.

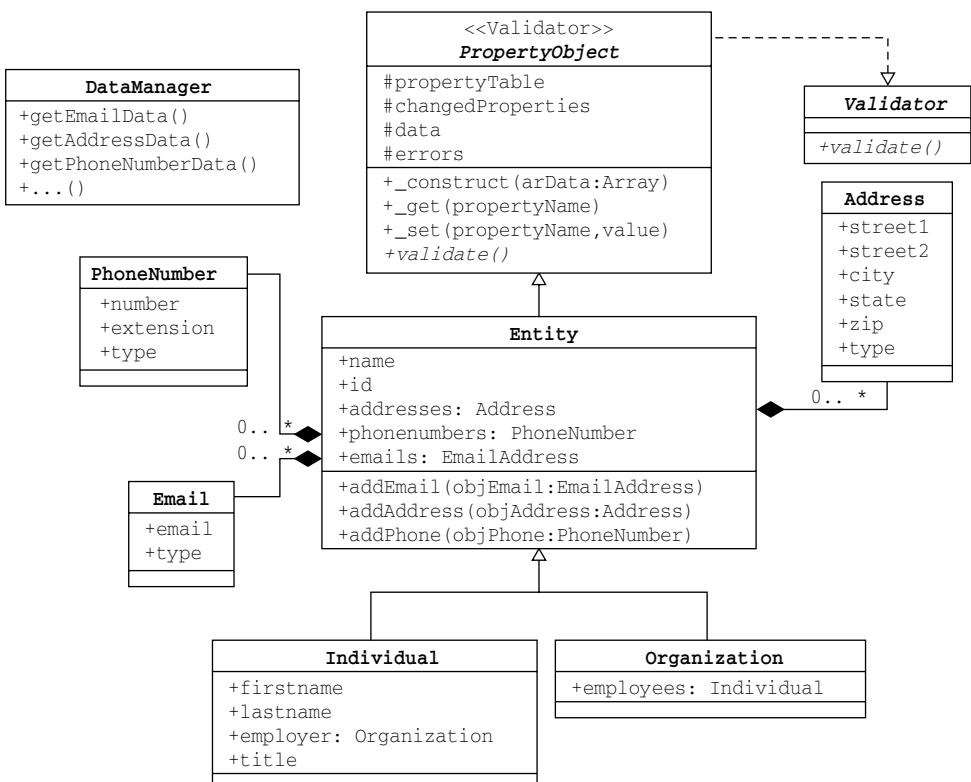


Рис. 3.7.

Теперь можно приступать к разработке классов Entity, Individual и Organization. Приведенный ниже код описывает полностью реализованный класс Entity. Его нужно поместить в файл class.Entity.php.

```
<?php

require_once('class.PropertyObject.php');
require_once('class.PhoneNumber.php');
require_once('class.Address.php');
require_once('class.EmailAddress.php');

abstract class Entity extends PropertyObject {

    private $_emails;
    private $_addresses;
    private $_phonenumbers;

    public function __construct($entityID) {
        $arData = DataManager::getEntityData($entityID);

        parent::__construct($arData);

        $this->propertyTable['entityid'] = 'entityid';
        $this->propertyTable['id'] = 'entityid';
        $this->propertyTable['name1'] = 'sname1';
        $this->propertyTable['name2'] = 'sname2';
        $this->propertyTable['type'] = 'ctype';

        $this->_emails = DataManager::getEmailObjectsForEntity($entityID);
        $this->_addresses = DataManager::getAddressObjectsForEntity($entityID);
        $this->_phonenumbers =
            DataManager::getPhoneNumberObjectsForEntity($entityID);

    }

    function setID($val) {
        throw new Exception('Вы не можете изменять значение поля ID!');
    }

    function setEntityID($val) {
        $this->setID($val);
    }

    function phonenumbers($index) {
        if(!isset($this->_phonenumbers[$index])) {
            throw new Exception('Задан некорректный номер телефона!');
        } else {
            return $this->_phonenumbers[$index];
        }
    }

    function getNumberOfPhoneNumbers() {
        return sizeof($this->_phonenumbers);
    }

    function addPhoneNumber(PhoneNumber $phone) {
        $this->_phonenumbers[] = $phone;
    }

    function addresses($index) {
        if(!isset($this->_addresses[$index])) {
            throw new Exception('Задан некорректный адрес!');
        } else {
```

```

        return $this->_addresses[$index];
    }

    function getNumberOfAddresses() {
        return sizeof($this->_addresses);
    }

    function addAddress(Address $address) {
        $this->_addresses[] = $address;
    }

    function emails($index) {
        if(!isset($this->_emails[$index])) {
            throw new Exception('Задан некорректный адрес электронной почты!');
        } else {
            return $this->_emails[$index];
        }
    }

    function getNumberOfEmails() {
        return sizeof($this->_emails);
    }

    function addEmail(Email $email) {
        $this->_emails[] = $email;
    }

    public function validate() {
        //Добавьте процедуры проверки
    }
}
?>
```

Перемещая всю функциональность методов доступа в родительский класс `PropertyObject`, вы упрощаете класс `Entity` (сущность) и этим обеспечиваете сужение его обязанностей до реализации самой сущности.

Класс `Entity` объявлен как абстрактный, так как он сам по себе бесполезен. Все сущности относятся либо к классу `Individual`, либо к `Organization`. Объекты класса `Entity` инстанцировать не придется. Если класс объявлен абстрактным, для него нельзя создать экземпляров.

Для создания таблицы `entities` в базе данных PostgreSQL выполните следующий код.

```
CREATE TABLE "entities" (
    "entityid" SERIAL PRIMARY KEY NOT NULL,
    "name1" varchar(100) NOT NULL,
    "name2" varchar(100) NOT NULL,
    "type" char(1) NOT NULL
);
```

В класс `DataManager` необходимо добавить несколько новых функций: `getEntityData()` и `get[x]ObjectsForEntity`. Функция `getEntityData()` возвращает данные, необходимые для инициализации сущности, подобно аналогичным функциям для отдельных типов контактной информации. Далее следует код этой новой функции в файле `class.DataManager.php`.

```
//верхняя часть файла опущена для краткости
...
die("Не удалось получить информацию о телефонном номере для $phoneID");
```

```

    }

    return pg_getch_assoc($res);
}
}

```

```

public static function getEntityData($entityID) {
    $sql = "SELECT * FROM \"entities\" WHERE \"entityid\" = $entityID";
    $res = pg_query(DataManager::_getConnection(), $sql);
    if(! ($res && pg_num_rows($res))) {
        die("Не удалось получить сущность $entityID");
    }
    return pg_fetch_assoc($res);
}
}

```

?>

Для добавления функции `get [x]ObjectsForEntity` вставьте следующий код в конец файла `class.DataManager.php` сразу после функции `getEntityData`.

```

public static function getAddressObjectsForEntity($entityID) {
    $sql = "SELECT \"addressid\" FROM \"entityaddress\" WHERE \"entityid\" =
        $entityID";
    $res = pg_query(DataManager::_getConnection(), $sql);

    if(!$res) {
        die("Невозможно получить данные для сущности $entityID");
    }

    if(pg_num_rows($res)) {
        $objs = array();
        while($rec = pg_fetch_assoc($res)) {
            $objs[] = new Address($rec['addressid']);
        }
        return $objs;
    } else {
        return array();
    }
}

public static function getEmailObjectsForEntity($entityID) {

$sql = "SELECT \"emailid\" FROM \"entityemail\" "
    WHERE \"entityid\" = $entityID";
$res = pg_query(DataManager::_getConnection(), $sql);
if(!$res) {
die("Невозможно получить данные электронной почты для сущности $entityID");
}

if(pg_num_rows($res)) {
    $objs = array();
    while($rec = pg_fetch_assoc($res)) {
        $objs[] = new EmailAddress($rec['emailid']);
    }
    return $objs;
} else {
    return array();
}
}

public static function getPhoneNumberObjectsForEntity($entityID) {
}
}

```

```

$sql = "SELECT \"phoneid\" FROM \"entityphone\""
      WHERE \"entityid\" = $entityID";
$res = pg_query(DataManager::__getConnection(), $sql);

if (!$res) {
    die("Невозможно получить номер телефона для сущности $entityID");
}

if (pg_num_rows($res)) {
    $objs = array();
    while ($rec = pg_fetch_assoc($res)) {
        $objs[] = new PhoneNumber($rec['phoneid']);
    }
    return $objs;
} else {
    return array();
}
}

```

В качестве параметра этим функциям передается значение идентификатора ID сущности. Они делают запрос к базе данных и определяют, существует ли указанный адрес электронной почты, почтовый адрес либо номер телефона для данной сущности. Если да, то создается массив объектов EmailAddress, Address либо PhoneNumber. Затем этот массив передается обратно объекту Entity, где он хранится в соответствующей закрытой переменной-члене.

Вся основная работа по созданию класса Entity выполнена, осталось выполнить только простые действия по реализации классов Individual и Organization. Создайте файл class.Individual.php и введите следующий код.

```

<?php
require_once('class.Entity.php');
require_once('class.Organization.php');

class Individual extends Entity {

    public function __construct($userID) {
        parent::__construct($userID);

        $this->propertyTable['firstname'] = 'name1';
        $this->propertyTable['lastname'] = 'name2';

    }

    public function __toString() {
        return $this->firstname . ' ' . $this->lastname;
    }

    public function getEmployer() {
        return DataManager::getEmployer($this->id);
    }

    public function validate() {
        parent::validate();

        // проверка данных для индивидуума
    }
}
?>

```

Коротко и ясно. Процесс создания класса Individual существенно упрощается благодаря наследованию. В этом классе задаются несколько новых свойств, упрощающих доступ к имени и фамилии контактного лица, не входящих в перечень свойств, определенных в классе Entity. Тут также определен новый метод getEmployer(), для работы которого потребуется новая функция в классе DataManager. Мы вернемся к этой функции после рассмотрения класса Organization, код которого приведен ниже. Создайте файл class/Organization.php и введите в него следующий код.

```
<?php
require_once('class.Entity.php');
require_once('class.Individual.php');

class Organization extends Entity {

    public function __construct($userID) {
        parent::__construct($userID);

        $this->propertyTable['name'] = 'name1';

    }

    public function __toString() {
        return $this->name;
    }

    public function getEmployees() {
        return DataManager::getEmployees($this->id);
    }

    public function validate() {
        parent::validate();
        //проверка корректности данных для организации
    }
}

?>
```

Благодаря наследованию структура этого класса довольно проста. В нем объявлено свойство name, упрощающее задание единственного имени для организации (свойство sname2 для организации не используется).

Чтобы добавить функции getEmployer() и getEmployee() в класс DataManager, поместите следующий код в конец class/DataManager.php.

```
public static function getEmployer($individualID) {
    $sql = "SELECT \"organizationid\" FROM \"entityemployee\" "
           "WHERE \"individualid\" = $individualID";
    $res = pg_query(DataManager::__getConnection(), $sql);
    if(! ($res && pgsql_num_rows($res))) {
        die("Невозможно получить информацию об организации для сотрудника $individualID");
    }

    $row = pgsql_fetch_assoc($res);

    if($row) {
        return new Organization($row['organizationid']);
    } else {
        return null;
    }
}

public static function getEmployees($orgID) {
```

```

$sql = "SELECT \"individualid\" FROM \"entityemployee\" "
      "WHERE \"organizationid\" = $orgID";
$res = pgsql_query(DataManager:::_getConnection(), $sql);
if(! $res && pgsql_num_rows($res)) {
    die("Невозможно получить информацию о сотруднике для организации $orgID");
}

if(pgsql_num_rows($res)) {
    $objs = array();
    while($row = pgsql_fetch_assoc($res)) {
        $objs[] = new Individual($row['individualid']);
    }
    return $objs;
} else {
    return array();
}
}

```

Эти две функции предполагают наличие таблицы entityemployee, запрос для создания которой приведен ниже. Эта таблица связывает сотрудников с соответствующими организациями. К примеру, сотрудники некоторой компании могут иметь различные индивидуальные идентификаторы, но один общий идентификатор организации.

```

CREATE TABLE "entityemployee" (
    "individualid" int NOT NULL,
    "organizationid" int NOT NULL,
    CONSTRAINT "fk_entityemployee_individualid"
        FOREIGN KEY ("individualid") REFERENCES "entity"(entityid),
    CONSTRAINT "fk_entityemployee_organizationid"
        FOREIGN KEY ("organizationid") REFERENCES "entity"("entityid")
);

```

Последняя функция, необходимая для функционирования всей системы в целом, – это метод класса DataManager, предназначенный для отображения всех сущностей в базе данных. Эта функция называется getAllEntitiesAsObjects() и завершает список действий, выполняемых над объектами.

```

public static function getAllEntitiesAsObjects() {
    $sql = "SELECT \"entityid\", \"type\" FROM \"entities\";";
    $res = pgsql_query(DataManager:::_getConnection(), $sql);

    if(!$res) {
        die("Невозможно получить все сущности");
    }

    if(pgsql_num_rows($res)) {
        $objs = array();
        while($row = pgsql_fetch_assoc($res)) {
            if($row['type'] == 'I') {
                $objs[] = new Individual($row['entityid']);
            } elseif ($row['type'] == 'O') {
                $objs[] = new Organization($row['entityid']);
            } else {
                die("Неизвестный тип сущности {$row['type']}");
            }
        }
        return $objs;
    } else {
        return array();
    }
}

```

Класс DataManager позволяет обрабатывать всю контактную информацию в системе. Он проверяет значение поля ctype таблицы entity и определяет, к какому типу относится контакт — Individual или Organization, — затем инициализирует объект соответствующего типа и добавляет его в возвращаемый массив.

Использование системы

Теперь вы можете оценить реальную мощь объектно-ориентированного подхода. Следующий код будет подробно отображать все записи о контактах из базы данных. Этот код нужно поместить в файл test.php.

```
<?php
require_once('class.DataManager.php');

function println($data) {
    print $data . "<br>\n";
}

$arContacts = DataManager::getAllEntitiesAsObjects();
foreach($arContacts as $objEntity) {

    if(get_class($objEntity) == 'individual') {
        print "<h1>Частное лицо - {$objEntity->__toString()}</h1>";
    } else {
        print "<h1>Организация - {$objEntity->__toString()}</h1>";
    }

    if($objEntity->getNumberOfEmails()) {
        //У нас есть электронная почта! Выведите заголовок
        print "<h2>Электронная почта</h2>";

        for($x=0; $x < $objEntity->getNumberOfEmails(); $x++) {
            println($objEntity->emails($x)->__toString());
        }
    }

    if($objEntity->getNumberOfAddresses()) {
        //У нас есть адреса!
        print "<h2>Адреса</h2>";

        for($x=0; $x < $objEntity->getNumberOfAddresses(); $x++) {
            println($objEntity->addresses($x)->__toString());
        }
    }

    if($objEntity->getNumberOfPhoneNumbers()) {
        //У нас есть телефоны!
        print "<h2>Телефоны</h2>";

        for($x=0; $x < $objEntity->getNumberOfPhoneNumbers(); $x++) {
            println($objEntity->phonenumbers($x)->__toString());
        }
    }

    print "<hr>\n";
}
?>
```

Читателю в качестве упражнения предлагается самостоятельно ввести данные в таблицы; это поможет составить представление о том, как работает созданное приложение. Пример результатов приложения показан на рис 3.8. Для его получения запустите в браузере сценарий test.php.

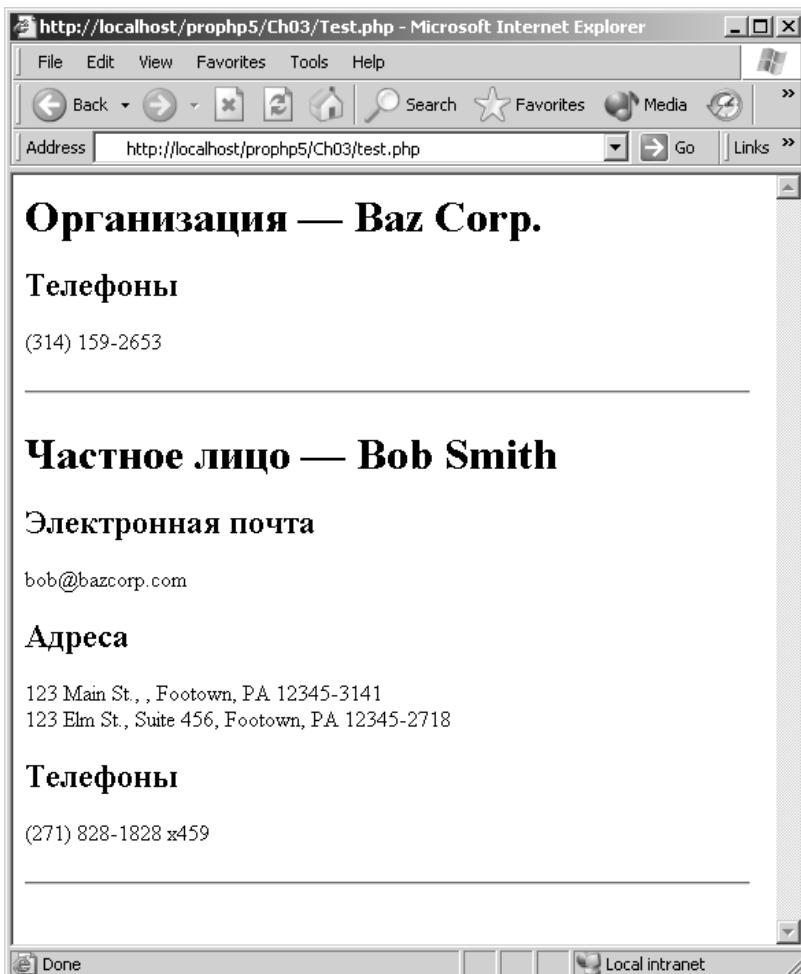


Рис. 3.8.

В 36 строках кода можно отобразить почти всю информацию о сущностях системы. Работодатель и работник здесь не показаны. Отображение информации о них снова оставим в качестве упражнения. Стока вызова метода `get_class()` в `test.php` может натолкнуть на мысль о том, с каким классом приходится иметь дело в данный момент, и определить, какую функцию вызывать: метод `getEmployer()` класса `Individual` либо метод `getEmployer()` класса `Organization`.

Резюме

Язык UML — это важнейший инструмент для моделирования сложных (и не очень) приложений. Разработанные должным образом диаграммы позволяют документировать серьезные системы гораздо проще и яснее, чем с помощью обычного текста. Использование диаграмм классов в процессе разработки программного обеспечения позволяет существенно облегчить проектирование классов и таблиц базы данных.

Использование преимуществ объектно-ориентированного подхода в PHP 5 помогает ускорить разработку приложения и создать библиотеку кода, которую легко использовать и расширять, а также сокращает общий объем кода, требуемый для реализации требований приложения.

Разделяя архитектуру приложения на уровень бизнес-логики, включающий класс `Individual`, и уровень работы с данными, представленный классом `DataManager`, вы упрощаете изменение источников данных, структуры таблиц или запросов без лишнего влияния на остальные части приложения. Объекты, отвечающие за реализацию бизнес-логики, не должны вмешиваться в механизм доступа к данным. Это может привести к путанице в алгоритме реализации и сделать его сложным для понимания.

4

Шаблоны проектирования

Из предыдущей главы вы узнали, что одни объекты могут наследовать свойства других, а также поняли, что объекты могут содержать ссылки на другие объекты. Например, объект `DrumSet` включает в себя объект `Drum`. В целом метод создания объектов путем включения других объектов называется *композицией* (*composition*).

Наследование и композиция объектов являются мощными средствами разработки объектно-ориентированного программного обеспечения и позволяют разрабатывать широкий круг решений. Конечно, слишком большой выбор не облегчает процесс принятия решений. Как же разработать приложение, обеспечив простоту его поддержки и легкость расширения? Как написать такой компонент, который другие члены команды разработчиков могут использовать через простой интерфейс? При разработке программы можно решать возникающие проблемы самостоятельно, основываясь на собственном опыте, используя свой интеллект, удачу, потребляя большое количество минеральной воды или любых других напитков.

Грамотно разработанный код зачастую повторно используется для решения других задач. Например, у вас может быть стандартный сценарий для установки соединения с базой данных. Шаблоны проектирования не сводятся к простому использованию старого кода. Они являются более абстрактными и обобщенными. Один и тот же шаблон проектирования можно использовать в совершенно разных приложениях. Шаблоны проектирования обеспечивают повторное использование идей. Изучив шаблон, вы поймете, где его можно применять, и сможете использовать при необходимости.

Шаблон проектирования — это определенный способ решения конкретной проблемы. Например, он может описывать способ структурирования объектов и их наборов, их взаимодействие и “общение” с другими объектами. Каждый шаблон включает описательное имя, например `Observer` или `Observable`, и конкретное проектное решение, которое можно отобразить на диаграмме классов.

Вначале шаблоны могут показаться не совсем понятными, не беспокойтесь — в этой главе вы познакомитесь с пятью различными шаблонами, которые построены на основе кода из нескольких предыдущих глав.

Шаблон Composite

У читателя на самом деле уже есть небольшой опыт работы с одним шаблоном (шаблоном Composite), полученный при изучении интерфейса `Instrument` в главе 2 “Унифицированный язык моделирования UML”. Еще раз посмотрим на диаграмму классов, показанную на рис. 4.1.

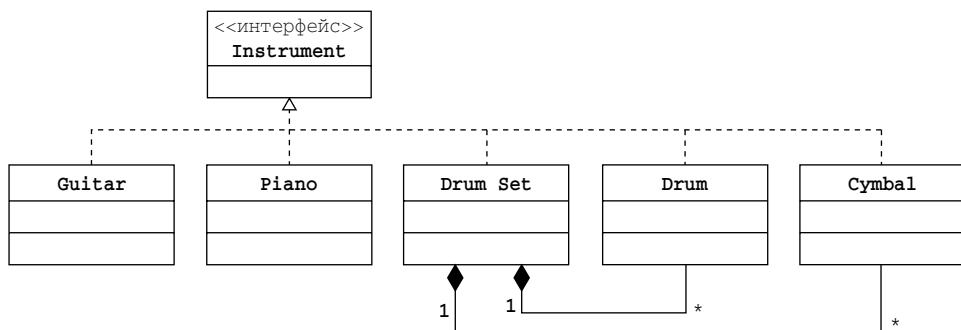


Рис. 4.1.

На рис. 4.1 показано, что объект `DrumSet` может состоять из объектов `Drum` и `Cymbal`. В свою очередь объекты `Drum` и `Cymbal` можно рассматривать как дочерние по отношению к объекту `DrumSet`. Объект `DrumSet` может содержать произвольное количество объектов `Drum` и `Cymbal`. Отметим, что `DrumSet` и его дочерние объекты принадлежат одному и тому же типу, а именно, `Instrument`. При взаимодействии с объектом `DrumSet` клиентский объект использует тот же интерфейс, что и при взаимодействии с дочерними объектами. Такие взаимоотношения определяются шаблоном Composite.

На рис. 4.2 показан общий случай композиции на основе этого шаблона.

Шаблон композиции предполагает наличие двух частей: абстрактного класса `Component` и класса `Composite`, который является конкретной реализацией класса `Component`. Все объекты класса `Composite` создаются на основе абстрактного класса `Component`. Любой объект `Component` может содержать другие объекты этого же типа (`Component`). Такой компонент можно рассматривать как композит (объект класса `Composite`). Объект `Component` без наследников можно считать пустым объектом типа `Composite`.

В некоторых реализациях шаблона различают объекты `Composite` и `Leaf`.

Объектами `Leaf` (листьями) являются те объекты `Component`, которые не имеют наследников. В данном примере все объекты `Component` могут выступать в роли объектов `Composite`.

В связи с этим необходимо изменить проектное решение и преобразовать интерфейс `Instrument` в абстрактный класс. Интерфейсы и абстрактные классы схожи в том, что их нельзя применять для непосредственного создания объектов. Ключевое различие между ними состоит в том, что абстрактный класс может содержать полностью реализованные методы, в то время как интерфейс содержит только их объявление. Используйте абстрактные классы, если хотите обеспечить одинаковые методы для всех подклассов и сохранить для них некоторую общую функциональность. Интерфейсы следует применять в том случае, если реализации всех методов для разных подклассов должны отличаться. На рис. 4.3 приведена новая диаграмма классов. Отметим, что на диаграмме абстрактный класс выделен курсивом.

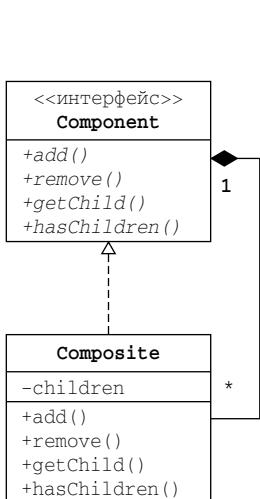


Рис. 4.2.

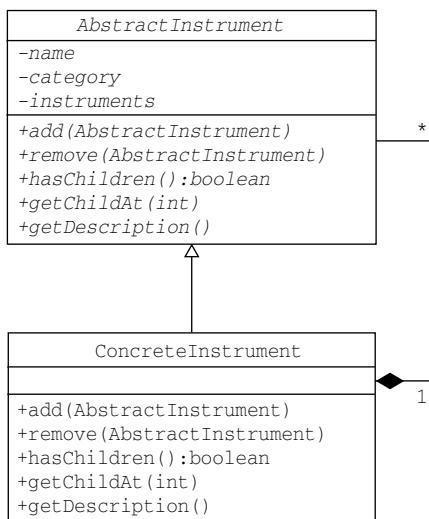


Рис. 4.3.

Линия ассоциации указывает на то, что каждый конкретный экземпляр класса `AbstractInstrument` может содержать любое количество (или не содержать вовсе) других объектов `AbstractInstrument`.

Реализация

Следующий код PHP иллюстрирует применение шаблона Composite.

```

<html>
<body>
<head>
<style>
body {font : 12px verdana; font-weight:bold}
td {font : 11px verdana;}
</style>
</head>
<?php

abstract class AbstractInstrument {

    private $name;
    private $category;
    private $instruments = array();

    public function add(AbstractInstrument & $instrument) {
        array_push($this->instruments, $instrument);
    }

    public function remove(AbstractInstrument & $instrument) {
        array_pop($this->instruments);
    }

    public function hasChildren() {
        return (bool)(count($this->instruments) > 0);
    }

    public function getDescription() {
        $description = $this->name . " (" . $this->category . ")";
        if ($this->instruments) {
            $description .= " (";
            foreach ($this->instruments as $instrument) {
                $description .= $instrument->getDescription() . ", ";
            }
            $description = substr($description, 0, -2);
            $description .= ")";
        }
        return $description;
    }
}
  
```

```
}

public function getChild($i) {
    return $instruments[$i];
}

public function getDescription() {
    echo "- один " . $this->getName();
    if ($this->hasChildren()) {
        echo " включает:<br>";
        foreach($this->instruments as $instrument) {
            echo "<table cellspacing=5
border=0><tr><td>&ampnbsp&ampnbsp&ampnbsp</td><td>-";
            $instrument->getDescription();
            echo "</td></tr></table>";
        }
    }
}

public function setName($name) {
    $this->name = $name;
}

public function getName() {
    return $this->name;
}

public function setCategory($category) {
    $this->category = $category;
}

public function getCategory() {
    return $this->category;
}

class Guitar extends AbstractInstrument {
    function __construct($name) {
        parent::setName($name);
        parent::setCategory("гитары");
    }
}

class DrumSet extends AbstractInstrument {
    function __construct($name) {
        parent::setName($name);
        parent::setCategory("барабаны");
    }
}

class SnareDrum extends AbstractInstrument {
    function __construct($name) {
        parent::setName($name);
        parent::setCategory("барабаны высокого звучания");
    }
}

class BaseDrum extends AbstractInstrument {
    function __construct($name) {
        parent::setName($name);
        parent::setCategory("барабаны низкого звучания");
    }
}
```

```

class Cymbal extends AbstractInstrument {
    function __construct($name) {
        parent::setName($name);
        parent::setCategory("тарелки");
    }
}

$drums = new DrumSet("ударные");
$drums->add(new SnareDrum("барабан высокого звучания "));
$drums->add(new BaseDrum("большой барабан низкого звучания"));

$cymbals = new Cymbal("набор тарелок");
$cymbals->add(new Cymbal("маленькая тарелка"));
$cymbals->add(new Cymbal("большая высокая шляпа"));
$drums->add($cymbals);

$guitar = new Guitar("гибсон ле поль ");

echo "Список инструментов: <p>";
$drums->getDescription();
$guitar->getDescription();

?>

</body>
</html>

```

Обратите внимание, что каждый конкретный инструмент, например DrumSet (набор барабанов) или Guitar (гитара), наследует свойства класса AbstractInstrument. Также отметим, что подклассы наследуют реализацию методов, определенных в абстрактном классе. В реализации метода `getDescription()` текущий инструмент проверяется на наличие наследников. Если он содержит дочерние объекты, то метод вызывается рекурсивно до тех пор, пока не будет пройдено все дерево.

Исторически (в PHP 4) символ & указывал на объект (в данном случае типа `AbstractInstrument`), передаваемый по ссылке.

```
public function add (AbstractInstrument & $instrument) {
```

Это важно, так как в противном случае в функции будет использована новая локальная копия. В PHP 5 объекты автоматически передаются по ссылке, поэтому символ & использовать не обязательно. Для рассматриваемой задачи это лучше, так как в разрабатываемой системе необходимо работать с самим объектом, а не с одной из его копий. На рис. 4.4 приведена *диаграмма объектов* (object diagram) для предыдущего кода, на которой показаны инструменты и их дочерние объекты. Диаграмма объектов похожа на диаграмму классов, только на ней показаны экземпляры объектов, имена которых выделены подчеркиванием. На этой диаграмме также представлены отношения между объектами в системе в определенный момент времени.

В этом примере классы конкретных инструментов практически не отличаются, разве что в классе `Guitar` переопределен метод `getDescription()`. Предположим, необходимо добавить методы связи с производителем для получения информации со склада. Если каждый производитель предпочитает свой способ общения с системой, можно определить метод `contact()`. Для этого сначала необходимо добавить абстрактный метод в класс `AbstractInstrument`, а затем реализовать его в каждом подклассе. Добавив абстрактный метод, вы не забудете реализовать его в подклассах, иначе будет выведено сообщение об ошибке.

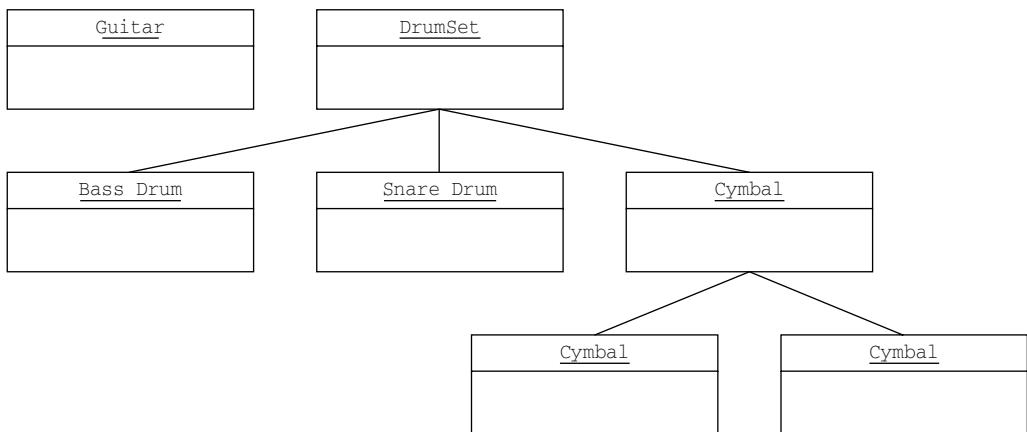


Рис. 4.4.

Если инструментам точно не потребуется никакой дополнительной функциональности, можно создать класс `GenericInstrument`, унаследовав его от класса `AbstractInstrument`. Тогда при создании нового экземпляра тарелок потребуется ввести код.

```
$cymbals = new GenericInstrument ("набор тарелок");
```

Ключевым моментом при таком подходе является то, что любой разработчик, получив доступ к интерфейсу `Instrument` (будь то вы, другой программист или другая часть приложения), не обязан знать, как он реализован. Вызов метода `getDescription()` возвращает описание структуры дерева `Instrument`, независимо от того, включает ли данный инструмент дочерние объекты, есть ли у них свои наследники или наследники наследников, и т.д. При вызове метода не нужно знать иерархию наследования.

Хотя все объекты инструментов имеют одинаковый интерфейс, они не обязаны единообразно реагировать на события. Следовательно, такие объекты можно рассматривать как полиморфные. Принцип полиморфизма описывается следующими свойствами: *одинаковый интерфейс и разная реализация* или, обобщая, *одинаковый интерфейс и разное поведение*.

Обсуждение

Описанный шаблон композиции очень гибок. Он не содержит ограничений, при которых инструмент может иметь дочерние объекты. Рассмотрим следующий код.

```
$cymbals->add(new Cymbal ("большая высокая шляпа"));
$drums->add($cymbals);
```

```
$cymbals->add($drums);
```

```
$guitar = new Guitar("гибсон ле поль");
```

Добавление барабанов к тарелкам, которые сами относятся к барабанам, создает циклическую связь. При вызове функции `getDescription()` подобный код приведет к сбою Web-сервера. Следует избегать подобных ошибок. В первую очередь при вызове метода `add()` нужно выполнить проверку, содержит ли добавляемый инструмент

ссылку на объект, вызывающий данный метод. Если это так, необходимо вывести сообщение об ошибке.

Вы можете поэкспериментировать с группами инструментов, запрещая добавление к ним определенных объектов. Это можно сделать в определении каждого класса, перекрывая метод `add()`. Другой способ — создать новый абстрактный класс, от которого унаследуют свойства все отдельные инструменты. Этот абстрактный класс может иметь метод `add()`, который либо ничего не делает, либо выводит сообщение об ошибке. Это больше соответствует описанному выше шаблону `Composite`, согласно которому класс `Component` наследуют два класса: класс `Leaf`, который не может иметь наследников, и класс `Composite`, который может. Диаграмма классов для данной ситуации приведена на рис. 4.5.

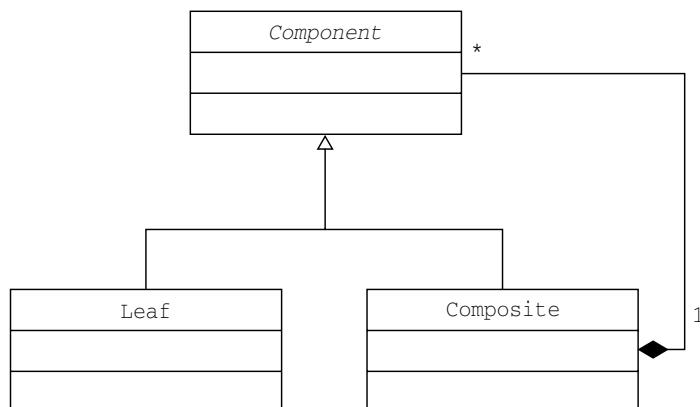


Рис. 4.5.

Существует еще один выход — можно запретить добавлять определенные инструменты в наборы или композитные объекты. Можно запретить добавлять объект класса `Guitar` (гитара) в набор барабанов `DrumSet`. Тогда допустимые типы можно определить с помощью атрибута `category`. Реализация этих свойств — хорошее упражнение для самостоятельного выполнения.

Шаблон Observer

Зачастую в приложении используются данные, которые время от времени изменяются. Предположим, для отображения этих данных используются некоторые компоненты графического интерфейса пользователя (GUI), которые необходимо обновлять при изменении данных. Как справиться с такой задачей? Одним из решений является передача вновь измененных данных методу компонента GUI, чтобы он мог обновить информацию на экране. Однако при таком подходе возникает проблема — не забывать делать это при каждом обновлении данных. Как же обеспечить автоматическое обновление компонента GUI, если частота изменения данных не известна?

Шаблон `Observer` (наблюдатель) позволяет решить эту проблему с использованием двух интерфейсов — `Observer` (наблюдатель) и `Observable` (наблюдааемый). Из самих названий очевидно, что `Observer` следит за изменениями объекта, реализующего интерфейс `Observable`.

В соответствии со здравым смыслом, шаблон *Observer* иногда называют *Listener* (слушатель), но в этой главе мы будем придерживаться первого названия.

В самой базовой реализации интерфейс *Observable* может добавлять различные интерфейсы *Observer*, обеспечивая их извещение при изменении своего состояния. А интерфейс *Observer* должен реагировать на эти изменения. В нашем примере данные являются наблюдаемыми *Observable*, а компоненты GUI — наблюдателями *Observer*. Изменение данных автоматически отразится в любом компоненте GUI, который зарегистрирован как наблюдатель *Observer*. На рис. 4.6 проиллюстрирован шаблон наблюдения *Observer*.

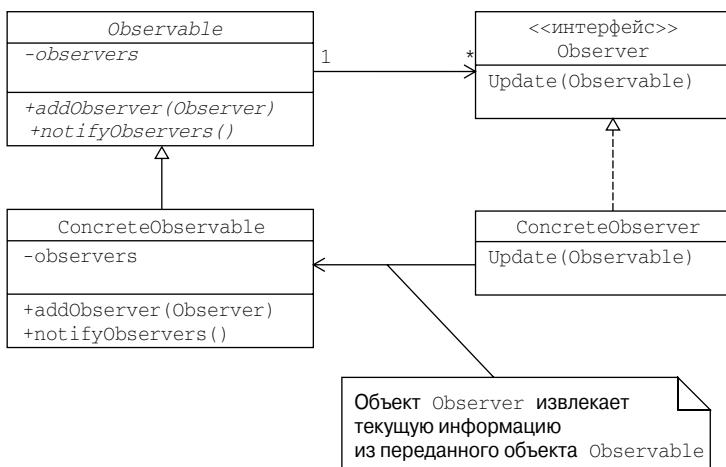


Рис. 4.6.

Элементы управления

Продолжая рассматривать предыдущий пример, используем шаблон *Observer* для вывода информации о ценах инструментов с помощью графических элементов на Web-странице. В первую очередь нужно определить некоторые простые графические компоненты. Пусть это будут базовые табличные структуры HTML, функциональность которых будет задаваться в соответствующих объектах. Такие компоненты обычно называют общим термином — *управляющие элементы* (widget).

Управляющие элементы должны отображать все ту же информацию об инструментах — в нашем примере это название инструмента и цена.

Проектирование управляющих элементов

У графического управляющего элемента всего две “обязанности”. Он должен отображать документ HTML на Web-странице и обновлять выводимые данные. Из рис. 4.6 видно, что метод *update()* определен в интерфейсе *Observer*.

Каждый управляющий элемент представляет собой *Observer*. Наблюдаемой является информация о названии и цене инструмента. Источник данных должен реализовывать интерфейс *Observable*.

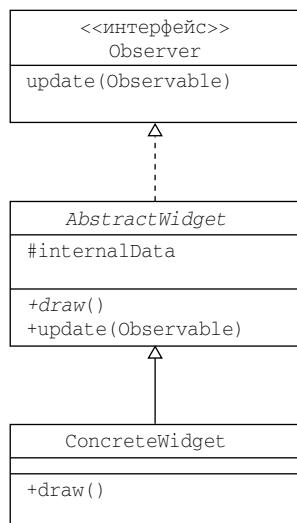


Рис. 4.7.

Таким образом, все управляющие элементы должны реализовывать интерфейс `Observer`. При этом каждый из них должен наследовать абстрактный класс, который определяет их общую функциональность. Это пример использования интерфейса вместе с абстрактным классом. Так как метод `update()` является общим для всех управляющих элементов, он может быть реализован в абстрактном классе. Соответствующая диаграмма классов представлена на рис. 4.7.

Сейчас все конкретные реализации управляющих элементов наследуют класс `AbstractWidget`, который, в свою очередь, реализует интерфейс `Observer`. Обратите внимание, что метод `update()` для класса `AbstractWidget` на диаграмме не выделен курсивом. Это означает, что метод уже реализован. Символ `#` свидетельствует о том, что свойство `internalData` является защищенным, т.е. подклассы `AbstractWidget` имеют доступ к нему. Если бы оно было определено в закрытой области, подклассы не могли бы к нему обращаться.

Рассмотрим следующий код класса `Widget`, который расположен в файле `abstract_widget.php`.

```

<?php

interface Observer {
    public function update();
}

abstract class Widget implements Observer {

    protected $internalData = array();

    abstract public function draw();

    public function update(Observable $subject) {
        $this->internalData = $subject->getData();
    }
}

class BasicWidget extends Widget {

    function __construct() {
    }

    public function draw() {
        $html = "<table border=1 width=130>";
        $html .= "<tr><td colspan=3 bgcolor=#cccccc>
                <b>Данные об инструментах</b></td></tr>";

        $numRecords = count($this->internalData[0]);
        for($i = 0; $i < $numRecords; $i++) {
            $instms = $this->internalData[0];
            $prices = $this->internalData[1];
            $years = $this->internalData[2];
            $html .= " <tr><td>$instms[$i]</td><td> $prices[$i]</td>
  
```

```

        <td>$years[$i]</td></tr>";
    }
$html .= "</table><br>";
echo $html;
}

class FancyWidget extends Widget {

function __construct() {

public function draw() {
$html =
"<table border=0 cellpadding=5 width=270 bgcolor=#6699BB>
<tr><td colspan=3 bgcolor=#cccccc>
<b><span class=blue>Текущие цены</span><b>
</td></tr>
<tr><td><b>инструмент</b></td>
<td><b>цена</b></td><td><b>дата выпуска</b>
</td></tr>";

$numRecords = count($this->internalData[0]);
for($i = 0; $i < $numRecords; $i++) {
$instms = $this->internalData[0];
$prices = $this->internalData[1];
$years = $this->internalData[2];

$html .=
"<tr><td>$instms[$i]</td><td>
$prices[$i]</td><td>$years[$i]
</td></tr>";
}
$html .= "</table><br>";
echo $html;
}
}

?>
```

В этом коде приведены две конкретные реализации класса `Widget` — это классы `FancyWidget` и `BasicWidget`. В обоих реализован метод `draw()`, как того требует абстрактный базовый класс, наследниками которого они являются. Однако реализации этого метода в каждом из классов различаются. Кроме того, оба класса наследуют от родительского класса метод `update()`.

Вам, наверное, интересно, какие преимущества дает использование интерфейса `Observer`, если можно просто добавить в подклассы один новый метод. В ранних версиях PHP нельзя явно гарантировать принадлежность параметра метода к определенному типу. Новинкой PHP 5 является возможность *задания типов*. Это позволяет гарантировать, что в качестве аргумента передается объект нужного типа. В следующем объявлении функции `ссылка`, передаваемая в качестве аргумента, должна принадлежать типу `Observer`, иначе будет выведено сообщение о ошибке.

```
public function addObserver (Observer $observer) {
```

Если конкретный метод в качестве аргумента получает объект типа `Observer`, как в данном примере метод `addObserver()` класса `Observable`, ему можно смело передавать объект класса `Widget`, так как все объекты `Widget` относятся к типу `Observer`. Все

управляющие элементы унаследованы от класса `AbstractWidget`, который реализует интерфейс `Observer`, следовательно, объекты класса `Widget` принадлежат к этому типу.

Возникает другой вопрос: почему в качестве параметра метода `addObserver()` не использовать просто объект класса `Widget`? В этом случае можно обойтись без интерфейса `Observer`. Однако, предположим, вам потребуется создать другой тип наблюдателя `Observer`, который не будет являться классом `Widget`. Тогда указание типа аргумента предотвратит передачу других типов объектов в качестве параметров метода `addObserver()`.

Источник данных

Объект `DataSource` содержит имя, цену и дату выпуска для группы музыкальных инструментов. Он также является объектом `Observable`. Основными методами интерфейса `Observable` являются `addObserver()` и `notifyObservers()`. Каждый объект `Observer` (в данном примере каждый объект `Widget`), который должен наблюдать за объектом `DataSource`, может быть зарегистрирован в списке наблюдателей с помощью метода `addObserver()`. Рассмотрим следующий код, расположенный в файле `observable.php`.

```
<?php
abstract class Observable {
    private $observers = array();

    public function addObserver(Observer & $observer) {
        array_push($this->observers, $observer);
    }

    public function notifyObservers() {
        for ($i = 0; $i < count($this->observers); $i++) {
            $widget = $this->observers[$i];
            $widget->update($this);
        }
    }
}

class DataSource extends Observable {

    private $names;
    private $prices;
    private $years;

    function __construct() {
        $this->names = array();
        $this->prices = array();
        $this->years = array();
    }

    public function addRecord($name, $price, $year) {
        array_push($this->names, $name);
        array_push($this->prices, $price);
        array_push($this->years, $year);
        $this->notifyObservers();
    }

    public function getData() {
        return array($this->names, $this->prices, $this->years);
    }
}
?>
```

Метод `addRecord()` дает возможность добавить новый инструмент во внутреннее хранилище `DataSource`. Отметим, что метод `addRecord()` выполняет еще одно действие.

```
$this->notifyObservers();
```

Как только внутренняя информация объекта `DataSource` изменяется, он сообщает об этом всем наблюдателям. Наблюдатели добавляются с помощью упомянутого ранее метода `addObserver()`. После добавления наблюдателя он сохраняется во внутреннем массиве `$observers`. При вызове метода `notifyObservers()` для каждого сохраненного в массиве `$observers` объекта `Observer` вызывается метод `update()`.

Единственным параметром метода `update()` является копия самого объекта `Observable`.

```
$widget->update($this);
```

Это позволяет объекту `Widget` (который реализует интерфейс `Observer`) получить копию текущего состояния объекта `DataSource` (реализующего интерфейс `Observable`). Отметим, что объекту `Widget` передается только значение — копия объекта `DataSource`, — а не реальная ссылка на объект. Таким образом, внутренняя информация `DataSource` не подлежит совместному использованию.

Связь наблюдателя и наблюдаемого

Теперь, когда вся тяжелая работа завершена, осталось пожинать плоды. Читатель получил в свое распоряжение простую в использовании и гибкую систему для связи `Observer` `Widget` с `Observable` `DataSource`. Рассмотрим следующий пример кода, находящегося в файле `widget.php`.

```
<?php
require_once("observable.php");
require_once("abstract_widget.php");

$dat = new DataSource();
$widgetA = new BasicWidget();
$widgetB = new FancyWidget();

$dat->addObserver($widgetA);
$dat->addObserver($widgetB);

$dat->addRecord("барабан", "$12.95", 1955);
$dat->addRecord("гитара", "$13.95", 2003);
$dat->addRecord("банджо", "$100.95", 1945);
$dat->addRecord("фортепиано", "$120.95", 1999);

$widgetA->draw();
$widgetB->draw();
?>
```

Все, что вам нужно сделать — это создать объекты `DataSource` и `Widget`, а затем добавить объекты `Widget` в список наблюдателей для объекта `DataSource`. Теперь можете внести любое количество записей в `DataSource`, не заботясь о обновлении управляющих элементов, — оно произойдет автоматически. В итоге при вызове метода `draw()` для любого из управляющих элементов он будет отображен на экране с корректной информацией.

Если вы разрабатываете настольное приложение, то можете включить функцию `redraw()` в метод `update()` интерфейса `Observer`, тогда вам не потребуется явно вызывать метод `draw()` — компонент автоматически будет перерисовываться при обновлении объекта `DataSource`. В серверных Web-приложениях это невозможно. Компонент перерисовывается только при обновлении всей страницы.

Еще одним важным свойством этого шаблона является возможность определения нескольких объектов `DataSource`. Вы всего лишь сообщаете объекту `DataSource`, какие управляющие элементы являются его “наблюдателями”, и можете повторно использовать одни и те же управляющие элементы для разных объектов `DataSource`. На рис. 4.8 показан результат выполнения приведенного выше кода.

Данные об инструментах		
барабан	\$12.95	1955
гитара	\$13.95	2003
банджо	\$100.95	1945
фортепиано	\$120.95	1999

Текущие цены		
инструмент	цена	дата выпуска
барабан	\$12.95	1955
гитара	\$13.95	2003
банджо	\$100.95	1945
фортепиано	\$120.95	1999

Рис. 4.8.

Обсуждение

Шаблон наблюдателя полезен в том случае, если данные из одного источника нужно выводить в различных представлениях. В этой главе был использован простой объект `DataSource`, но можно создавать более сложные источники данных, например, получающие информацию из базы данных или файла XML. Можно создать класс `Widget`, который будет представлен как объект `DataSource`, полученный из XML-файла или базы данных. Если объекты `DataSource` имеют единый интерфейс, способ извлечения данных не играет роли.

Объект класса `Widget` в данном примере выводит таблицу в три колонки, но способ отображения может быть куда более гибким. В качестве упражнения можно попробовать написать код для отображения произвольного количества столбцов. Также можно попробовать использовать небольшие вспомогательные объекты вместо массивов для передачи информации между классами `Observable` и `Observer`.

Шаблон Decorator

Созданные в предыдущем разделе два конкретных управляющих элемента выполнены в разных стилях, поэтому выглядят аляповато. Если вам нужно разработать единый новый стиль для отображения управляющих элементов, можно создать подкласс

Widget и реализовать в нем метод draw() для вывода кода HTML. Допустим, ко всем существующим управляющим элементам требуется добавить некоторый элемент, например рамку. Для этого можно в каждый метод draw() добавить соответствующий код HTML, но внешний вид рамки будет жестко задан, и каждый управляющий элемент должен будет ее отображать. Можно создать новый набор подклассов существующих конкретных управляющих элементов и реализовать рамку в методе draw(). Если у вас только два управляющих элемента, как в примере для шаблона Observer, это еще приемлемо — в результате вы получите четыре управляющих элемента. Но если изначально существуют четыре управляющих элемента, это будет уже не удобно, поскольку в результате получится слишком много элементов.

Существует другой путь решения подобных проблем, который не требует создания новых подклассов для каждого управляющего элемента, к которому нужно добавить рамку. Использование шаблона Decorator (декоратор) позволит добавлять новые свойства или функциональность существующим объектам без использования наследования. На рис. 4.9 приведена диаграмма классов для шаблона Decorator.

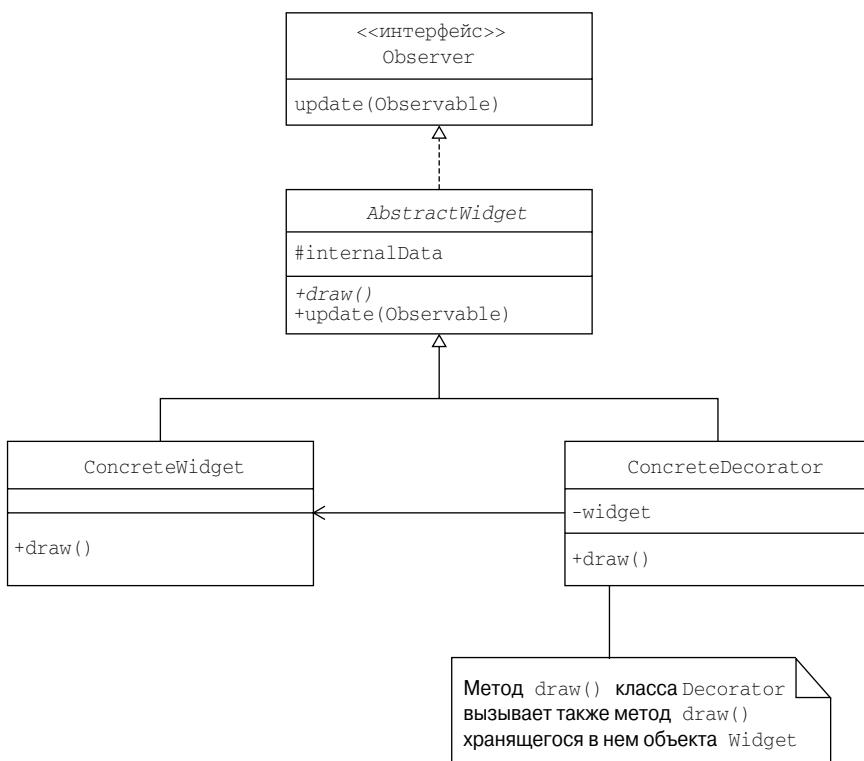


Рис. 4.9.

Из этой диаграммы классов видно, что класс ConcreteDecorator тоже относится к типу Widget. Это хорошо, так как не требует изменения способа доступа к объекту Widget в зависимости от того, декорирован ли он. Рассмотрим код следующего класса BorderDecorator, который хранится в файле `border_decorator.php` и рисует рамку для управляющего элемента.

```

<?php

require_once("abstract_widget.php");

class BorderDecorator extends Widget {

    private $widget;

    function __construct(Widget & $widget) {
        $this->widget = $widget;
    }

    public function draw() {

        $this->widget->update($this->getSubject());

        echo "<table border=0 cellpadding=1 bgcolor=#3366ff>";
        echo "<tr bgcolor=#ffffff><td>";
        $this->widget->draw();
        echo "</td></tr></table>";
    }
}
?>

```

В этом примере все классы и интерфейсы мы поместили в отдельные файлы, используя для их импортирования функцию `require_once()`. Метод `draw()` генерирует некоторый код HTML для вывода таблицы. Отметим, что внутри рамки выводимой таблицы метод `draw()` вызывает метод `draw()` объекта `Widget`, который был передан конструктору. Таким образом объект `Widget` получает дополнительное “украшение”, в нашем случае — рамку. Приведем пример использования шаблона `Decorator`.

```

$widgetA = new BasicWidget();
$widgetA = new BorderDecorator($widgetA);

```

После создания класса согласно шаблону `Decorator` ему остается только передать объект `Widget`. Теперь “декорированный” объект `Widget` можно использовать таким же образом, как и обычный. На рис. 4.10 представлен объект `BasicWidget`, отображенный с применением объекта `BorderDecorator` и без него.

Данные об инструментах		
барабан	\$12.95	1955
гитара	\$13.95	2003
банджо	\$100.95	1945
фортепиано	\$120.95	1999

Данные об инструментах		
барабан	\$12.95	1955
гитара	\$13.95	2003
банджо	\$100.95	1945
фортепиано	\$120.95	1999

Рис. 4.10.

Реализация

Для применения шаблона Decorator нужно внести несколько небольших, но важных изменений в объекты Widget. Во-первых, защищенный массив InternalData в абстрактном классе Widget нужно заменить закрытой ссылкой на объект DataSource, хранимый в переменной \$subject.

```
abstract class Widget implements Observer {
    private $subject;
    abstract public function draw();
    public function update(Observable $subject) {
        $this->subject = $subject;
    }
    public function getSubject() {
        return $this->subject;
    }
}
```

Затем при реализации методов draw() в каждом конкретном классе Widget объект Widget должен получать доступ к данным через унаследованный метод getSubject().

```
public function draw() {
    $data = $this->getSubject()->getData();
    $numRecords = count($data[0]);
    $html = "<table border=1 width=130>";
    $html .= "<tr><td colspan=3 bgcolor=#cccccc>
        <b>Данные об инструментах<b></td></tr>";
    for($i = 0; $i < $numRecords; $i++) {
        $instms = $data[0];
        $prices = $data[1];
        $years = $data[2];
        $html .= "<tr><td>$instms[$i]</td><td> $prices[$i]</td>
            <td>$years[$i]</td></tr>";
    }
    $html .= "</table><br>";
    echo $html;
}
```

И наконец, так как класс-декоратор на самом деле является наблюдателем для объекта DataSource, при вызове метода draw() класса-декоратора BorderDecorator он должен обновлять информацию о DataSource в своем объекте Widget. Это позволит объекту Widget получить доступ к данным.

```
public function draw() {
    $this->widget->update($this->getSubject());
    echo "<table border=0 cellpadding=1 bgcolor=#3366ff>";
    echo "<tr bgcolor=#ffffff><td>";
    $this->widget->draw();
    echo "</td></tr></table>";
}
```

Использование шаблона Decorator

Завершив разработку классов `Decorator`, вы получите очень гибкий и мощный шаблон. Его не только легко использовать. Разные классы-декораторы можно объединять для получения нескольких эффектов декорирования одного управляющего элемента. Их даже можно применять в разном порядке. Рассмотрим следующий класс-декоратор, сохранив его в файле `closebox_decorator.php`.

```
<?php
require_once ("abstract_widget.php");

class CloseBoxDecorator extends Widget {
    private $widget;

    function __construct(Widget $widget) {
        $this->widget = $widget;
    }

    public function draw() {
        $this->widget->update($this->getSubject());

        print "<table border=0 cellspacing=1 bgcolor=#666666>";
        print "<tr bgcolor=#666666>";
        print "<td align=right>";
        print "    <table width=10 height=10 bgcolor="cccccc">";
        print "        <tr><td><b>x</b></td></tr>";
        print "    </table>";
        print "</td>";
        print "</tr>";
        print "<tr bgcolor=#ffffff>";
        print "<td>";

        $this->widget->draw();

        print "</td>";
        print "</tr>";
        print "</table>";
    }
}
```

Этот класс-декоратор `CloseBoxDecorator` выводит простое меню с клавишей закрытия над ним. Объединить два объекта `Widget` и два класса-декоратора очень просто. Сохраним следующий код в файле `decorator.php`.

```
<?php
require_once ("abstract_widget.php");
require_once ("closebox_decorator.php");
require_once ("border_decorator.php");
require_once ("observable.php");

$dat = new DataSource();
$widgetA = new BasicWidget();
$widgetB = new FancyWidget();

$widgetB = new BorderDecorator($widgetB);
$widgetB = new CloseBoxDecorator($widgetB);

$widgetA = new CloseBoxDecorator($widgetA);
```

```
$widgetA = new BorderDecorator($widgetA);

$dat->addObserver($widgetA);
$dat->addObserver($widgetB);

$dat->addRecord("барабан", "$12.95", 1955);
$dat->addRecord("гитара", "$13.95", 2003);
$dat->addRecord("банджо", "$100.95", 1945);
$dat->addRecord("фортепиано", "$120.95", 1999);

$widgetB->draw();
echo "<br>";
$widgetA->draw();
?>
```

Отметим, что к первому объекту *Widget* классы с суффиксом *Decorator* применяются в прямом порядке, а ко второму — в обратном. На рис. 4.11 показан результат выполнения этого кода.

инструмент	цена	дата выпуска
барабан	\$12.95	1955
гитара	\$13.95	2003
банджо	\$100.95	1945
фортепиано	\$120.95	1999

Данные об инструментах		
барабан	\$12.95	1955
гитара	\$13.95	2003
банджо	\$100.95	1945
фортепиано	\$120.95	1999

Рис. 4.11.

Обсуждение

В этом разделе в классах с суффиксом *Decorator* жестко задана цветовая гамма. Но оформление можно сделать более гибким. Класс *BorderDecorator* можно модифицировать для обеспечения различных вариаций цвета и толщины линии. Используя DHTML, можно добиться, чтобы объект *CloseBoxDecorator* сам исчезал при щелчке на соответствующей кнопке.

Следует помнить, что шаблон *Decorator* применяется не только к графическим элементам. В случае классов *Widget* шаблон позволяет генерировать дополнительный код HTML для улучшения вида управляющих элементов. Однако шаблон *Decorator* можно использовать для внесения дополнительной информации в некоторую структуру данных. Например, с помощью шаблона *Decorator* некоторый фрагмент кода XML можно включить в более развитую структуру XML.

Чтобы проверить уровень своего понимания объектно-ориентированного программирования, читатель может изменить способ передачи *DataSource* объекту

Widget из объекта типа Decorator. Это можно сделать несколькими способами. Как объект Widget может получить копию объекта DataSource без передачи ее из объекта типа Decorator с помощью метода draw()?

Шаблон Facade

Чтобы разобраться с шаблоном Facade (фасад), рассмотрим диаграмму компонентов системы до и после его использования.

На рис. 4.12 представлено взаимодействие клиентской части Web-приложения с серверной частью. С правой стороны клиентская страница обращается к различным объектам серверной части приложения. На диаграмме слева страница обращается только к внешнему объекту (фасаду), который в свою очередь делегирует обязанности внутренним объектам.

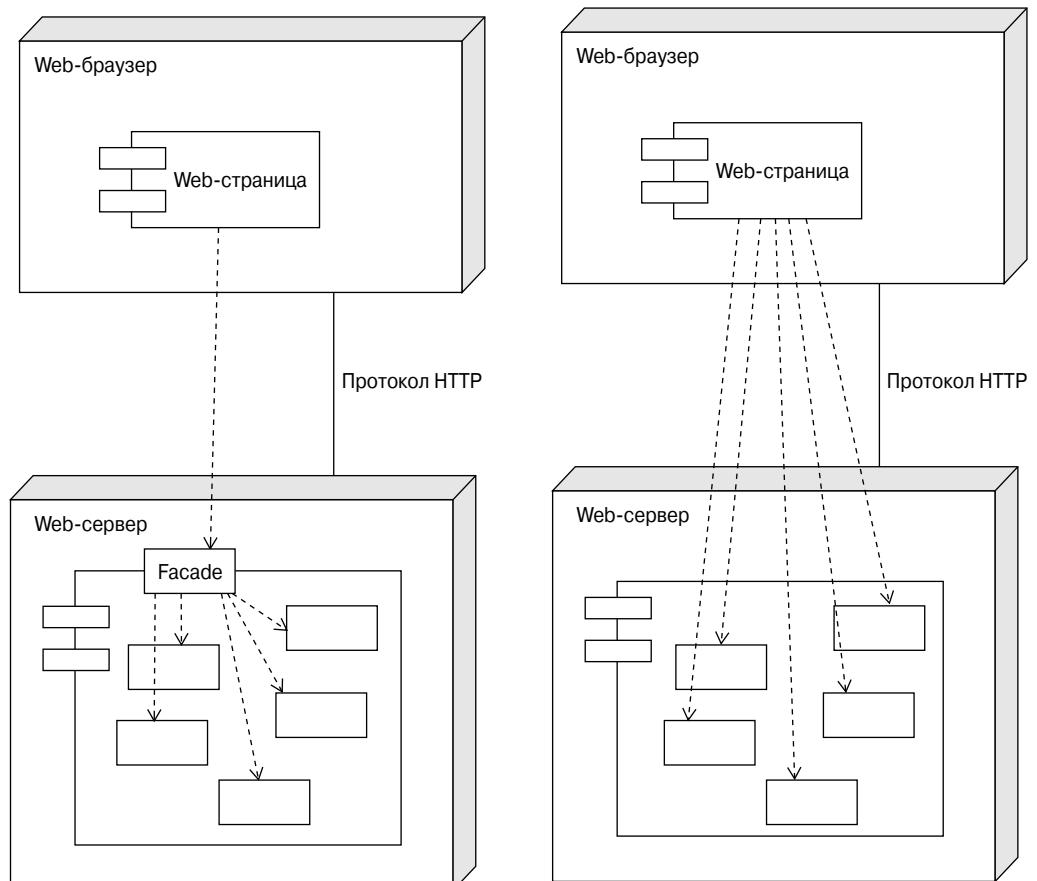


Рис. 4.12.

Допустим, вы разрабатываете небольшое Web-приложение. Со временем заказчик просит добить к нему новые свойства, и число классов приложения растет. Страницы Web-узла многократно обращаются к различным объектам. Такие страницы могут выглядеть следующим образом.

```
$dbManager = new DBManager();
$userArray = $dbManager->getNewUsers();
$emailer = new Emailer();
$stats = new StatLog();

for($i = 0; $i < count($userArray[$i]); $i++) {

    $user = $userArray[$i];
    $userPref = $user->getMailPreference();
    $userMail = $user->getEmail();

    if ($userPref == true) {
        $emailer->sendMailToUser($userMail);
    } else {
        $stats->storeUnmailedUser($user->getID());
    }
}
```

Приведенный гипотетический код отправляет новым пользователям по электронной почте сообщение в том случае, если они подписались на получение подобных сообщений. Если они не хотят получать почту, их идентификаторы userID будут переданы той части приложения, которая ведет статистику пользователей. Хотя этот код достаточно симпатичен, он может оказаться не самым мудрым решением, если приходится объединять большие фрагменты HTML и получать информацию для сообщений пользователям. Еще один вариант использования шаблона Facade выглядит следующим образом.

```
Application::mailNewUsers();
```

Теперь при необходимости использовать эту функциональность страница должна обращаться только к фасадному объекту, в данном случае — классу Application. Все сообщения от клиентской части приложения адресуются классу Application и никогда не обращаются напрямую к другим объектам системы. Методы объекта Application могут быть статическими, так как вам не потребуется создавать экземпляры этого класса.

Отметим, что клиент класса Application не обязан знать, как именно он работает. Одним из преимуществ такого подхода, помимо более ясного кода, является возможность разделения уровней представления и функциональности. Чем меньше приложение задействовано в непосредственном выводе HTML, тем более удобным будет его повторное использование. Более подробное обсуждение этих приемов проектирования содержится в главе 13 “Модель, вид, контроллер”.

Шаблон Builder

Возвращаясь к шаблону Composite, можно заметить, что объекты, составляющие композит, были созданы вручную. Это показано в следующем примере.

```
$drums = new DrumSet("ударные");
$drums->add(new SnareDrum("барабан высокого звучания "));
$drums->add(new BaseDrum("большой барабан низкого звучания"));
```

```
$cymbals = new Cymbal("набор тарелок");
$cymbals->add(new Cymbal("маленькая тарелка"));
$cymbals->add(new Cymbal("большая высокая шляпа"));
$drums->add($cymbals);
```

Код для создания композитного объекта DrumSet не очень сложен. Однако если объекты Instrument или наборы инструментов придется создавать для разных музыкантов (объектов Musician), а объекты Musician в свою очередь входят в объект Band, ситуация усложнится.

Вполне реально написать код, который будет создавать новые объекты и добавлять их в объекты-композиты, иначе вам быстро наскучит возиться с несколькими оркестрами, каждый из музыкантов в которых играет на своем наборе инструментов. Если же приходится создавать ансамбли в ответ на действия пользователя, это может вызвать некоторые сложности, особенно если каждый тип ансамбля с его музыкантами и инструментами нужно жестко задавать.

Реализация

Допустим, необходимо создать мастер, который генерирует объект Band для конечного пользователя. Пользователь выбирает жанр из списка (например, рок, кантри, попса и т.д.), в результате чего создается объект Band вместе с музыкантами и их инструментами. На рис. 4.13 показана диаграмма классов для шаблона Builder.

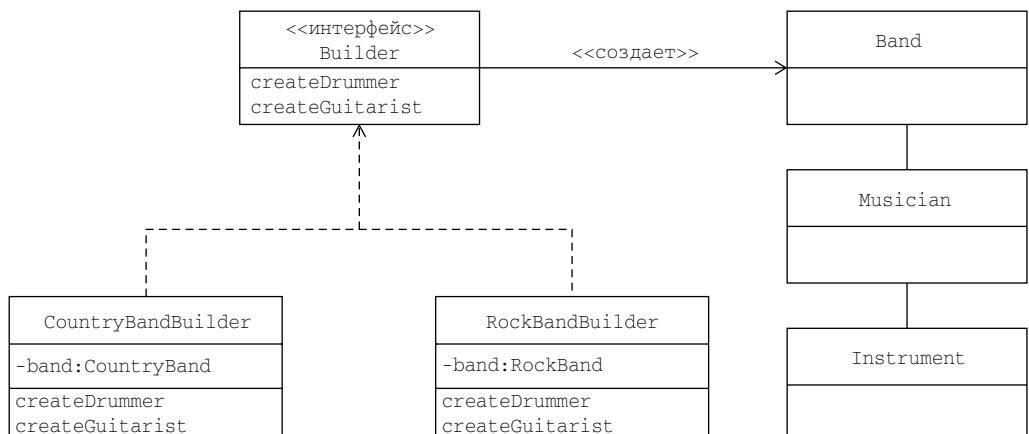


Рис. 4.13.

В этом примере на основе шаблона Builder создаются объекты Band, которые содержат только два типа музыкантов Musicians: гитариста и барабанщика.

```
<?php

interface Builder {
    public function buildDrummer();
    public function buildGuitarist();
}

?>
```

Каждый раз при инстанцировании конкретного класса-конструктора Builder создается объект Band. Рассмотрим следующий код класса RockBandBuilder.

```
<?php
require_once("interface_builder.php");
require_once("class_rockband.php");
require_once("class_musician.php");
require_once("class_instrument.php");

class RockBandBuilder implements Builder {

    private $band;

    function __construct($name) {
        $this->band = new RockBand($name);
    }

    public function getBand() {
        return $this->band;
    }

    public function buildDrummer() {

        $musician = new Musician("барабанщик в стиле рок");

        $drumset = new Instrument("барабан для рок-ансамбля");
        $drumset->add(new Instrument("тарелка"));
        $drumset->add(new Instrument("барабан низкого звучания"));
        $drumset->add(new Instrument("барабан высокого звучания"));

        $musician->addInstrument($drumset);
        $this->band->addMusician($musician);
    }

    public function buildGuitarist() {

        $musician = new Musician("рок-гитарист");

        $guitar = new Instrument("электрогитара");

        $musician->addInstrument($guitar);
        $this->band->addMusician($musician);

    }
}
?>
```

Обратите внимание, как в реализации конструктора класса RockBandBuilder создается объект RockBand. Подобным образом конструктор класса CountryBandBuilder создает объект CountryBand. Это показано в следующем коде, который необходимо сохранить в файле countryband_builder.php.

```
<?php
require_once("interface_builder.php");
require_once("class_musician.php");
require_once("class_countryband.php");
require_once("class_instrument.php");

class CountryBandBuilder implements Builder {
```

```

private $band;

function __construct() {
    $this->band = new CountryBand();
}

public function getBand() {
    return $this->band;
}

public function buildDrummer() {

    $musician = new Musician("музыкант с бубном");

    $drumset = new Instrument("бубен");

    $musician->addInstrument($drumset);
    $this->band->addMusician($musician);
}

public function buildGuitarist() {

    $musician = new Musician("гитарист в стиле кантри");

    $guitar = new Instrument("акустическая гитара");

    $musician->addInstrument($guitar);
    $this->band->addMusician($musician);
}
?>
```

Рассмотрим метод `buildDrummer()` класса `RockBandBuilder` и сравним его с аналогичным методом класса `CountryBandBuilder`. Каждый из них создает не только объект `Musician`, но инструменты для этого музыканта. Благодаря наличию интерфейса `Builder` в классах-конструкторах с суффиксом `Builder` необходимо реализовать методы `buildDrummer()` и `buildGuitarist()`, которые совершенно по-разному создают объекты `Musician`, включая принадлежащие этим музыкантам объекты `Instrument`.

Несложно заметить, что классы `Musician` и `Instrument` уже не являются подклассами определенных подтипов, таких как `Guitarist`. Это сделано для того, чтобы продемонстрировать использование классов-конструкторов с суффиксом `Builder` для создания разных объектов (как в конструкторах этих классов) или одних и тех же объектов с разными параметрами и операциями (как метод `buildDrummer()`).

Шаблон Director

Шаблон `Builder` подразумевает использование еще одного шаблона —`Director`. Шаблон `Director` отвечает за вызов методов класса с суффиксом `Builder` для создания конечного продукта, которым в данном случае является объект `Band`. В этом примере шаблон `Director` реализован объектом `Application`, похожим на используемый в шаблоне `Facade`. Класс `Application` следующего вида нужно сохранить в файле `application.php`.

```

<?php

class Application {

    public static function createBand(Builder & $builder) {
```

```

    $builder->buildGuitarist();
    $builder->buildDrummer();

    return $builder->getBand();
}

?>

```

Класс Application содержит один метод, `createBand()`, которому в качестве аргумента передается объект типа Builder. Затем объект Application вызывает методы `create()` переданных объектов Builder. Следует запомнить два момента. Во-первых, объект Application сам определяет, какой из методов класса Builder следует вызвать. Во-вторых, ему не важно, какой именно тип объекта-конструктора передан ему.

Ниже приведен пример совместного использования объекта Application (реализующего шаблон Director) и класса Builder. Просто создайте нужный объект Builder и передайте его методу `createBand()` объекта Application.

```

$builder = new RockBandBuilder();
$band = Application::createBand($builder);

```

Объекты Application и Builder совместно используются при пошаговом создании корректного объекта Band. Объект Band хранится в объекте Builder до тех пор, пока его не запросит метод `getBand()`.

Обсуждение

Шаблон Builder удобен для создания объектов-композитов и иерархических структур объектов, таких как `Band -> Musician -> Instrument`. Шаблон Builder возвращает готовый объект Band, который ничем не отличается от созданного вручную. Это означает, что его можно изменять после создания.

Класс Director можно модифицировать, реализовав в нем несколько методов создания объектов. В объекте Application содержится только один метод `createBand()`, однако можно создать различные методы, обеспечивающие разные конфигурации групп с помощью одного и того же конструктора. Например, можно создать метод `createTrio()`. Передавая в качестве параметра различные объекты Builder, с помощью одного и того же метода можно создать либо рок-трио, либо трио, исполняющее музыку в стиле кантри.

Резюме

В этой главе вы познакомились с пятью различными шаблонами проектирования.

- Composite
- Observer
- Decorator
- Facade
- Builder

Хотя каждый шаблон решает свою проблему, их объединяет одна общая особенность. После завершения разработки шаблона реально используемый код очень прост

и легко расширяем. Шаблоны позволяют разработчикам обмениваться сложными идеями в едином контексте. Шаблоны — это повторно используемые решения. Один шаблон может быть реализован в совершенно разном программном обеспечении. Вы часто будете узнавать шаблоны проектирования в различных приложениях и на собственном опыте поймете, как они работают независимо от языка программирования.

Пять представленных шаблонов составляют лишь малую часть существующего многообразия. Читатель может продолжить изучение шаблонов проектирования самостоятельно — они являются мощным инструментом для серьезного разработчика.

Часть II

Разработка повторно используемого набора объектов: простые служебные классы и интерфейсы

В ЭТОЙ ЧАСТИ...

Глава 5. Класс Collection

Глава 6. Класс CollectionIterator

Глава 7. Класс GenericObject

Глава 8. Уровни абстракции базы данных

Глава 9. Интерфейс Factory

Глава 10. Управление событиями

Глава 11. Регистрация событий и отладка

Глава 12. Протокол SOAP

5

Класс Collection

Теперь, когда вы познакомились со всеми подробностями создания объектно-ориентированных приложений с использованием языка PHP, может показаться, что самое время пуститься в свободное плавание и приступить к написанию своего собственного приложения. Однако на самом деле для достижения желанной цели еще придется пройти долгий путь. Существует большое количество различных утилит и повторно используемых классов, совместное применение которых является чрезвычайно мощным и гибким инструментом разработки. В этой части книги этот инструментарий будет подробно рассмотрен, а его практическое применение будет продемонстрировано на реальных примерах. Первый класс из этого списка называется *Collection* (коллекция). Именно его изучению и посвящена данная глава.

Класс *Collection* — это объектно-ориентированный аналог традиционного типа данных *array*. Как и в обычном массиве, в коллекции могут содержаться элементы, хотя чаще всего в качестве таких элементов используются другие объекты, а не переменные простых типов, например строки, целые и т.д. Класс *Collection* позволяет добавлять, удалять и извлекать элементы для использования в приложениях. Как вы увидите ниже в этой главе, использование класса *Collection* для хранения набора инстанцированных объектов предоставляет гораздо большие преимущества по сравнению с применением обычных массивов.

Как и во всех других главах этой части, в данной главе вы не только познакомитесь с программным кодом самого класса *Collection*, но и увидите, как его можно использовать в контексте конкретных требований. Кроме того, здесь будут рассмотрены также вопросы *позднего* инстанцирования на основе обратных вызовов, основные принципы использования класса *Collection* и предложения по его усовершенствованию.

Вместе с классом *CollectionIterator*, который рассматривается в следующей главе, класс *Collection* предоставляет мощный, но вместе с тем простой в использовании механизм обработки групп объектов в приложениях.

Назначение класса Collection

Зачастую в приложениях используются объекты, которые содержат группу других объектов. Например, в приложении для университетской администрации наверняка пригодится класс `Student` (студент) и `Course` (курс). Скорее всего, объект `Student` будет связан с несколькими объектами `Course`. Наиболее очевидным способом конструирования такой взаимосвязи является создание массива объектов `Course`, который будет использоваться как переменная-член объекта `Student`.

```
/* класс Student */
class Student {
    public $courses = array();

    // ...другие методы/свойства
}

/* использование класса Student */
$objStudent = new Student(1234); // конструктор выше не показан
foreach($objStudent->courses as $objCourse) {
    print $objCourse->name;
}
```

Конечно, если бы наиболее очевидный подход был наилучшим, эту главу писать не было бы необходимости.

В приведенном выше фрагменте есть несколько проблем. Во-первых, открытый массив объектов `Course` нарушает инкапсуляцию. Кроме того, отсутствует возможность проверки факта изменения массива или состояния объекта `Student`, что зачастую может оказаться весьма необходимым. Во-вторых, из реализации не совсем понятно, как индексируются элементы массива курсов и как можно перебрать эти элементы, чтобы найти требуемый объект `Course`. И самое главное, если понадобится получить данные только о студентах, придется извлекать из базы данных также и всю информацию о курсах. Это означает, что если нужно напечатать лишь имена студентов, то придется извлекать и много лишней информации, что существенно увеличит нагрузку на сервер базы данных и приведет к замедлению работы приложения.

Для решения всех этих проблем и предназначен класс `Collection`. Он предоставляет объектно-ориентированную оболочку вокруг массива и реализует механизм *позднего инстанцирования* (*lazy instantiation*). Этот механизм позволяет отложить создание членов коллекции до того момента, когда это действительно станет необходимым. Такое инстанцирование называется поздним, поскольку решение о создании объектов принимается самим приложением. Одновременно с созданием объекта `Student` объекты `Course` не должны создаваться. Эта операция должна быть выполнена лишь при первой попытке доступа к курсам. Причем это должно осуществляться автоматически, без каких бы то ни было дополнительных усилий разработчика.

Таким образом, к классу `Collection` можно сформулировать следующие функциональные требования.

1. Предоставление оболочки вокруг массива объектов.
2. Обеспечение простых и понятных механизмов добавления, удаления и извлечения объектов.
3. Простое определение количества объектов в коллекции.
4. Поддержка позднего инстанцирования для рационального использования системных ресурсов.

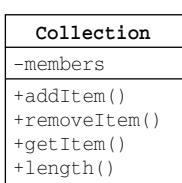


Рис. 5.1.

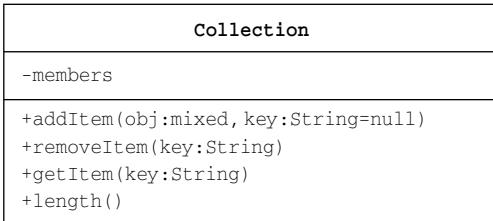


Рис. 5.2.

Проектирование класса Collection

Перед написанием какого-либо кода нужно удостовериться, что все требуемые классы тщательно спроектированы. Воспользовавшись требованиями 1–3 из приведенного выше перечня, можно построить следующую диаграмму классов (рис. 5.1).

Для “простого и понятного” управления операциями добавления, удаления и извлечения членов коллекции пользователи этого объекта должны иметь возможность задания имени ключа при добавлении объекта. Ключ должен использоваться подобно строковому индексу в ассоциативном массиве и точно определять местоположение в коллекции, в которой хранятся соответствующие объекты. Позднее этот ключ можно использовать для извлечения или удаления объекта. В предыдущем примере приложения для университета в качестве ключа для каждого курса может использоваться его код (например, CS101 — для начального курса компьютерных наук (computer sciences)). В некоторых случаях осмысленные ключи могут и не понадобиться. Тогда ключ коллекции должен выбираться автоматически. Обновленная диаграмма на языке UML показана на рис. 5.2.

Параметр `obj` метода `addItem()` — это элемент, добавляемый в коллекцию. Обычно значением этого параметра является объект, однако могут использоваться и значения любого другого типа данных. Строковый параметр `key` методов `addItem()`, `removeItem()` и `getItem()` (в методе `addItem()` его использовать необязательно) является ключом.

После определения базового кода для манипуляции содержимым коллекции мы еще раз вернемся к этой диаграмме и более подробно разберемся с поздним инстанцированием.

И наконец, метод `length()` просто возвращает текущее количество элементов коллекции.

Основы класса Collection

Руководствуясь диаграммой UML, показанной на рис. 5.2, каркас класса `Collection` можно представить следующим образом.

```

<?php
class Collection {

    private $_members = array();

    public function addItem($obj, $key = null) {
  
```

```

    }

    public function removeItem($key) {
    }

    public function getItem($key) {
    }

    public function length() {
    }

}

?>

```

Переменная `$_members` предоставляет место для хранения объектов, которые являются членами коллекции. Метод `addItem()` позволяет добавлять в коллекцию новые объекты. Метод `removeItem()` предназначен для удаления объекта, а метод `getItem()` — для извлечения объекта. И наконец, метод `length()` возвращает количество элементов коллекции. Классу `Collection` конструктор не требуется.

Метод `addItem()`

При добавлении нового объекта в коллекцию он вставляется в массив `$_members` в позиции, которая определяется параметром `$key`. Если ключ явно не указан, то его значение будет выбрано интерпретатором PHP. При попытке добавления объекта в коллекцию с уже существующим ключом должно быть сгенерировано исключение. Это позволит избежать случайного перезаписывания и, соответственно, потери данных.

```

class Collection {

    private $_members = array;

    public function addItem($obj, $key = null) {
        if($key) {
            if(isset($this->_members[$key])) {
                throw new KeyInUseException("Ключ \"{$key}\" уже используется!");
            } else {
                $this->_members[$key] = $obj;
            }
        } else {
            $this->_members[] = $obj;
        }
    }
}

```

Как и в большинстве подклассов класса `Exception`, используемых в этой книге, класс `KeyInUseException` не имеет тела и полностью наследует класс `Exception` из комплекта поставки PHP 5.

```
class KeyInUseException extends Exception { }
```

Класс `KeyInUseException` позволяет избежать повторного использования ключа и, как следствие, предотвратить перезаписывание данных. Значение ключа используется в качестве индекса в массиве `$_members`. Если ключ не задан, то для элементов выбирается числовой индекс. В любом случае объект размещается в массиве в соответствии со значением либо заданного, либо сгенерированного ключа.

В подклассах класса Collection можно переопределить метод addItem() и обеспечить, чтобы в коллекцию добавлялись лишь объекты требуемого типа. Например:

```
<?php
class CourseCollection extends Collection {

    public function addItem(Course $obj, $key = null) {
        parent::addItem($obj, $key);
    }
}
?>
```

Используя приведенный подход, можно создавать подклассы, которые обеспечивают добавление в коллекцию лишь элементов требуемого типа.

Методы getItem() и removeItem()

Методы removeItem() и getItem() получают ключ в качестве параметра и позволяют точно определить, какой из элементов будет удален или извлечен из коллекции. При передаче некорректного ключа генерируется исключение.

```
public function removeItem($key) {
    if(isset($this->_members[$key])) {
        unset($this->_members[$key]);
    } else {
        throw new KeyInvalidException("Некорректный ключ \\"$key\\" !");
    }
}

public function getItem($key) {
    if(isset($this->_members[$key])) {
        return $this->_members[$key];
    } else {
        throw new KeyInvalidException("Некорректный ключ \\"$key\\" !");
    }
}
```

Класс InvalidKeyException аналогичен классу KeyInUseException.

```
Class KeyInvalidException extends Exception { }
```

Другие методы

Поскольку параметр \$key методу addItem() передавать необязательно, то вовсе не нужно знать ключ каждого элемента коллекции. Получить массив со всеми ключами коллекции позволяет функция keys().

```
public function keys() {
    return array_keys($this->_members);
}
```

Возможно, понадобится определить, сколько элементов содержится в коллекции. Функция PHP sizeof возвращает количество элементов в массиве, поэтому ее можно использовать для реализации метода length().

```
public function length() {
    return sizeof($this->_members);
}
```

Поскольку после задания неверного ключа с помощью метода `getItem()` сгенерируется исключение, то в классе `Collection` должны быть реализованы средства, с помощью которых можно определить, используется ли в коллекции заданный ключ. Для этого предназначен метод `exists()`, который можно вызывать перед методом `getItem()`.

```
public function exists($key) {
    return (isset($this->_members[$key]));
}
```

Таким образом, для обеспечения корректности выполнения кода можно либо воспользоваться блоком `try...catch` для перехвата некорректных ключей, либо прибегнуть к вызову метода `exists()` перед вызовом метода `getItem()`. Какой из способов выбрать, зависит от конкретной ситуации.

Теперь, когда в класс `Collection` были добавлены все основные методы, можно перейти к изучению способов его использования.

Использование класса `Collection`

Для того чтобы воспользоваться классом `Collection`, создайте файл `class.Collection.php` и разместите в нем код класса. Создайте также файлы для классов `KeyInvalidException` и `KeyInUseException`. Для обеспечения возможности использования классов исключений добавьте соответствующие операторы `require_once` в начало файла `class.Collection.php`. И наконец, в файле `testCollection.php` сохраните следующий код.

```
<?php
/* простой класс для тестирования */
class Foo {
    private $_name;
    private $_number;

    public function __construct($name, $number) {
        $this->_name = $name;
        $this->_number = $number;
    }

    public function __toString() {
        return $this->_name . ' это номер ' . $this->_number;
    }
}

$colFoo = new Collection();
$colFoo->addItem(new Foo("Стив", 14), "стив");
$colFoo->addItem(new Foo("Эд", 37), "эд");
$colFoo->addItem(new Foo("Боб", 49));

$objSteve = $colFoo->getItem("стив");
print $objSteve;          //вывод строки "Стив это номер 14"
$colFoo->removeItem("стив"); //удаление объекта 'стив'

try {
    $colFoo->getItem("стив"); // генерация исключения KeyInvalidException
} catch (KeyInvalidException $kie) {
    print "В коллекции не может содержаться ничего с ключом 'стив'";
}
?>
```

Конечно, приведенный пример еще не очень интересен, однако его вполне достаточно, чтобы сформулировать некоторые идеи по использованию класса `Collection`. В следующем разделе вы узнаете, как использовать позднее инстанцирование, которое является одним из главных преимуществ этого класса.

Реализация позднего инстанцирования

Позднее инстанцирование — это свойство класса `Collection`, которое позволяет отложить создание членов коллекции до того момента, когда это действительно необходимо. В примере приложения для университета, который обсуждался в начале этой главы, объект `Student` был связан с несколькими объектами `Course`. При использовании объекта `Student` для отображения имени студента не требуется никакой информации о курсах. Однако в соответствии с определенным интерфейсом объекты класса `Course` должны быть доступными переменными-членами объекта класса `Student`.

Для обеспечения простоты отображения перечня курсов для заданного студента программный интерфейс должен предоставлять возможность написания следующего кода.

```
<?php
```

```
$objStudent = StudentFactory::getStudent(12345); //12345 - ID студента
print "Имя: " . $objStudent->name . "<br>\n";
print "Курсы: <br>\n";
foreach($objStudent->courses as $objCourse) {
    print $objCourse->coursecode . " - " . $objCourse->name . "<br>\n";
}
?>
```

Сейчас достаточно знать лишь то, что `StudentFactory` — это класс со статическим методом `getStudent`, который возвращает новый объект `Student`, получая его идентификатор (ID) в качестве параметра. Именно этот класс выполняет все необходимые действия над базой данных. Классы-фабрики подробно рассматриваются в главе 9.

Можно предположить, что заполнением списка курсов занимается конструктор класса `Student`. Однако если требуется получить просто список студентов, то нет необходимости тратить дополнительные ресурсы на извлечение курсов. Еще раз предположим, что `StudentFactory` — это класс, позволяющий создавать объекты класса `Student` путем извлечения информации из базы данных. Тогда можно построить коллекцию, выполнив поиск по фамилиям студентов.

```
<?php
```

```
$colStudents = StudentFactory::getByLastName("Смит");
print "<h1>Студенты с фамилией 'Смит'</h1>";
foreach($colStudents as $objStudent) {
    print $objStudent->name . "<br>\n";
}
?>
```

В рассмотренном примере информация о курсах не требуется, поскольку отображаются только имена студентов. Однако как в этом случае сохранить простой интерфейс класса `Student`, не изменяя взаимодействие с базой данных? Можно добавить метод `$objStudent->loadCourses()`, который заполнял бы коллекцию и вызывался бы до взаимодействия с коллекцией курсов, однако это не очень наглядно и приводит к снижению ясности интерфейса. Можно также реализовать метод `CourseFactory::getCoursesForStudent($objStudent)`, который возвращал бы коллекцию курсов для данного студента. Однако и в этом случае решение будет не очень очевидным для

новичка в команде разработчиков, которому требуется воспользоваться функцией для получения курсов для данного студента. Существует более удачный способ, который связан с использованием обратных вызовов.

Обратные вызовы

Если у вас есть опыт программирования на языке JavaScript и вы умеете связывать выполнение некоторых действий с наступлением события `onClick`, значит, вы уже применяли обратный вызов. *Обратный вызов* (callback) — это прием, с помощью которого можно указать приложению выполнить некоторую функцию в ответ на наступление определенного события. Это событие не контролируется разработчиком. Более того, нельзя даже сказать, произойдет ли это событие вообще. Все, что можно сделать в данной ситуации, заключается в том, чтобы сообщить компьютеру о необходимости выполнения функции при наступлении определенного события. В языке JavaScript обработчик события `onSubmit` можно использовать для проверки данных формы в клиентской части приложения, если пользователь решил отправить их на сервер. Причем неизвестно, когда это событие произойдет, если произойдет вообще. После определения обработчика события всю оставшуюся работу берет на себя модуль JavaScript. Конечно же, подобный подход можно использовать и при разработке серверной части приложения.

При разработке класса `Student` в приложении для университета нельзя точно сказать, когда будет предпринята попытка получения доступа к коллекции объектов `Course`. Поскольку для извлечения этих объектов требуются существенные затраты системных ресурсов, то для снижения ресурсоемкости этой операции можно воспользоваться обратным вызовом. Другими словами, необходимо обеспечить следующее: при попытке обращения к коллекции `Course` сначала ее нужно создать.

Однако, поскольку класс `Collection` входит в состав повторно используемого инструментария, то вряд ли удачной идеей будет жесткая привязка к имени функции внутри класса (это существенно ограничит возможность его повторного использования). Кроме того, вы наверняка не захотите создавать новый подкласс класса `Collection` каждый раз, когда этот класс нужно использовать. (Это ни что иное, как лишняя работа.) Гораздо лучше обеспечить возможность указать имя функции или ссылку на объект, метод которого нужно вызвать для создания коллекции. Для реализации такого подхода необходимо познакомиться со специальной встроенной функцией языка PHP.

Использование функции `call_user_func()`

При вызове функции обычно используется ее имя. Однако язык PHP позволяет вызывать функции (или методы объектов) с помощью строковых переменных. С точки зрения синтаксиса PHP следующий фрагмент абсолютно корректен.

```
<?php
$myFunc = "pow";
print $myFunc(4, 2); // вывод 16, или pow(4, 2)
?>
```

Интерпретатор PHP оценивает имя переменной и выполняет функцию с заданным именем. Такой прием можно использовать как для вызова встроенных функций, так и для вызова пользовательских функций. Тот же подход допустимо применять и для вызова метода объекта.

```
<?php
$myMethod = 'sayHello';
$obj = new Person();
$obj->$myMethod();
```

В этом примере описанный подход использовался для вызова метода `sayHello()` класса `Person`. (Точно так же можно вызывать и статические методы, т.е. `Person::$myMethod()`.)

При использовании описанного приема существует одна проблема, которая связана с тем, что не совсем очевидно то, что конструкция `$obj->$myMethod()` приводит к вызову метода `$obj->sayHello()`. Это может оказаться еще менее очевидным, если значение переменной `$myMethod` будет передаваться в качестве параметра некоторой функции-оболочки. Язык PHP предоставляет встроенный механизм подобных вызовов, который оказывается гораздо более наглядным и понятным.

Функция `call_user_func()` имеет один обязательный параметр, который используется для задания вызываемой функции и может получать несколько дополнительных параметров, передаваемых в качестве аргументов в функцию, заданную пользователем. Она определена следующим образом.

```
mixed call_user_func(функция_обратного_вызова [, mixed параметр [, mixed ...]])
```

Параметр `функция_обратного_вызова` может принимать одну из следующих трех форм.

- ❑ `string $имяФункции`. Эта строка определяет имя вызываемой функции.
- ❑ `array(object $объект, string $имяФункции)`. Этот массив содержит имя инстанцированного объекта, а строка определяет имя вызываемого метода этого объекта.
- ❑ `array (string $имяКласса, string $имяФункции)`. Массив содержит имя класса, а строка соответствует имени *статического* метода этого класса.

Вот несколько примеров использования функции `call_user_func`.

```
<?php
class Bar {
    private $_foo;

    public function __construct($fooVal) {
        $this->foo = $fooVal;
    }

    public function printFoo() {
        print $this->_foo;
    }

    public static function sayHello($name) {
        print "Здравствуй, $name!";
    }
}

//отдельная функция, а не часть класса Bar
function printCount($start, $end) {
    for($x = $start; $x <= $end; $x++) {
        print "$x";
    }
}

//вывод 1 2 3 4 5 6 7 8 9 10
call_user_func('printCount', 1, 10); /* пример 1 */

//вызов $objBar->printFoo()
$objBar = new Bar('слон'); /* пример 2 */
call_user_func(array($objBar, 'printFoo'));
```

```
//вызов Bar::sayHello('Стив')
call_user_func(array('Bar', 'sayHello'), 'Стив'); /* пример 3 */

//Генерация фатальной ошибки об использовании переменной
// $this вне контекста объекта, поскольку вызываемая
//функция Bar::printFoo не является статическим методом
call_user_func(array('Bar', 'printFoo')); /* пример 4 */
?>
```

Первый пример вызова функции `call_user_func()` чрезвычайно прост. Имя функции передается в виде строки вместе с двумя ожидаемыми параметрами. Во втором примере в качестве первого параметра передается массив. Поскольку его первым элементом является объект, то вызывается метод `printFoo()` этого объекта. В третьем и четвертом примерах первым элементом массива является строка. Следовательно, интерпретатор PHP попытается вызвать статический метод указанного класса. Если требуемый метод не существует или в нем используются нестатические переменные класса (например, `$this`), будет сгенерирована фатальная ошибка времени выполнения. Именно это и произойдет при обработке последней строки кода при вызове метода `printFoo()`.

Терминологическая справка. Статические методы не зависят от нестатических переменных-членов класса и могут вызываться без предварительного инстанцирования объекта этого класса. Для их вызова используется следующий синтаксис: `ИмяКласса::имяМетода()`. Ключевое слово `$this` в статических методах использовать нельзя.

Реализация обратного вызова

Для создания обратного вызова в одном из классов можно воспользоваться функцией `call_user_func()`. Создав метод с такими же параметрами, что и функция `call_user_func()`, вне данного класса можно определить функцию или метод, который будет вызываться при наступлении в этом классе определенного события.

Для создания обратного вызова в классах выполните следующие действия.

1. Определите, какие события могут происходить в классе.
2. Для каждого из этих событий создайте метод `on [ИмяСобытия]` с двумя следующими параметрами.
 - Имя функции.
 - Необязательное имя объекта или класса. Параметр-объект должен быть инстанцированной переменной-объектом. Имя класса должно быть строковым.
3. Для каждого события создайте скрытую переменную-член класса, которая будет хранить параметры в виде строки (если имя объекта или класса не были указаны) или в виде массива (если второй параметр имеет значение).
4. В каждой точке, где должны быть выполнены обратные вызовы, обратитесь к скрытой переменной-члену, созданной в п. 3, и проверьте, было ли установлено ее значение. Если значение было установлено, то можно вызвать функцию `call_user_func()` и передать ей это значение в качестве параметра.

В следующем примере показано, как создать событие `onspeak` в классе `Dog`. Создайте функцию `onspeak()`, которая в виде строки получает имя функции и (необязательно)

имя объекта или класса. Далее сконструируйте строку или массив, которые должны быть переданы функции `call_user_func()`, и выполните их проверку на корректность. Если проверка прошла успешно, сохраните это значение в скрытой переменной-члене для последующего использования. Чтобы увидеть, как эта переменная используется, проанализируйте метод `bark()`.

```
<?php
class Dog {
    private $_onspeak;
    public function __construct($name) {
        $this->_name = $name;
    }

    public function bark() {
        if(isset($this->_onspeak)) {
            if(! call_user_func($this->_onspeak)) {
                return false;
            }
        }
        print "Гав, гав!";
    }

    public function onspeak($functionName, $objOrClass = null) {
        if($objOrClass) {
            $callback = array($objOrClass, $functionName);
        } else {
            $callback = $functionName;
        }

        //убедитесь, что это значение корректно
        if(!is_callable($callback, false, $callableName)) {
            throw new Exception("$callableName не является ".
                "корректным параметром onspeak");
            return false;
        }

        $this->_onspeak = $callback;
    }
} //завершение класса Dog

//функция
function isEveryoneAwake() {
    if(time() < strtotime("today 8:30am") || time() > strtotime("today 10:30pm")) {
        return false;
    } else {
        return true;
    }
}

$objDog = new Dog('Фидо');
$objDog->onspeak('isEveryoneAwake');
$objDog->bark(); //воспитанный пес

$objDog2 = new Dog('Мара');
$objDog2->bark(); //всегда лает!

//При вызове onspeak генерируется исключение.
$objDog3 = new Dog('Лэсси');
$objDog3->onspeak('nonExistentFunction', 'NonExistentClass');
$objDog3->bark();
?>
```

Функция `isEveryoneAwake()` проверяет, попадает ли текущее время в промежуток между 8:30 и 22:30. Если это так, она возвращает истинное значение, тем самым “позволяя” собаке лаять. В противном случае возвращается ложное значение, и тишина лаем не нарушается. Если бы таким образом можно было запрограммировать настоящего пса! Рассмотренный подход корректно работает только для объекта `$objDog`, поскольку для объекта `$objDog2` не задано требуемых параметров обратного вызова.

Обратите внимание на использование функции `is_callable()` в методе `onSpeak()`. Это еще одна встроенная функция PHP. С ее помощью выполняется тестовая проверка корректности параметров, которые будут переданы при обратном вызове. При передаче параметров методу `$objDog3->onSpeak()` будет сгенерировано исключение, поскольку были указаны некорректные имена функции и класса.

Метод `setLoadCallback()` класса Collection

Теперь, когда вы познакомились с обратными вызовами, нужно разобраться, как их можно применить в классе `Collection`, и реализовать одно из его главных преимуществ — позднее инстанцирование.

Снова вернемся к классу `Student` из приложения для университета. Как уже упоминалось в начале этой главы, необходимо сохранить его простой интерфейс, в котором объекты класса `Course` являются переменными-членами класса `Student`. Однако вместе с тем нужно обеспечить минимальное потребление системных ресурсов при извлечении объектов класса `Student`, т.е. информацию о курсах нужно получать лишь при первом обращении к ней. Для этого можно воспользоваться обратным вызовом в классе `Collection` и вызвать некоторую функцию перед тем, как извлечь класса к требуемой информации будет осуществлен доступ.

Перед каждым вызовом методов `addItem()`, `getItem()`, `length()`, `exists()` и `keys()` сначала нужно удостовериться, что коллекция была загружена, и, если это не так, с использованием обратного вызова получить всю необходимую информацию. Именно функции, используемые при обратном вызове, и должны обеспечивать заполнение коллекции. Кроме того, в своем распоряжении также нужно иметь возможность проверки того, что обратный вызов уже был выполнен. Это позволит избежать ненужных повторных попыток заполнения коллекции, которая уже была загружена, и таким образом предотвратить снижение общей производительности приложения.

В приведенном ниже фрагменте представлен завершенный класс `Collection` со всеми необходимыми изменениями, которые позволяют реализовать позднее инстанцирование (они выделены серым фоном).

```
<?php
class Collection {

    private $_members = array(); //члены коллекции

    private $_onload;           //поле для обратного вызова
    private $_isLoading = false; //флаг проверки, был ли
                                //выполнен обратный вызов

    public function addItem($obj, $key = null) {
        $this->_checkCallback(); // _checkCallback определяется позже
    }
}
```

```

if($key) {
    if(isset($this->_members[$key])) {
        throw new KeyInUseException("Ключ \"\$key\" уже используется!");
    } else {
        $this->_members[$key] = $obj;
    }
} else {
    $this->_members[] = $obj;
}

public function removeItem($key) {
    $this->_checkCallback();

    if(isset($this->_members[$key])) {
        unset($this->_members[$key]);
    } else {
        throw new KeyInvalidException("Неправильный ключ \"\$key\"!");
    }
}

public function getItem($key) {
    $this->_checkCallback();

    if(isset($this->_members[$key])) {
        return $this->_members[$key];
    } else {
        throw new KeyInvalidException("Неправильный ключ \"\$key\"!");
    }
}

public function keys() {
    $this->_checkCallback();

    return array_keys($this->_members);
}

public function length() {
    $this->_checkCallback();

    return sizeof($this->_members);
}

public function exists($key) {
    $this->_checkCallback();

    return (isset($this->_members[$key]));
}

/**
 * Используйте этот метод для задания функции, которая
 * будет вызываться перед обращением к коллекции.
 * Функция должна использовать коллекцию как
 * единственный параметр.
 */
public function setLoadCallback($functionName, $objOrClass = null) {
    if($objOrClass) {
        $callback = array($objOrClass, $functionName);
    }
}

```

```

} else {
    $callback = $functionName;
}

//убедитесь, что функция/метод корректны
if(!is_callable($callback, false, $callableName)) {
    throw new Exception("$callableName не является ".
        "корректным параметром onload");
    return false;
}

$this->_onload = $callback;
}

/**
 * Позволяет определить, установлен ли обратный вызов,
 * и если так, был ли он использован.
 * Если нет, запускается функция обратного вызова.
 */
private function _checkCallback() {
    if(isset($this->_onload) && !$this->_isLoaded) {
        $this->_isLoaded = true;
        call_user_func($this->_onload, $this);
    }
}
}
?>

```

Функция `setLoadCallback()` позволяет задать имя функции и при необходимости имя класса или объекта, которые будут использоваться для заполнения коллекции. Если к коллекции нужно применить обратный вызов, в скрытой переменной-члене `$_onload` размещается строка или массив, которые будут переданы функции `call_user_func`.

В реализацию остальных методов класса `Collection` в качестве первой добавлена строка `$this->checkLoadCallback()`. Это позволяет проверить существование коллекции и при необходимости осуществить обратный вызов. Обратите внимание также на проверку значения `$this->_isLoaded`, которая обеспечивает использование только одного вызова функции `call_user_func()`. Любая функция, используемая для загрузки коллекции класса `Collection`, должна получать только один параметр, который должен иметь тип `Collection`. Ссылка на данную коллекцию (переменная `$this`) передается функции обратного вызова в следующей строке.

```
call_user_func($this->_onload, $this)
```

Такой подход позволяет функции обратного вызова оперировать непосредственно объектом коллекции, который инициировал вызов функции. Следующий пример демонстрирует, как это работает в реальной ситуации.

Создадим класс `NightClub` (ночной клуб). Этот класс содержит свойство `name` и коллекцию объектов `Singer` (певец). Предположим, что в некоторых случаях вполне достаточно вывести на экран название ночного клуба. В других ситуациях некоторые действия нужно осуществить и над коллекцией певцов. Чтобы гарантированно не выполнять лишний код, воспользуемся методом `setLoadCallback()` коллекции объектов.

В следующем фрагменте кода приведены классы `NightClub` и `Singer`. Этот код нужно поместить в файл `club_singer.php`.

```
<?php
require_once("class.Collection.php");
class Singer {
    public $name;
```

```

public function __construct($name) {
    $this->name = $name;
}

class NightClub {
    public $name;
    public $singers;

    public function __construct($name) {
        $this->name = $name;
        $this->singers = new Collection();
        $this->singers->setLoadCallback('_loadSingers', $this);
    }

    private function _loadSingers(Collection $col) {
        print "(Загружаем певцов!)<br>\n";

        //здесь можно воспользоваться базой данных
        $col->addItem(new Singer('Frank Sinatra'));
        $col->addItem(new Singer('Dean Martin'));
        $col->addItem(new Singer('Sammy Davis, Jr.'));
    }
}
?>
```

Обратите внимание на оператор `print` в методе `_loadSingers()`. Здесь он используется только в демонстрационных целях. Запустите следующий код (его можно добавить в конец предыдущего фрагмента) и убедитесь, что оператор `print` не выводится на экран.

```
<?php
    $objNightClub = new NightClub('Золотые пески');
    print "Добро пожаловать в " . $objNightClub->name . "<br>\n";
?>
```

Теперь попробуйте снова, выполнив некоторые действия над коллекцией `$singers`.

```
>?php
    $objNightClub = new NightClub('Золотые пески');
    print "Добро пожаловать в " . $objNightClub->name . "<br>\n";

    print "У нас есть " . $objNightClub->singers->length() . " певцов" .
        "чтобы вы получили удовольствие сегодня вечером";
?>
```

В результате выполнения этого кода на экране должна появиться следующая информация.

```
Добро пожаловать в Золотые пески
(Загружаем певцов!)
У нас есть 3 певцов, чтобы вы получили удовольствие сегодня вечером
```

Как вы видите, это существенно упрощает использование коллекции `singers`. Создав класс `NightClub`, его можно передать менее опытным членам команды разработчиков, которые смогут использовать его для вывода информации на экран (как показано в предыдущем примере) без каких-либо знаний о том, как или когда эта коллекция была инициализирована. И вы можете быть уверены в том, что приложение не будет занимать больше системных ресурсов, чем это действительно необходимо.

В следующем разделе мы вернемся к приложению для университета и покажем, как использовать описанную технологию в реальной системе.

Использование класса Collection

Как упоминалось выше в этой главе, приложение для университета содержит класс `Student`, который включает объекты `Course`. Оба класса довольно просты. Диаграмма классов на языке UML показана на рис. 5.3.

Конечно, класс `Course` должен содержать коллекцию студентов `students`. Соответствующий код будет аналогичен тому, с помощью которого была реализована коллекция курсов `courses` для класса `Student`. Поэтому модификацию диаграммы UML и добавление кода, необходимого для обеспечения коллекции студентов для класса `Course`, оставим читателям для самостоятельной проработки в качестве упражнения.

Для класса `Collection` используем в точности тот же код, который приведен выше в примере с объектом `NightClub`, поэтому, если вы уже создали этот класс, вам не нужно вносить какие-либо изменения. Если вы еще не создали файл, содержащий этот класс, сделайте это сейчас.

Из диаграммы UML видно, что класс `Course` предельно прост: он содержит только имя, код курса и числовой идентификатор ID. Код курса — это идентификатор, например CS101, для класса начального курса по компьютерным наукам. Имя — это полное название курса. ID — это числовой первичный ключ из базы данных. В реальном приложении вы, вероятно, захотите иметь доступ к перечню курсов, предваряющих данный (еще одна коллекция), к расписанию занятий по курсу, к информации о преподавателе и т.д. Все это оставим для самостоятельного выполнения в качестве упражнения.

Рассмотрим следующий код.

```
<?php
class Course {
    private $_id;
    private $_courseCode;
    private $_name;

    function __construct($id, $courseCode, $name) {
        $this->_id = $id;
        $this->_courseCode = $courseCode;
        $this->_name = $name;
    }

    public function getName() {
        return $this->_name;
    }

    public function getID() {
        return $this->_id;
    }
    public function getCourseCode() {
        return $this->_courseCode;
    }

    public function __toString() {
        return $this->_name;
    }
}
?>
```

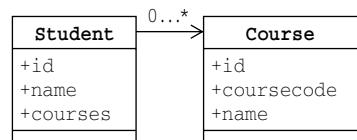


Рис. 5.3.

В соответствии с диаграммой классов курс имеет два свойства: \$id и \$name.

Поскольку в этой части приложения не требуется изменять свойства \$id или \$name данного класса, для хранения этих значений необходимо использовать закрытые переменные-члены. Для доступа к ним предназначены методы `getID()` и `getName()`. При создании частей приложения, в которых потребуется изменять эти значения (средства администрирования для курсов), в данный класс можно добавить методы `setId()` и `setName()`. Эти методы должны содержать некоторые механизмы обеспечения безопасности, чтобы гарантировать защиту от несанкционированного изменения свойств курса посторонними.

В приведенном фрагменте также реализован метод `_toString()`. Этот “магический” метод можно определить для любого объекта в PHP 5. Он позволяет написать следующий код.

```
print "Этот курс называется " . $objCourse . "<br>"
```

Здесь `$objCourse` явно не строка, но поскольку для данного объекта определен специальный метод `_toString()`, PHP для преобразования объекта в строку использует возвращаемое значение этого метода.

Так как коллекция курсов `courses` состоит только из объектов `Course`, создадим подкласс класса `Collection`, называемый `CourseCollection`. Перекроем метод `addItem()`, включив в него указание типа, которое позволит добавлять в коллекцию только объекты `Course`.

```
<?php
class CourseCollection extends Collection {
    public function addItem(Course $obj, $key = null) {
        parent::addItem($obj, $key);
    }
}
?>
```

Это позволяет PHP выполнять проверку соответствия типов и гарантировать, что в коллекции `$courses` содержатся только объекты `Course`.

Класс `Student` содержит свойства `$id` и `$name`, которые представляют уникальный идентификатор и имя данного студента. В этот класс необходимо также добавить свойство `$courses`, представляющее коллекцию курсов, на которые зарегистрировался данный студент.

```
<?php
class Student {
    private $_id;
    private $_name;
    public $courses;

    public function __construct($id, $name) {
        $this->_id = $id;
        $this->_name = $name;

        $this->courses = new CourseCollection();
        $this->courses->setLoadCallback('_loadCourses', $this);
    }

    public function getName() {
        return $this->_name;
    }

    public function getID() {
```

```

    return $this->_id;
}

private function _loadCourses(Collection $col) {
    $arCourses = StudentFactory::getCoursesForStudent($this->_id, $col);
}
public function __toString() {
    return $this->_name;
}
?>

```

Подобно классу `Course`, свойства `$id` и `$name` реализованы как закрытые переменные-члены, доступ к которым осуществляется через функции `getID()` и `getName()`. Пока оставим переменную-член `$courses` в открытой области доступа. Обратите внимание на создание метода `_toString()` для этого класса. Наличие этого метода для любого класса несколько упрощает отображение объекта в виде общепринятого строкового представления. В данном случае это касается имени студента.

В конструкторе вызывается метод `setLoadCallback()`, которому передается имя метода заполнения коллекции (загружающего метода) `_loadCourses()` и ссылка на объект, для которого вызывается этот метод, т.е. `$this`.

Метод `_loadCourses()` в качестве единственного параметра получает объект `Collection` (как требуется для функции `setLoadCallback()`) и использует статический метод класса `StudentFactory` (который еще не обсуждался). При вызове этого метода необходимо указать идентификатор ID студента и передать объект `Collection`, в который будут загружены курсы.

Класс `StudentFactory` содержит два статических метода, которые выполняют все действия по извлечению информации о студентах и курсах из базы данных. Для этого приложения необходимо создать в базе данных три таблицы: `student`, `course` и `studentcourse`. Назначение первых двух таблиц совершенно очевидно — хранение базовой информации о студентах и курсах. Третья таблица содержит два поля: `studentid` и `courseid` и служит для связи между студентами и курсами.

Ниже приведены операторы SQL для работы с базой данных PostgreSQL (предпочитаемый авторами сервер баз данных). Однако при незначительном изменении их можно использовать и для большинства других реляционных баз данных.

```

CREATE TABLE "student" (
    "studentid" SERIAL NOT NULL PRIMARY KEY,
    "name" varchar(255)
);

CREATE TABLE "course" (
    "courseid" SERIAL NOT NULL PRIMARY KEY,
    "coursecode" varchar(10),
    "name" varchar(255)
);

CREATE TABLE "studentcourse" (
    "studentid" integer,
    "courseid" integer,
    CONSTRAINT "fk_studentcourse_studentid"
        FOREIGN KEY ("studentid")
            REFERENCES "student"("studentid"),
    CONSTRAINT "fk_studentcourse_courseid"
        FOREIGN KEY ("courseid")
            REFERENCES "course"("courseid")
);

```

```
CREATE UNIQUE INDEX "idx_studentcourse_unique"
    ON "studentcourse"("studentid", "courseid");
```

Удостоверьтесь в создании внешних ключей и не забудьте обеспечить уникальность индексов для таблицы `studentcourse`. Студент не должен иметь возможности дважды зарегистрироваться на одном и том же курсе, и это требование обеспечивается на уровне базы данных.

Введите некоторые примеры данных в каждую из трех таблиц. Для этого можно воспользоваться следующими операторами.

```
INSERT INTO "student"(name) VALUES('Боб Смит');           -- ID студента 1
INSERT INTO "student"(name) VALUES('Джон До');           -- ID студента 2
INSERT INTO "student"(name) VALUES('Джейн Бейкер');       -- ID студента 3

INSERT INTO "course"("coursecode", "name")
VALUES('CS101', 'Основы компьютерных наук');      -- ID курса 1
INSERT INTO "course"("coursecode", "name")
VALUES('HIST369', 'История Великобритании 1945-1990'); -- ID курса 2
INSERT INTO "course"("coursecode", "name")
VALUES('BIO546', 'Генетика');                      -- ID курса 3

INSERT INTO "studentcourse"("studentid", "courseid") VALUES(1, 1);
INSERT INTO "studentcourse"("studentid", "courseid") VALUES(1, 2);
INSERT INTO "studentcourse"("studentid", "courseid") VALUES(1, 3);
INSERT INTO "studentcourse"("studentid", "courseid") VALUES(2, 1);
INSERT INTO "studentcourse"("studentid", "courseid") VALUES(2, 3);
INSERT INTO "studentcourse"("studentid", "courseid") VALUES(3, 2);
```

Последние шесть операторов необходимо уточнить в зависимости от значений `studentid` и `courseid`, получаемых в результате выполнения оператора `insert` для таблиц `student` и `course`.

Создав таблицы базы данных и заполнив их некоторыми примерами, можно приступить к написанию и тестированию класса `StudentFactory`. Этот класс содержит два статических метода: `getStudent()` и `getCourseForStudent()`. Первый из них отвечает за создание объекта `Student` и назначение ему идентификатора `ID`. Второй заполняет коллекцию курсов для данного студента. В приведенном ниже коде мы не будем демонстрировать использование конкретных функций базы данных. Поэтому вместо вызова реальных функций базы данных здесь приводится некоторый псевдокод. В главе 8 “Уровни абстракции базы данных” будет показано, как использовать абстрактные функции базы данных `PEAR::DB`, но пока псевдокод можно произвольным образом заменить на явные вызовы функций `pg_connect()`, `pg_query()` и т.д. (или любые другие функции PHP для выбранного типа базы данных).

```
<?php
class StudentFactory {

    public static function getStudent($id) {
        $sql = "SELECT * from \"student\" WHERE \"studentid\" = $id";
        $data = $db->select($sql); //псевдокод. Он показывает,
                                // что после обработки запроса возвращается
                                // массив с заполненными строками.
        if(is_array($data) && sizeof($data)) {
            return new Student($data[0]['studentid'], $data[0]['name']);
        } else {
            throw new Exception("Студент $id не существует.");
        }
    }
}
```

```

public static function getCoursesForStudent($id, $col) {
    $sql = "SELECT \"course\".\"courseid\",
              \"course\".\"coursecode\",
              \"course\".\"name\"
        FROM \"course\", \"studentcourse\" WHERE
              \"course\".\"id\" = \"studentcourse\".\"courseid\" AND
              \"studentcourse\".\"studentid\" = $id";
    $data = $db->select($sql); //некоторый псевдокод в getStudent()
    if(is_array($data) && sizeof($data)) {
        foreach($data as $datum) {
            $objCourse = new Course($datum['courseid'], $datum['coursecode'],
                                   $datum['name']);
            $col->addItem($objCourse, $objCourse->getCourseCode());
        }
    }
}
?>

```

Метод `getStudent()` возвращает новый объект `Student`, заполненный соответствующими данными. Если в базе данных не существует студента с заданным идентификатором ID, генерируется исключение. Все вызовы функции `StudentFactory::getStudent()` должны помещаться в блок `try...catch`.

Метод `getCoursesForStudent()` получает коллекцию `$courses` в результате вызова метода `_loadCourses()`. Запрос к базе данных выбирает всю информацию из таблицы `course`, неявно применяя оператор `JOIN` для таблицы `studentcourse`, чтобы получить курсы, связанные с данным студентом.

На основе информации, полученной из базы данных, создаются новые объекты `Course`, которые добавляются в коллекцию. Обратите внимание на использование значения `studentcourse` в качестве ключа при добавлении нового курса в коллекцию `$courses`.

Поскольку все объекты в PHP 5 передаются по ссылке, содержимое коллекции можно изменить в методе `getCoursesForStudent()` и проверить, какие изменения произойдут в переменной `$courses` объекта `Student`. Если фраза “передаются по ссылке” ни о чем вам не говорит, обратитесь к главе 4 “Шаблоны проектирования”, в которой описано различие между передачей параметров в функцию по ссылке и по значению.

Осталось испытать этот код в действии.

```

<?php
$studentID = 1; //корректный ID студента из таблицы student

try {
    $objStudent = StudentFactory::getStudent($studentID);
} catch (Exception e) {
    die("Студент #$studentID в БД не существует!");
}

print $objStudent .
    ($objStudent->courses->exists('CS101') ? ' в ' : ' не в ') .
    'текущем списке курса CS101';
//выводит строку "Боб Смит в текущем списке курса CS101"
?>

```

В данном приложении необходимо предусмотреть корректное завершение операции, если данный студент не найден. Надеемся, вы понимаете, почему нужно использовать блок `try...catch`, чтобы избежать внезапного завершения работы приложения.

После получения `$objStudent` данная программа выводит простое сообщение, позволяющее узнать, зарегистрировался ли данный студент на изучение курса CS101.

Перед вызовом метода `exists()` коллекция курсов корректно заполняется с помощью класса `StudentFactory` на основе информации из базы данных. Поскольку код курса используется в качестве ключа при добавлении объекта `Course` к коллекции, его (код курса ‘CS101’) можно использовать в качестве параметра метода `exists()`.

Усовершенствование класса Collection

“Улучшить класс `Collection`? Это невозможно! Как можно сделать его более мощным, более практичным и более привлекательным?” — спросите вы. Проблема в том, что вы еще не знаете простого способа работы со всей коллекцией, например, для отображения списка курсов. Сейчас вы можете обратиться только к тем элементам, о существовании которых точно известно. Очевидно, что содержание коллекции не всегда известно до начала ее использования, поэтому необходимо написать следующий код.

```
<?php
$objStudent = StudentFactory::getStudent(1);
foreach($objStudent->courses as $objCourse) {
    print $objStudent . ' в текущем списке ' . $objCourse . '<br>\n';
}
?>
```

Если вы сейчас попытаетесь запустить код, то получите сообщение об ошибке или вообще ничего не увидите (в зависимости от заданного уровня ошибок `error_reporting`), потому что `$objStudent->courses` является не массивом, а коллекцией. В PHP4 в качестве операндов оператора `foreach` можно было использовать только массивы. В PHP5 существуют два новых встроенных интерфейса — `Iterator` и `IteratorAggregate`, позволяющих использовать оператор `foreach` для объектов, реализующих эти интерфейсы.

Эти два интерфейса будут рассмотрены в следующей главе, поэтому ознакомьтесь с ней и постараитесь сделать класс `Collection` еще более полезным!

Резюме

Класс `Collection` — очень удобная объектно-ориентированная альтернатива для традиционных массивов, которую можно использовать практически в каждом разрабатываемом приложении. Этот класс обеспечивает корректное управление элементами коллекции и согласованный программный API-интерфейс, упрощающий написание кода, использующего данный класс.

Подкласс `Collection` и перекрытый метод `addItem()`, включающий указание типа в качестве первого параметра, позволяет задавать тип объектов, добавляемых в коллекцию. Этот метод обеспечивает дополнительный уровень проверки ошибок.

Заключая все вызовы функций `addItem()`, `getItem()`, `removeItem()` в блоки `try...catch`, вы обеспечиваете полный контроль над ошибками, которые могут возникать при вызове этих методов. Обязательно обработайте эти ошибки с помощью более осмысленного метода, чем вызов метода `die()`, приведенного в примерах.

Обратный вызов — очень мощный прием, с помощью которого можно реализовать позднее инстанцирование. Это позволяет экономить системные ресурсы и работать только с теми данными, которые действительно необходимы. Имея полную иерархию объектов, обратный вызов можно использовать для автоматического управления созданием дочерних объектов и избежания создания тех объектов, которые совершенно не нужны для текущей работы приложения. Постарайтесь понять этот прием и использовать его в других ситуациях.

6

Класс CollectionIterator

В предыдущей главе рассматривался класс `Collection`, объектно-ориентированная оболочка для управления массивами объектов. Этот класс представляет собой чрезвычайно мощный и гибкий компонент повторно используемого набора инструментов. Однако в нем все же отсутствуют некоторые функции. В частности, класс `Collection` не позволяет итерационно пройти по всем его элементам. Например, в приложении для университета из предыдущей главы использовался класс `Student` с переменной-членом `courses`, которая представляла собой коллекцию объектов. Вполне возможно, что в таком приложении совсем нeliшней окажется возможность отображения списка всех курсов для определенного студента. Это проще всего осуществить с помощью следующего фрагмента кода.

```
<?php
    $objStudent = new Student(12345);
    foreach($objStudent->classes as $objClass) {
        print $objClass . "\n";
    }
?>
```

Если у вас имеется опыт работы на языке PHP версии 4, то вам наверняка известен оператор `foreach`, используя который можно без особых проблем перебрать все элементы массива. В качестве параметра этого оператора `foreach` в PHP 4 допускалось использование только массива, а применение данных любого другого типа приводило к возникновению ошибки времени выполнения.

В PHP 5 оператор `foreach` позволяет работать и с объектами. При совместном использовании встроенных интерфейсов `Iterator` и `IteratorAggregate` можно перебрать элементы объекта класса `Collection`, которые сами тоже являются объектами.

В этой короткой главе вы увидите, как использовать эти интерфейсы и обеспечить возможность прохода по коллекции объектов традиционным способом, т.е. с помощью оператора `foreach`, который ранее был применим лишь для работы с традиционными массивами.

Интерфейс Iterator

Во встроенным абстрактном интерфейсе `Iterator` языка PHP 5 определено пять методов, позволяющих применять оператор `foreach` для обработки параметра, который не является массивом. Вот определение этого интерфейса.

```
<?php
/**
 * Traversable - это пустой интерфейс
 */
interface Traversable {}

interface Iterator implements Traversable {
    /**
     * Перемещает итератор на первый элемент.
     */
    function rewind();

    /**
     * Возвращает текущий элемент.
     */
    function current();

    /**
     * Возвращает ключ текущего элемента.
     */
    function key();

    /**
     * Перемещает указатель на следующий элемент.
     */
    function next();

    /**
     * Проверяет, существует ли текущий элемент после
     * обращения к методам rewind() или next().
     */
    function hasMore();
}

?>
```

Первым определен интерфейс `Traversable`. Это простой абстрактный интерфейс, который нельзя реализовать в программном коде. Он используется самим модулем PHP для определения допустимых конструкций оператора `foreach`. При реализации классов, которые будут обрабатываться с помощью оператора `foreach`, нужно реализовать интерфейс `Iterator`, который в свою очередь реализует интерфейс `Traversable`.

Оба интерфейса, `Traversable` и `Iterator`, являются встроенными. Поэтому повторно объявлять их в программном коде не нужно. В противном случае будет сгенерирована ошибка `Fatal error: Cannot redeclare class` (Фатальная ошибка: нельзя повторно объявить класс).

Обратите внимание на то, что большинство методов интерфейса `Iterator` имеет те же имена, что и встроенные функции, используемые для работы с массивами. Функция `rewind()` устанавливает внутренний указатель массива на его первый элемент. Функция `current()` возвращает элемент массива, на который этот указатель

ссылается в данный момент. Функция `key()` возвращает ключ текущего элемента, а `next()` перемещает указатель на следующий элемент. При итерационном проходе по объекту `Iterator` эти методы применяются аналогичным образом. Другими словами, при расширении интерфейса `Iterator` нужно определить скрытую (`private`) переменную-член, с помощью которой отслеживалось бы текущее положение указателя. Функция `hasMore()` должна возвращать истинное или ложное значение в зависимости от того, является ли текущий элемент последним в списке.

Следующий фрагмент кода демонстрирует, какие действия выполняются при использовании оператора `foreach` для обработки класса, который реализует интерфейс `Iterator`.

```
<?php

// эквивалентно foreach($objIt as $key=>$member)
$objIt = new MyIterator();
for ($objIt->rewind(); $objIt->hasMore(); $objIt ->next()) {
    $key = $objIt->key();
    $member = $objIt->current();
}
?>
```

После анализа этого фрагмента вы должны лучше понять назначение интерфейса `Iterator`. Теперь самое время применить его на практике. В последующих примерах для итерационного прохода по объекту класса `Collection` будет использоваться класс `CollectionIterator`.

Класс CollectionIterator

Вспомните, что как и обычный массив, класс `Collection` хранит элементы в позиции, которая определяется либо строковым ключом, либо автоматически выбранным целым числом. В предыдущей главе был создан метод `keys()`, однако на практике он еще не использовался. Вспомните, что `keys()` — это метод, который возвращает массив всех ключей коллекции. Этот метод будет применяться классом `CollectionIterator` для получения списка элементов коллекции и их итерационной обработки.

Ниже представлен чрезвычайно простой код из файла `class.collectioniterator.php`.

```
<?
class CollectionIterator implements Iterator {

    private $_collection;
    private $_currIndex = 0;
    private $_keys;

    function __construct(Collection $objCol) {
        $this->_collection = $objCol;
        $this->_keys = $this->_collection->keys();
    }

    function rewind() {
        $this->_currIndex = 0;
    }

    function hasMore() {
        return $this->_currIndex < $this->_collection->length();
    }

    function key() {
        return $this->_keys[$this->_currIndex];
    }

    function next() {
        $this->_currIndex++;
    }

    function current() {
        return $this->_collection->get($this->_currIndex);
    }

    function valid() {
        return $this->_currIndex < $this->_collection->length();
    }
}
```

```

function key() {
    return $this->_keys[$this->_currIndex];
}

function current() {
    return $this->_collection->getItem(
        $this->_keys[$this->_currIndex]);
}

function next() {
    $this->_currIndex++;
}
?>

```

С учетом того, что класс реализует интерфейс `Iterator`, нужно объявить все методы, определенные в этом интерфейсе. Переменная-член `$_collection` содержит ссылку на обрабатываемый объект `Collection`. `$_currIndex` — это внутренняя переменная, содержащая указатель на элемент коллекции, который используется в методах. Это значение фактически является указателем в массиве `$_keys`.

В массиве `$_keys` хранятся значения `Collection::keys()`, которые будут использоваться для извлечения требуемого члена коллекции. При каждом вызове метода `next()` значение `$_currIndex` увеличивается на единицу. При вызовах методов `current()` и `key()` для определения ключа элемента используется значение `$_currIndex`. Поскольку реальные ключи элементов коллекции не всегда являются числовыми, то для итерационного прохода по этим элементам необходимо иметь в своем распоряжении некоторый упорядоченный по номерам список. Для этого можно воспользоваться методом `Collection::keys()`, получить массив, проиндексированный по номерам элементов `Collection`, и использовать значения этого массива для получения реального ключа элемента коллекции. Следующий пример поможет лучше разобраться с этим приемом.

```

<?php
require_once ("class.collectioniterator.php");
require_once ("class.collection.php");
/* Добавление нескольких элементов в коллекцию */
$objCol = new Collection();
$objCol->addItem(new Foo(), "foo1"); // ключ = "foo1"
$objCol->addItem(new Bar(), "bar1"); // ключ = "bar1"
$objCol->addItem(new FooBar()); // ключ = 0
$objCol->addItem(new FooBar()); // ключ = 1

/* Инстанцирование объекта CollectionIterator */
$objIt = new CollectionIterator($objCol);

/* Переменная-член $objIt->keys теперь выглядит так:
 * $objIt->keys[0] = "foo1";
 * $objIt->keys[1] = "bar1";
 * $objIt->keys[2] = 0;
 * $objIt->keys[3] = 1;
 */

$objIt->rewind(); // $_currIndex = 0;

$objFoo = $objIt->current(); // $_currIndex = 0, текущий ключ = "foo1"
// $objFoo = первый объект Foo

$objIt->next(); // $_currIndex = 1;

```

```

$objBar = $objIt->current();      // $_currIndex = 1, текущий ключ = "bar1"
                                    // $objBar = первый объект Bar
$objIt->next();                  // $_currIndex = 2;
                                    // $objBar - первый объект FooBar
?>

```

Из этого примера хорошо видно, как для отслеживания всего процесса обработки элементов коллекции используется массив `$_keys` и переменная `$_currIndex`.

В приведенном фрагменте для прохода по элементам коллекции еще нельзя воспользоваться оператором `foreach`. (В данный момент его можно применить только для работы с объектом `CollectionIterator`.) Чтобы обеспечить эту возможность, сначала нужно познакомиться с еще одним встроенным абстрактным интерфейсом.

Интерфейс IteratorAggregate

Язык PHP версии 5 позволяет использовать оператор `foreach` для работы с реализациями интерфейса `Iterator` или интерфейса `IteratorAggregate`. В классе, реализующем интерфейс `IteratorAggregate`, содержатся элементы, которые можно пройти с помощью интерфейса `Iterator`. Другими словами, элементы класса `Collection` можно итерационно обработать с помощью класса `CollectionIterator`. Для того чтобы оператор `foreach` можно было применить для объекта `Collection`, нужно модифицировать этот класс так, чтобы он реализовывал простой интерфейс со следующим определением.

```

<?php
/**
 * Интерфейс для создания внешнего итератора.
 */
interface IteratorAggregate implements Traversable
{
    /**
     * Возвращает итератор для объекта.
     */
    function getIterator();
}
?>

```

Как и интерфейсы `Iterator` и `Traversable`, интерфейс `IteratorAggregate` также является встроенным. Поэтому в разрабатываемом коде его не нужно повторно объявлять.

В интерфейсе `IteratorAggregate` определен один метод `getIterator()`. Для класса, который реализует этот интерфейс, функция `getIterator()` возвращает объект `Iterator`, позволяющий итерационно проходить по элементам этого класса. Модифицировав класс `Collection` таким образом, чтобы он реализовывал этот интерфейс, можно обеспечить возможность применения оператора `foreach` к этому классу напрямую.

Модифицируем объявление класса и включим в него реализацию метода, требуемого для интерфейса `IteratorAggregate`.

```
<?php
```

```
class Collection implements IteratorAggregate {
```

```
// ...исходная часть описания класса Collection
// удалена для краткости

public function getIterator() {
    $this->_checkCallback();
    return new CollectionIterator($this);
}

?
```

Вспомните, что в классе `Collection` реализован механизм позднего инстанцирования. Поэтому перед возвратом значения из функции `getIterator()` не забудьте проверить, вызывалась ли функция обратного вызова. Перед выполнением итераций нужно удостовериться в том, что все данные были загружены в коллекцию.

Теперь с помощью созданного класса `CollectionIterator` и модифицированного класса `Collection`, реализующего интерфейс `IteratorAggregate`, можно выполнить фрагмент кода, приведенного в конце предыдущей главы.

```
<?php
$objStudent = StudentFactory::getStudent(12345); // студент #12345
foreach ($objStudent->courses as $key => $objCourse) {
    print $objStudent . ' уже внесен в список ' . $objCourse . "<br>\n";
}
?>
```

С помощью интерпретатора PHP функции вызываются следующим образом.

```
<?php
$objStudent = StudentFactory::getStudent(12345);
$itCourses = $objStudent->courses ->getIterator();

for($itCourses->rewind(); $itCourses->hasMore(); $itCourses->next()) {
    $key = $itCourses->key();
    $objCourse = $itCourses->current();
    print $objStudent . ' уже внесен в список ' . $objCourse . "<br>\n";
}
?>
```

Если необходимо создать итераторы для других классов, очень важно разобраться с тем, как вызываются методы интерфейсов `Iterator` и `IteratorAggregate`. Если параметром оператора `foreach` является объект `IteratorAggregate`, вызывается метод `getIterator()`. Внутренний указатель объекта `Iterator` сбрасывается с помощью метода `rewind()`. Хотя в коде используются и другие элементы (`hasMore()`), достаточно сказать, что создаются переменные для хранения текущего ключа и текущего элемента, а затем выполняется остальная часть блока `foreach`. Указатель сдвигается вперед (`next()`), и выполнение цикла продолжается до тех пор, пока не будет достигнут последний элемент (т.е. когда метод `hasMore()` не вернет значение `false`).

Защита содержимого объекта `Iterator` с помощью оператора `clone`

На данный момент код можно считать завершенным, если его планируется использовать только для отображения элементов `Collection`. Однако что произойдет, если потребуется изменить содержимое коллекции “на лету”, т.е. внутри цикла `foreach`? В приведенном ниже коде в цикле просматривается список курсов для дан-

ного студента и выполняется поиск курсов, для прохождения которых требуется дополнительная оплата (например, на содержание лаборатории). Если в данный момент студент не в состоянии оплатить обучение, его следует удалить из списка студентов для данного курса и вывести сообщение об ошибке.

```
<?php
function checkCourseFees (Student $objStudent) {
    foreach($objStudent->courses as $key => $objCourse) {
        if( $objCourse->hasSpecailFee &&
            !$objStudent->inGoodStanding() ) {
            $objStudent->courses->removeItem($key);
            print("Извините, нам пришлось удалить Вас " .
                "из списка слушателей этого курса");
        }
    }
?>
```

Этот код кажется безопасным и корректным. На первый взгляд, при проходе по коллекции и удалении ее элемента ничего опасного не происходит. Однако на самом деле этот код может приводить к возникновению неожиданных ошибок. Для того чтобы разобраться с этой проблемой, нужно заглянуть чуть-чуть поглубже.

Предположим, что объект студента связан с тремя курсами, двумя обычными и одним платным.

```
$objStudent->courses->addCourse(new Course(123)); //бесплатный, ключ = 0
$objStudent->courses->addCourse(new Course(987)); //платный, ключ = 1
$objStudent->courses->addCourse(new Course(456)); //бесплатный, ключ = 2
```

При передаче параметра `$objStudent->courses` в оператор `foreach` вызывается метод `getIterator()`. При вызове в конструкторе класса `CollectionIterator` метода `$_collection->keys()` создается внутренний массив `$_keys`. При проходе по первому элементу значение переменной `$_currIndex` равно 0. Далее извлекается первый курс. Пока не возникло никаких проблем. На второй итерации значение `$_currIndex` равно 1. Этот курс является платным, поэтому он удаляется из коллекции. Именно в этот момент и возникает проблема.

Поскольку коллекция была передана в объект `Iterator` по ссылке, то вызов метода `$objStudent->courses->removeItem()` оказывает на эту коллекцию непосредственное влияние. Теперь переменная-член `$collection` содержит только два элемента с ключами 0 и 1. Попытка извлечения третьего ключа со значением 2 завершится неудачей. Для устранения этой проблемы нужно сделать две вещи.

Во-первых, не модифицируйте коллекцию в процессе прохода по ее элементам. Все операции добавления и удаления должны выполняться за пределами цикла `foreach`. Во-вторых, при передаче коллекции в качестве параметра передавайте ее копию, а не ссылку на нее (как это было реализовано выше).

При передаче параметра в конструктор класса, реализующего интерфейс `Iterator`, в методе `getIterator()` класса `Collection` используйте оператор `clone`. Он позволяет создать копию этого параметра. Действие оператора `clone` противоположно модификатору `&`. С учетом вышесказанного метод `Collection::getIterator()` можно модифицировать следующим образом.

```
public function getIterator(){
    $this->_checkCallback();

    return new CollectionIterator(clone $this);

}
```

При использовании описанных рекомендаций будет создана копия текущего объекта, которая и будет передана в класс `Iterator`. Эта копия будет создана путем инстанцирования новой коллекции (без вызова конструктора) и копирования в нее всех открытых (`public`) и закрытых (`private`) переменных-членов из исходного объекта. Если при клонировании объекта требуется выполнить какие-либо специальные действия, которые должны выполняться по умолчанию, можно воспользоваться методом `__clone()`. Обратите внимание на то, что функция `__clone()` является скрытой (`private`). Поэтому при ее непосредственном вызове будет генерирована ошибка. Для копирования объекта всегда используйте оператор `clone`.

Резюме

Интерфейсы `Iterator` и `IteratorAggregate` позволяют использовать оператор `foreach` для итерационного доступа к объектам, в которых содержатся дочерние элементы. Несмотря на то, что интерфейс `Iterator` можно реализовать и непосредственно в самом классе `Collection`, при таком подходе интерфейс этого класса окажется слишком громоздким. Разместив функции итератора в отдельном классе, можно существенно повысить ясность всего программного кода.

Если нужно получить копию объекта, используйте оператор `clone`. В ранних версиях документации по языку PHP 5 содержался пример кода, в котором метод `__clone()` вызывался напрямую. В настоящее время подобные вызовы уже не поддерживаются. Вместо этого нужно использовать оператор `clone`. Если при копировании объектов необходимо выполнять какие-либо определенные действия, то их описание следует разместить в методе `__clone()`.

Классы `Collection` и `CollectionIterator` предоставляют мощный объектно-ориентированный механизм для управления группами объектов. Хотя для совместного использования этих двух классов понадобится написать изрядное количество кода, это придется осуществить только один раз. Впоследствии его можно многократно применять во многих приложениях.

7

Класс GenericObject

Сейчас вы уже хорошо разбираетесь с идеей объектно-ориентированного программирования, а также оценить все его преимущества. Действительно, объектно-ориентированный подход позволяет разрабатывать легко поддерживаемый, понятный и логически упорядоченный код, который хорошо согласуется с представлением о приложении даже самого неопытного пользователя. Несмотря на то, что при использовании такого подхода от разработчиков приложения потребуются большие усилия, получаемые при этом преимущества станут прекрасно понятны, если кому-либо понадобится еще раз взглянуть на ранее разработанный программный код.

Однако одним из недостатков ООП является то, что для полного удовлетворения всех требований иногда приходится жертвовать производительностью. Например, во многих Web-приложениях используются простые страницы, позволяющие редактировать некоторые данные, например информацию о заказчике, продукте, заказе, покупателе и т.д. В этом случае традиционный подход оказывается совершенно прямолинейным, поскольку для решения этой задачи вполне достаточно HTML-формы с единственным оператором SQL UPGRADE для внесения изменений. В отличие от этого, объектно-ориентированный подход может показаться просто пугающим. Для решения той же задачи на его основе понадобится реализовать класс *Customer* с большим количеством методов только для того, чтобы внести изменения в соответствующие свойства в базе данных. Кроме того, реальная проблема также будет связана и с тем, что большие фрагменты кода для каждого класса, который является представлением заказчика, продукта, заказа и клиента, соответственно, будут многократно повторяться.

К счастью, существует и другой путь, связанный с применением класса *GenericObject*. В этой главе вы познакомитесь с этим классом, а также с его наследником, *GenericObjectCollection*. Эти классы позволяют наполовину сократить время разработки, добиться полного соответствия принципам объектно-ориентированного подхода и воспользоваться всеми его преимуществами.

Класс GenericObject

Класс `GenericObject` является абстрактным суперклассом. В программном коде его нельзя инстанцировать напрямую. Это можно осуществить лишь путем расширения класса `GenericObject` или использования подклассов, наследующих его свойства и методы.

Если термин “наследование” оказался не очень понятным, обратитесь к главе 3 “Объектный подход в действии”, где этот вопрос рассматривался более детально.

Класс `GenericObject` позволяет представить в виде класса отдельную строку (или группу строк) из одной таблицы базы данных и таким образом обеспечить ее обработку с использованием объектно-ориентированного подхода.

Когда нужно использовать класс `GenericObject`

К основным примерам таблиц, которые могут использоваться в приложении и хорошо подходить для объектного представления с помощью класса `GenericObject`, относятся `user`, `customer`, `product`, `order` и т.д.

Однако такое представление лучше не использовать для таблиц, которые каким-либо способом просто связывают две сущности. Например, в приложении может содержаться таблица `order_product` со столбцами `order_id`, `product_id` и `quantity`. Если подобная таблица используется, то можно только порадоваться. В конце концов, это отличный пример базы данных, приведенной к третьей нормальной форме (более подробную информацию можно найти по адресу <http://databases.about.com/library/glossary/bldef-3nf.htm>).

Такие таблицы не являются представлением сущностей в чистом виде, поскольку в них нет свойств, значения которых можно считывать или устанавливать, даже несмотря на то, что в них могут содержаться квалификиаторы, такие как поле `quantity` из предыдущего примера.

Что позволяет делать класс `GenericObject`

Функциональность класса `GenericObject` не является чисто символической. Кроме того, он также не является и частью некоторого набора инструментов, предназначенного исключительно для обеспечения точного соответствия принципам объектно-ориентированного программирования. Это на самом деле полезный инструмент.

Любую заданную строку (скажем, 314-ю) произвольной таблицы (скажем, таблицы `order`) вы можете представить в виде объекта, расширяющего класс `GenericObject`. Это позволяет вам быстро и эффективно выполнять следующие действия.

- ❑ Считывать значение любого заданного свойства (столбца) этого объекта (строки).
- ❑ Устанавливать значение любого заданного свойства (столбца) этого объекта (строки).
- ❑ Сохранять изменения в этом объекте (строке).

Что еще более важно, можно также создавать новые объекты определенной сущности, устанавливать значения ее свойств, а затем сохранять изменения аналогичным образом (т.е. с помощью объектного представления).

Рассмотрим пример объекта *customer* (покупатель). Допустим, в схеме базы данных имеется таблица *customer* со столбцами *id*, *first_name*, *last_name*, *addr_line_1* и т.д.

Предположим, нужно получить имя заказчика с идентификатором 3139. Для этого потребуется выполнить следующие действия.

- Инстанцировать объект *customer* (класс которого расширяет класс *GenericObject*) с идентификатором 3139.
- Получить значение свойства *first_name* этого объекта.

Вот и все.

А что, если после этого понадобится изменить имя с *John* на *Jane*? Что нужно для этого сделать?

- Инстанцировать объект *customer* (класс которого расширяет класс *GenericObject*) с идентификатором 3139.
- Задать для свойства *first_name* этого объекта значение *Jane*.
- Сохранить внесенные изменения.

Предположим, нужно создать новый объект *customer*. Для этого достаточно выполнить примерно те же действия.

- Инстанцировать объект *customer* (класс которого расширяет класс *GenericObject*) без параметров.
- Для свойства *first_name* этого объекта задать значение *Jane*.
- Сохранить внесенные изменения.
- Получить идентификатор (*id*) только что созданного объекта *customer* для дальнейшего использования.

В последнем примере при инстанцировании не задается никаких идентифицирующих параметров. Тем самым объекту *GenericObject* сообщается о том, что создается новый экземпляр.

После сохранения внесенных данных идентификатор, размещенный в новой строке базы данных, станет доступным для обращения к соответствующему свойству.

Следует отметить, что все приведенные выше примеры можно легко реализовать и с помощью обычного процедурного кода с внедренными в него операторами SQL. Вполне возможно, что такая реализация окажется несколько быстрее, чем при использовании класса *GenericObject*. Такой недостаток присущ многим объектно-ориентированным приложениям, однако получаемые при этом преимущества, в частности ясность программного кода, этого стоят. Несколько часов, которые придется дополнитель но потратить на объектную реализацию, впоследствии позволят сэкономить гораздо больше времени при поддержке или доработке системы.

Преимущества использования

Для использования класса *GenericObject* должно выполняться одно важное условие. Сущности в таблицах базы данных должны иметь уникальный идентификатор, который автоматически возрастал бы и был их первичным ключом.

Как правило, именно так и есть. При правильном проектировании базы данных разработчики всегда приходят к необходимости использования первичного ключа. Для обеспечения высокой производительности первичный ключ чаще всего является числовым.

В качестве хорошего примера базы данных (при использовании сервера PostgreSQL), взаимодействие с которой удобно реализовать с помощью класса `GenericObject`, можно привести следующую.

```
CREATE TABLE "user" (
    "id" SERIAL PRIMARY KEY NOT NULL,
    "username" character varying(32),
    "first_name" character varying(64),
    "last_name" character varying(64)
);
```

Как можно увидеть, в таблице имеется столбец `id`, который однозначно определяет каждого пользователя, а также его свойства `username`, `first_name` и `last_name`, которые можно считывать или записывать.

Гораздо хуже для совместного использования с классом `GenericObject` подходит следующая база данных.

```
CREATE TABLE "airport" (
    "iata_airport_code" character(3) PRIMARY KEY NOT NULL,
    "airport_name" character varying(128),
    "airport_city" character varying(128),
);
```

Этот пример таблицы является вполне жизнеспособным. Аэропорты довольно часто однозначно определяются их трехсимвольным IATA-кодом (LAX для Los Angeles International, JFK для Kennedy Airport и т.д.). Однако такую структуру не очень удобно использовать вместе с классом `GenericObject`, если не добавить столбец с числовыми идентификаторами, которые должны быть первичным ключом. Причина этого заключается в том, что объект `GenericObject` ожидает, что новые объекты будут иметь уникальный идентификатор, сгенерированный базой данных. При этом пользователи не должны иметь возможность задать его самостоятельно. Трехсимвольный код аэропорта система управления базой данных сгенерировать не сможет, а числовой идентификатор — без каких бы то ни было проблем.

Вот еще один пример таблицы, которая не очень подходит для совместного использования с классом `GenericObject`.

```
CREATE TABLE "user_group" (
    "user_id" int2,
    "group_id" int2,
    PRIMARY KEY (user_id, group_id)
);
```

Это классический пример ассоциативной таблицы, которая используется в тех случаях, когда база данных корректно нормализована. Однако такая таблица не является представлением какого-либо объекта, а по существу содержит составной первичный ключ. Скорее всего, в данной ситуации использовать класс `GenericObject` не стоит.

Как правило, если перед именем таблицы можно поставить артикль (такой как *the*, *a* или *an*) и не нарушить грамматики, то такую таблицу следует считать представлением объекта (или сущности). Если при этом используется числовой первичный ключ, то использование класса `GenericObject` для доступа к базе данных оказывается очень удобным.

Типичная реализация GenericObject

Рассмотрим пример таблицы user из предыдущего раздела, которую удобно использовать для хранения имен пользователей некоторого защищенного Web-приложения внутренней корпоративной сети.

Если не брать во внимание преимущества класса GenericObject, то соответствующий класс можно описать следующим образом.

```
class User {
    private $user_id;

    public function __construct($id) {
        $this->user_id = $id;
    }

    public function GetField($strFieldName) {
        // ...
    }

    public function SetField($strFieldName, $strValue) {
        // ...
    }

    public function Save() {
        // ...
    }

    public function Destroy() {
        // ...
    }

    //и т.д.
}
```

Вполне очевидно, что при такой реализации понадобится выполнить большое количество операций копирования/вставки. Однако это не единственный недостаток такого подхода. Базовая функциональность для каждого класса (например, метод AddToGroup() для класса User) также может ввести в заблуждение. В реализованном классе будет содержаться и много других методов, аналогичных приведенным выше. Конечно, они являются чрезвычайно важными элементами, однако такие функции представляют собой лишь готовые стандартные элементы для каждого класса. Поэтому с точки зрения разработчика они вряд ли покажутся очень интересными и заслуживающими внимания. По-настоящему важными являются функции, которые специально написаны для данного класса. Наличие стандартных, а также менее важных элементов способно привести лишь к усложнению разработки более критичных методов.

С использованием класса GenericObject тот же класс можно описать так.

```
class User extends GenericObject {
    public function __construct($id) {
        $this->initialize("user", $id);
    }
}
```

Код самого класса GenericObject более подробно будет рассмотрен ниже.

На практике после объявления нового класса User как расширения класса GenericObject можно сразу же приступить к его использованию. Однако следует не забыть еще

заполнить таблицу `user` некоторыми тестовыми данными. При работе с СУБД PostgreSQL прямо в консольном окне можно ввести следующие операторы SQL.

```
COPY "user" (id, username, first_name, last_name) FROM stdin;
1 ed           Эд          Леки-Томсон
2 steve        Стив        Новицкий
3 alec         Алек        Ков
4 jim          Джим        Eide-Goodman
5 john         Джон        До
6 jane         Джейн       До
\.
SELECT pg_catalog.setval('user_id_seq', 6, true);
```

Если в базу данных были введены некоторые тестовые данные, то с помощью новой версии класса `User`, расширяющего класс `GenericObject`, можно легко получить полезную информацию.

```
$objUser = new User(1);
$strUsername = $objUser->GetField("username");
print $strUsername;
```

При выполнении этого фрагмента в качестве результата будет получена строка `ed`, поскольку именно это значение содержится в столбце `username` строки, в которой в поле `id` содержится значение 1.

Если для создания объекта воспользоваться параметром 2, то будет получена строка `steve`, если 3 — строка `alec` и т.д.

С помощью класса `GenericObject` так же просто изменить и значения свойств.

```
$objUser = new User(1);
$strUsername = $objUser->GetField("username");
print $strUsername. <br />\n;
$objUser->SetField("username", "edward");
$strUsername = $objUser->GetField("username");
$objUser->Save();
$id = $objUser->GetID();
print $id;
```

В этом фрагменте имя пользователя было изменено с `ed` на `edward`. Вызов метода `Save()` приводит к генерации и выполнению операторов SQL, обеспечивающих внесение изменений в базу данных.

Создание новых пользователей тоже не должно вызывать никаких осложнений.

```
$objUser = new User();
$objUser->SetField("username", "clive");
$objUser->SetField("first_name", "Кли夫");
$objUser->SetField("last_name", "Гарднер");
$objUser->Save();
$id = $objUser->GetID();
print $id;
```

При сохранении данных в новой строке таблицы класс `GenericObject` приостанавливает работу, чтобы установить, какой идентификатор был назначен базой данных. Для этого определяется последнее использованное число, применяемое для идентификации строки таблицы, это значение увеличивается на 1 и размещается в поле новой строки. Для обеспечения эффективного доступа к новой строке класс `GenericObject` определяет текущее значение `id`, которое только что было присвоено. После этого все требуемые изменения объекта с новым идентификатором будут выполняться с использованием оператора `UPDATE` и нового `id`.

На этом можно завершить рассмотрение основных рекомендаций по применению класса GenericObject. Однако настоящему разработчику на языке PHP нужно знать, как же это все работает. Именно этим мы теперь и займемся.

Встреча с предком

Вот полный исходный код класса GenericObject, с которым стоит познакомиться. Затем можно более детально рассмотреть его методы и свойства. Сохраните этот класс как genericobject.phpm.

```
<?
class GenericObject {

    # Переменные-члены

    private $id;
    private $table_name;

    private $database_fields;
    private $loaded;
    private $modified_fields

    # Методы

    public function Reload() {
        $sql = new sql();
        $id = $this->id;
        $table_name = $this->table_name;
        $sql->query("SELECT * FROM \"{$table_name}\" WHERE id='{$id}'");
        $result_fields = $sql->get_row_hash();
        $this->database_fields = $result_fields;
        $this->loaded = 1;
        if (sizeof($this->modified_fields) > 0) {
            foreach ($this->modified_fields as $key => $value) {
                $this->modified_fields[$key] = false;
            };
        };
    };

    private function Load() {
        $this->Reload();
        $this->loaded = 1;
    }

    public function ForceLoaded() {
        $this->loaded = 1;
    }

    public function GetField($field) {
        if ($this->loaded == 0) {
            $this->Load();
        };
        return $this->database_fields[$field];
    }

    public function GetAllFields() {
        if ($this->loaded == 0) {
            $this->Load();
        };
        return ($this->database_fields);
    }
}
```

```

public function GetID() {
    return $this->id;
}

public function Initialize($table_name, $tuple_id = "") {
    $this->table_name = $table_name;
    $this->id = $tuple_id;
}

public function SetField($field, $value) {
    if ($this->loaded == 0) {
        if ($this->id) {
            $this->Load();
        };
    };
    $this->database_fields[$field] = $value;
    $this->modified = 1;
    $this->modified_fields[$field] = true;
}

public function Destroy() {
    $id = $this->id;
    $table_name = $this->table_name;
    if ($id) {
        $sql = new sql();
        $stmt = "DELETE FROM \"\" . $table_name . \"\" WHERE id='\" . $id . '\"';
        $sql->query($stmt);
    };
}

public function Save() {
    $id = $this->id;
    $table_name = $this->table_name;
    $sql = new sql();
    if (!$id) {
        $this->loaded = 0;
    };
    if ($this->loaded == 0) {
        # Новая сущность
        $stmt = "INSERT INTO \"\" . $table_name . \"\"(";
        foreach ($this->database_fields as $key => $value) {
            if (!is_numeric($key)) {
                $key = str_replace("'", "\'", $key);
                if ($value != "") {
                    $stmt .= "\"$key\",";
                };
            };
        };
        # Удаление последней запятой
        $stmt = substr($stmt, 0, strlen($stmt)-1);
        $stmt .= ") VALUES (";
        foreach ($this->database_fields as $key => $value) {
            if (!is_numeric($key)) {
                if ($value != "") {
                    $value = str_replace("'", "\'", $value);
                    $stmt .= "'$value',";
                };
            };
        };
        # Удаление последней запятой
        $stmt = substr($stmt, 0, strlen($stmt)-1);
        $stmt .= ")";
    } else {
        $stmt = "UPDATE \"\" . $table_name . \"\" SET ";
    }
}

```

```

foreach ($this->database_fields as $key => $value) {
    if (!is_numeric($key)) {
        if ($this->modified_fields[$key] == true) {
            $value = str_replace("'", "\'", $value);
            if ($value == "") {
                $stmt .= "\"$key\" = NULL, ";
            } else {
                $stmt .= "\"$key\" = '$value', ";
            }
        };
    };
};

# Удаление последней запятой и пробела
$stmt = substr($stmt, 0, strlen($stmt)-2);
$stmt .= " WHERE id='$id'";
};

$return_code = $sql->query($stmt, 1);
if ($this->loaded == 0) {
    # Попытка получения нового ID.
    $stmt = "SELECT MAX(id) AS id FROM \"{$table_name}\" WHERE ";
    foreach ($this->database_fields as $key => $value) {
        if (!is_numeric($key)) {
            if ($value) {
                if ($this->modified_fields[$key] == true) {
                    $value = str_replace("'", "\'", $value);
                    $stmt .= "\"$key\" = '$value' AND ";
                };
            };
        };
    };
    # Удаление последнего лишнего " AND "
    $stmt = substr($stmt, 0, strlen($stmt)-5);
    $sql->query($stmt);
    $result_rows = $sql->get_table_hash();
    $proposed_id = $result_rows[0]["id"];
    if ($proposed_id > 0) {
        $this->loaded = 1;
        $this->id = $proposed_id;
        return true;
    } else {
        return false;
    };
};
return($return_code);
}

```

Взаимодействие класса GenericObject с базой данных

В первую очередь следует обратить внимание на то, что класс GenericObject существенно затрудняет возможность использования класса sql, предоставляющего интерфейс к базе данных PostgreSQL.

На самом деле можно обеспечить взаимодействие этого класса практически с любой базой данных: Microsoft SQL Server, MySQL, Oracle и т.д. Для этого достаточно внести лишь небольшие изменения. Однако сейчас для работы с базой данных PostgreSQL воспользуемся следующим кодом. Сохраните этот класс в файле sql.phpm.

```

class sql {
    private $result_rows; # Хеширование результирующих строк
    private $query_handle; # db: дескриптор запроса

```

```

private $link_ident; # db: идентификатор связи

public function __construct() {
    $db_username = "gobjtest";
    $db_password = "";
    $db_host = "db";
    $db_name = "gobjtest";
    $this->link_ident = pg_Connect("user='$db_username' password='$db_password'
                                    dbname='$db_name' host='$db_host')");
}

public function query($sql, $code_return_mode = 0) {
    $q_handle = pg_exec($this->link_ident, $sql);

    for ($i=0; $i<=pg_numrows($q_handle)-1; $i++) {
        $result = pg_fetch_array($q_handle,$i);
        $return_array[$i] = $result;
    };
    if (!$q_handle) {
        error_log("ЗАПРОС ОБРАБОТАТЬ НЕ УДАЛОСЬ: $sql\n");
    };
    $this->result_rows = $return_array;
    if (!$q_handle) {
        return(1);
    } else {
        return(0); # 0 в случае неудачи
    };
}

public function get_result($row_num, $column_name) {
    return ($this->result_rows[$row_num][$column_name]);
}

public function get_row_hash($row_num) {
    return ($this->result_rows[$row_num]);
}

public function get_table_hash() {
    return $this->result_rows;
}

public function done($close_connection = 0) {
    if ($close_connection) {
        pg_Close($this->link_ident);
    };
}

```

Обратите внимание на то, что в приведенном фрагменте имя пользователя базы данных, пароль (в данном случае пустой) и имя базы данных жестко заданы. В реальных проектах такой подход оказывается абсолютно неприемлемым. Данные подобного типа гораздо лучше экспорттировать в специальные или конфигурационные текстовые файлы. Однако в этой главе для обеспечения простоты изложения материала был выбран наиболее простой вариант реализации.

В приведенном выше классе sql нет ничего сложного. Вот простой пример его использования.

```

$sql = new sql();
$sql->query("SELECT id FROM \"user\"");
$result_rows = $sql->get_table_hash();
for ($i=0; $i<=sizeof($result_rows)-1; $i++) {
    print ($result_rows[$i] . "\n");
}

```

```
};  
$sql->done(1);  
};
```

При вызове метода `done()` с ненулевым параметром связь с базой данных закрывается. Если это нежелательно, то вызывать метод `done()` не нужно.

Как можно было увидеть, в целом класс оказался не очень сложным. В следующей главе будет обсуждаться концепция уровней абстракции при работе с базой данных. Ее можно использовать для обобщения и обеспечения независимости программного кода от используемой базы данных.

Методы и свойства класса GenericObject

Снова вернемся к классу `GenericObject`. В этом разделе принципы его реализации рассматриваются более подробно.

Свойства

Если вы еще проанализируете код из предыдущего раздела, то увидите, что в классе `GenericObject` имеются следующие свойства.

Свойство	Описание
<code>id</code>	Идентификатор строки в запросе, например 6
<code>table_name</code>	Имя таблицы в запросе, например <code>user</code>
<code>database_fields</code>	Ассоциативный массив с ключами, которые являются именами столбцов таблицы, и их значениями для данной строки (например, <code>username => ed</code>)
<code>loaded</code>	Показывает, был ли данный объект заполнен информацией из базы данных. До проверки этого значения загружать данные нет необходимости. В этом поле содержится значение 0, если данные не загружены, и 1 в противном случае
<code>modified_fields</code>	Аналогичен переменной-члену <code>database_fields</code> , однако содержит значение <code>true</code> или <code>false</code> в зависимости от того, было ли изменено данное поле с момента его последней загрузки из базы данных (например, <code>username => true, first_name => false</code>)
<code>modified</code>	Содержит значение 1, если с момента последней загрузки из базы данных что-либо было изменено, и 0 — в противном случае

Методы

При внимательном изучении кода из предыдущего раздела можно увидеть, что в классе `GenericObject` реализованы следующие методы.

Метод	Параметры	Описание
<code>Initialize</code>	<code>table_name</code> — имя таблицы базы данных; <code>tuple_id</code> — идентификатор строки в запросе	Вызывается подклассом для задания имени таблицы и идентификатора строки в запросе
<code>Load</code>	Нет	Альтернативная точка входа в функцию <code>Reload</code>

Окончание таблицы

Метод	Параметры	Описание
Reload	Нет	Заполняет переменную-член database_fields текущими значениями базы данных
ForceLoaded	Нет	Обеспечивает выполнение условия успешной загрузки объекта подкласса, даже если на самом деле это не так. Этот метод полезно использовать, если значения были заданы вручную каким-либо другим способом, поскольку это позволит предотвратить любую автоматическую загрузку при вызове метода GetField
GetID	Нет	Предоставляет идентификатор текущей строки. Этот идентификатор определяется либо при инстанцировании объекта, либо при вызове метода Save() для сохранения нового объекта
GetField	field — имя поля, значение которого извлекается из базы данных	Предоставляет значение поля по запросу. Если оно еще не было загружено, сначала автоматически вызывается функция Reload()
GetAllFields	Нет	Аналогичен предыдущему методу, но возвращает все поля и их значения. При необходимости сначала также вызывается метод Reload()
SetField	field — имя поля, значение которого устанавливается; value — устанавливаемое значение	Обновляет внутреннюю хеш-таблицу объекта (database_fields), а затем для флага modified и соответствующего значения массива modified_fields устанавливает значение 1
Destroy	Нет	Перманентно удаляет сущность из базы данных. После вызова этого метода объект уже нельзя использовать!
Save	Нет	Сохраняет содержимое объекта в базе данных

Теперь нужно более подробно познакомиться с работой метода Save, поскольку он является одним из наиболее сложных методов класса GenericObject.

Метод Save

Этот метод сначала определяет, является данная сущность новой или она уже существует в базе данных. Для этого просто проверяется значение свойства id. Если оно имеет значение null, значит, сущность является новой, в противном случае — нет. Результаты этой проверки впоследствии используются для определения того, какой оператор SQL, UPDATE или INSERT, использовать для взаимодействия с базой данных. Оператор UPDATE можно использовать для обновления существующей записи, а INSERT — для вставки новой записи.

При использовании оператора INSERT SQL-запрос строится с использованием всех пар “имя поля–значение”, которые были определены до вызова метода. При этом объекту не известно, для каких из столбцов таблицы задан флаг NOT NULL и какие из них являются необязательными. Поэтому при разработке программного кода нужно самостоятельно позаботиться о том, чтобы значение было присвоено каждому

требуемому полю. В противном случае операция добавления записи в базу данных завершится неудачей.

Например, при задании для полей `first_name` и `last_name` объекта осмысленных значений (например, Джон и До, соответственно) будет сгенерирован следующий оператор SQL.

```
INSERT INTO "user" ("first_name", "last_name") VALUES ('Джон', 'До')
```

Обратите внимание, что в приведенном запросе не указаны имена пользователей (для поля `username`), поэтому они и не будут вставлены в базу данных, т.е. ее соответствующие поля останутся пустыми. Однако если для столбца установлен флаг NOT NULL, то будет получено сообщение об ошибке.

После успешной вставки новой записи в базу данных расширенный объект класса `GenericObject` качественно изменяется. Теперь работу можно продолжить не с новой сущностью, а с существующей. А существующей сущности, как и всем другим сущностям, должно соответствовать значение `id`. Как же его можно определить?

Как MySQL, так и PostgreSQL, позволяет определить последнее значение `id`, которое было автоматически сгенерировано при выполнении последней операции вставки. Такая возможность доступна при использовании не всех систем управления баз данных. Кроме того, ее реализации также сильно отличаются.

Для определения значения `id` проще всего воспользоваться оператором `SELECT`. Следующий оператор позволяет получить требуемое значение в 90% случаев.

```
SELECT MAX(id) FROM "user"
```

Однако в интервале времени между выполнением операторов `INSERT` и `SELECT` кто-то еще может выполнить операцию вставки, что может привести к искажению результата.

Более безопасный подход заключается в добавлении в запрос условия, с использованием которого и будет выполняться поиск требуемой записи. Для определения условия нужно воспользоваться максимально точными параметрами. Шансы того, что найдется еще одна запись, вставленная в тот же промежуток времени и удовлетворяющая заданным параметрам поиска, весьма незначительны.

```
SELECT MAX(id) FROM "user" WHERE "first_name"='Джон' AND "last_name"='До'
```

После выполнения всех описанных выше операций полученное значение `id` связывается с объектом, и он считается загруженным.

При обновлении существующей записи в определении значения `id` нет никакой необходимости, поскольку оно уже известно. Для этого используется простой оператор `UPDATE`, который используется точно так же, как показано выше для оператора `INSERT`. Однако следует заметить, что вместо обновления значений всех столбцов изменения вносятся лишь в те из них, значения которых были модифицированы.

Это является важным не только для экономии времени, но и для обеспечения целостности базы данных. Любые NULL-значения из базы данных преобразуются в пустые строки объекта `GenericObject`. Такой нюанс может и не иметь большого значения, однако если в процессе обновления все NULL-значения будут преобразованы в пустые строки, а в последующих запросах для фильтрации результатов используется условие `NOT NULL`, то можно получить совершенно непредвиденный результат.

Как можно было убедиться, метод `Save()` является достаточно интеллектуальным и представляет собой ядро класса `GenericObject`.

Преимущества использования класса GenericObject

К этому моменту вам стали уже известны основные принципы, заложенные в реализацию класса GenericObject, а также главные аспекты его поведения. Авторы надеются, что его исходный код был внимательно проанализирован, а соответствующее описание осмыслено. Ниже в этой главе будет рассмотрено небольшое демонстрационное приложение, а сейчас самое время подвести итог и сформулировать, какие же преимущества можно получить при использовании GenericObject.

- ❑ На написание операторов SQL нужно гораздо меньше времени. Свободное время можно посвятить написанию программного кода.
- ❑ Существенно упрощается перенос приложения с одной системы управления базой данных на другую. При этом операторы SQL можно размещать в одном файле, а не в сотнях.
- ❑ Программный код становится меньшим и гораздо более понятным, что значительно облегчает его поддержку.
- ❑ Зная, что операции манипулирования данными реализованы полностью и не содержат ошибок, можно без особых проблем сосредоточиться на разработке ядра приложения.

Конечно, работать с отдельными сущностями очень удобно, однако зачастую разработчики сталкиваются с тем, что гораздо лучше использовать их как часть целой коллекции объектов. Поэтому самое время познакомиться с классом GenericObjectCollection, который как раз это и позволяет делать.

Класс GenericObjectCollection

Существует две основных ситуации, когда нужно генерировать коллекцию объектов-сущностей, например пользователей, покупателей или любых других объектов, которые расширяют класс GenericObject.

К этим двум ситуациям относятся следующие.

- ❑ Необходимо создать группу объектов, которые соответствовали бы какому-нибудь другому объекту.
- ❑ Нужно сформировать группу объектов, соответствующих заданному критерию поиска.

В качестве хорошего примера каждой из ситуаций можно привести следующие.

- ❑ Создание коллекции объектов-пользователей, которые являются членами группы администраторов.
- ❑ Создание набора объектов-пользователей, имя которых начинается с символа А.

Следует отметить, что различия между этими двумя случаями весьма незначительны, но в то же время они являются важными.

В первом случае для решения поставленной задачи прекрасно подходит метод внешнего объекта. К классу Group можно добавить метод GetAllUsersWhoAreMembers, который каким-либо образом должен вернуть коллекцию пользователей, которые являются членами группы.

Во втором случае критерий определяется каким-либо простым параметром, который не представляется в виде другого класса. Конечно, можно реализовать класс `LetterOfTheAlphabet` с методом `GetAllUsersWhoseFirstNameBeginsWith`, однако с точки зрения ООП такая реализация будет некорректной.

Вместо описанных подходов многие разработчики предпочитают использовать так называемый класс `Home`. Основной задачей этого класса является создание других классов. Именно поэтому класс `Home` иногда называют классом `Factory` (классом-фабрикой).

В предыдущем примере можно было бы воспользоваться классом `UserHome` с методом `GetAllUsersWithFirstNameBeginningWith`, которому в качестве параметра передавался бы один символ. Он использовался бы следующим образом.

```
$arUserCollection = Array();
$arUserCollection =
    UserHome::GetAllUsersWithFirstNameBeginningWith('A');
```

Обратите внимание на инстанцирование экземпляра `UserHome` как статического класса. Именно поэтому между любым его экземпляром нет никакой разницы. Для получения более подробной информации по этому вопросу читайте главу 3 "Объектный подход в действии".

Теперь давайте разберемся, как же можно реализовать метод из предыдущего примера. Ниже вы узнаете, почему подобное решение нельзя считать наилучшим.

Традиционная реализация

По существу, следующий код выглядит корректно.

```
class UserHome {
    public function GetAllUsersWithFirstNameBeginningWith($strLetter) {
        $sql = new sql();
        $strLetter = strtolower($strLetter);
        $sql->query("SELECT id FROM \"user\" WHERE lower(first_name) LIKE
'$strLetter%'");
        $result_rows = $sql->get_table_hash();
        $arUserObjects = Array();
        for ($i=0; $i<sizeof($result_rows)-1; $i++) {
            $arUserObjects[] = new User($result_rows[$i]["id"]);
        };
        return($arUserObjects);
    }
};
```

Попробуйте выполнить этот фрагмент. Сохраните его в файле `userhome.phpm` и попробуйте запустить следующий код.

```
require ("genericobject.phpm");
require ("genericobjectcollection.phpm");
require ("user.phpm");
require ("userhome.phpm");
$arUsers = UserHome::GetAllUsersWithFirstNameBeginningWith('д');
print sizeof($arUsers);
```

Если использовались тестовые данные, приведенные в этой главе выше, в результате на экране появится цифра 2, т.е. будут найдены Джон До и Джейн До.

Аналогичный метод можно использовать для сбора любой коллекции объектов. Он может принадлежать либо классу `Home`, либо быть частью более сложного метода внешнего объекта.

Однако на самом деле все не так хорошо, как может показаться на первый взгляд. В следующем разделе вы узнаете, как можно модифицировать класс `GenericObjectCollection` и получить от его применения еще большую пользу.

Когда традиционная реализация оказывается неудачной

Подход, описанный в предыдущем разделе, является неприемлемым с точки зрения эффективности выполнения запросов SQL.

Вернемся к предыдущему примеру. Все, что было сделано, — это генерация запроса на получение коллекции объектов `user`. Даже если в базе данных содержалось двадцать, а не две записи, для решения поставленной задачи будет вполне достаточно одного запроса, как в приведенном выше методе.

Однако что изменится, если над полученной коллекцией из двадцати пользователей понадобится осуществить какие-либо определенные действия, например, вывести список их полных имен?

Взгляните на следующий фрагмент кода.

```
$arUsers =
    UserHome:: GetAllUsersWithFirstNameBeginningWith('д');
for ($i=0; $i<=sizeof($arUsers)-1; $i++) {
    print $arUsers[$i]->GetField("first_name") . " ";
    $arUsers[$i]->GetField("last_name") . "<br />\n";
}
```

Конечно, этот фрагмент кода является работоспособным, однако давайте проанализируем каждую итерацию цикла. Для заполнения полей каждого объекта вызывается метод `Load()` класса `GenericObject`.

Если имеется двадцать пользователей, удовлетворяющих запросу, то всего придется выполнить 21 запрос: 1 для поиска всех этих пользователей и 20 — для получения информации о каждом из них.

```
SELECT id FROM "user" WHERE first_name LIKE 'j%';
SELECT * FROM "user" WHERE id = 12091;
SELECT * FROM "user" WHERE id = 12092;
SELECT * FROM "user" WHERE id = 12093;
```

Такой подход является не очень эффективным. Как можно увидеть, только один запрос является действительно необходимым.

```
SELECT id FROM "user" WHERE first_name LIKE 'j%';
```

Существует ли более эффективный способ получения того же набора данных, который способен обеспечить большую эффективность и наиболее полное соответствие принципам объектно-ориентированного подхода?

Конечно, такой способ существует. И он называется `GenericObjectCollection`.

Принципы, положенные в основу класса `GenericObjectCollection`

Причина, по которой выше пришлось использовать двадцать один запрос, является достаточно простой. С помощью первого запроса был получен набор пустых и незаполненных объектов, в каждом из которых содержался только идентификатор `id` и ничего больше. После этого для взаимодействия с каждым объектом в отдельности на каждой

итерации цикла пришлось генерировать отдельные запросы. Однако нельзя ли обращаться ко всем этим объектам как к одному единому набору и заполнять их одновременно?

Например, при выполнении первого запроса стало известно о том, что десять строк удовлетворяют заданному критерию. Эти строки имеют следующие идентификаторы id.

```
12091, 12092, 12093, 12094, 12095, 12096, 12097, 12098, 12099, 12100
```

Тогда для заполнения всей коллекции можно выполнить один следующий запрос.

```
SELECT * FROM "user" WHERE id IN (12091, 12092, 12093, 12094, 12095, 12096, 12097, 12098, 12099, 12100)
```

Такой подход является менее эффективным по сравнению с применением единственного запроса, приведенного выше. Однако два запроса — это все же гораздо лучше, чем одиннадцать запросов при реализации традиционного подхода.

В классе GenericObjectCollection запрос на множественное заполнение используется очень эффективно. Сначала выполняется поиск идентификаторов id строк, которые удовлетворяют заданному критерию, однако затем эти значения размещаются в стеке. В самом начале необходимо задать требуемые имена класса и таблицы в базе данных, в которой содержатся строки с нужными значениями id. После этого класс GenericObjectCollection создает массив инстанцированных объектов и, что очень важно, заполняет их данными. При этом все соответствующие поля помечаются как загруженные (см. выше), так что при последующих обращениях повторное заполнение осуществлять не нужно.

В следующем разделе представлен исходный код класса GenericObjectCollection.

Исходный код класса GenericObjectCollection

Вот исходный код класса GenericObjectCollection из файла genericobjectcollection.php. А в следующем разделе приведена улучшенная версия класса UserHome, которую можно использовать на практике

```
<?
class GenericObjectCollection {

    # Переменные-члены
    var $table_name;
    var $class_name;

    var $items_per_page;
    var $item_count = 0;

    var $id_array;
    var $obj_array;

    function __construct($table_name, $class_name) {
        $this->table_name = $table_name;
        $this->class_name = $class_name;
    }

    function AddTuple($id) {
        if (!(!$this->id_array)) {
            $this->id_array = array();
        };
        array_push($this->id_array, $id);
        $this->item_count = sizeof($this->id_array);
    }
}
```

```

}

function SetPageSize($items_per_page) {
    $this->items_per_page = $items_per_page;
}

function GetItemCount() {
    return $this->item_count;
}

function GetNumPages() {
    return(ceil($this->item_count / $this->items_per_page));
}

function _GetCommaSeparatedIDList($start_lim = 0, $end_lim = -1) {
    $s = "";
    if ($end_lim == -1) {
        $end_lim = sizeof($this->id_array)-1;
    };
    for ($i=$start_lim; $i<=$end_lim; $i++) {
        if (is_numeric($this->id_array[$i])) {
            $s = $s . $this->id_array[$i] . ",";
        };
    };
    $s = substr($s, 0, strlen($s) - 1);
    return $s;
}

function _GetIndexFromTupleID($tuple_id) {
    $index = -1;
    for ($i=0; $i<=sizeof($this->id_array)-1; $i++) {
        if ($this->id_array[$i] == $tuple_id) {
            $index = $i;
        };
    };
    return $index;
}

function PopulateObjectArray($page_num = 0) {
    $items_per_page = $this->items_per_page;
    if ($this->item_count > 0) {
        if ($page_num > 0) {
            $start_lim = ($items_per_page * ($page_num - 1));
            $end_lim = ($start_lim + $items_per_page) - 1;
            if ($end_lim > ($this->item_count-1)) {
                $end_lim = $this->item_count - 1;
            };
            $stmt = "SELECT * FROM \" . $this->table_name . "\" WHERE id IN
                (" . $this->_GetCommaSeparatedIDList($start_lim, $end_lim). ")";
        } else {
            $stmt = "SELECT * FROM \" . $this->table_name . "\" WHERE id IN
                (" . $this->_GetCommaSeparatedIDList(). ")";
        };
        # Выполнение SQL-запроса
        $sql = new sql();
        $sql->query($stmt);
        $result_rows = $sql->get_table_hash();

        for ($i=0; $i<=sizeof($result_rows)-1; $i++) {
            $this_row = $result_rows[$i];
            $this_db_row_id = $this_row["id"];
            $this_index = $this->_GetIndexFromTupleID($this_db_row_id);
            if ($this_index >= 0) {
                $refObjArrayIndexObj = &$this->obj_array[$this_index];

```

```

    $s = "\$refObjArrayIndexObj = new " . $this->class_name .
        "(" . $this_db_row_id . ")";
    eval($s);
    $refObjArrayIndexObj->ForceLoaded();
    foreach ($this_row as $key => $value) {
        if (!is_numeric($key)) {
            $refObjArrayIndexObj->SetField($key, $value);
        };
    };
};

};

function RetrievePopulatedObjects($page_num = 0) {
    if ($page_num > 0) {
        $items_per_page = $this->items_per_page;
        # Вычисления начального и конечного ограничения.
        $start_lim = ($items_per_page * ($page_num - 1));
        $end_lim = ($start_lim + $items_per_page) - 1;
        if ($end_lim > ($this->item_count-1)) {
            $end_lim = $this->item_count - 1;
        };
    } else {
        $start_lim = 0;
        $end_lim = $this->item_count - 1;
    };
    $return_array = array();
    $counter = 0;
    for ($i=$start_lim; $i<=$end_lim; $i++) {
        $return_array[$counter] = $this->obj_array[$i];
        $counter++;
    };
    return($return_array);
}
}

?>

```

Типичное использование класса GenericObjectCollection

Ниже приведена новая версия класса UserHome (сохраните ее в файле user-home.phpm), в которой используется класс GenericObjectCollection. Возможно, вы обратили внимание на то, что в этом классе содержатся переменные-члены, которые не позволяют применять его в качестве статического класса. Его объекты должны создаваться точно так же, как и объекты любого другого класса. Чуть ниже вы узнаете, почему понадобились подобные изменения.

```

<?
class UserHome {

    var $items_per_page = 12;
    var $item_count;
    var $page_count;

    public function __construct() {
        return(true); # В классе Home конструктор отсутствует
    }

    public function SetItemsPerPage($items_per_page) {
        $this->items_per_page = $items_per_page;
    }
}

```

```

public function GetItemCount() {
    return ($this->item_count);
}

public function GetPageCount() {
    return ($this->page_count);
}

public function GetAllUsersWithFirstNameBeginningWith($strLetter, $page_num=1) {
    $dbc = new GenericObjectCollection("user", "User");
    $sql = new sql();
    $strLetter = strtolower($strLetter);
    $sql->query("SELECT id FROM \"user\" WHERE lower(first_name) LIKE '$strLetter%'");
    $result_rows = $sql->get_table_hash();
    for ($i=0; $i<=sizeof($result_rows)-1; $i++) {
        $dbc->AddTuple($result_rows[$i] ["id"]);
    };
    $dbc->SetPageSize($this->items_per_page);
    $dbc->PopulateObjectArray($page_num);
    $objArray = $dbc->RetrievePopulatedObjects($page_num);

    $this->item_count = $dbc->GetItemCount();
    $this->page_count = $dbc->GetNumPages();

    return($objArray);
}
?>

```

Обратите внимание на существенные различия между двумя приведенными в данной главе реализациями класса `UserHome`, даже несмотря на использование в них одного и того же базового метода `GetAllUsersWithFirstNameBeginningWith`. В обоих случаях этот метод возвращает массив объектов `User`.

Как вы скоро увидите, оба метода работают совсем по-разному.

Тестирование класса `UserHome`

Попробуйте на практике воспользоваться новым классом `UserHome`, используя следующий фрагмент кода.

```

$uH = new UserHome();
$arUsers = $uH->GetAllUsersWithFirstNameBeginningWith('д');
for ($i=0; $i<sizeof($arUsers)-1; $i++) {
    print $arUsers[$i]->GetField("first_name") . " " .
        $arUsers[$i]->GetField("last_name") . "<br />\n";

```

Если вы воспользовались тестовыми данными, приведенными в начале этой главы, то получите тот же результат, что и ранее.

Джон До
Джейн До

Как это работает

Новый класс `UserHome` не расширяет класс `GenericObjectCollection`. Он просто использует его в качестве вспомогательного класса.

На первый взгляд, метод, используемый для получения пользователей, чье имя начинается с заданного символа, достаточно похож на приведенный выше одно-

именный метод. Однако перед строкой формирования запроса можно увидеть важную новую строку.

```
$dbc = new GenericObjectCollection("user", "User");
```

В этой строке создается новый экземпляр класса `GenericObjectCollection`, который связан с таблицей `user` объекта `User`. Обратите внимание на то, что объект `User` должен быть предварительно объявлен. Имена таблицы и класса хранятся как скрытые (`private`) переменные-члены объекта `GenericObjectCollection` и доступны для дальнейшего использования.

Далее в обоих методах выполняются одни и те же действия, связанные с генерацией запроса на поиск пользователей, которые соответствуют заданному критерию, и связанных с ними значений идентификаторов `id`.

```
$sql = new sql();
$strLetter = strtolower($strLetter);
$sql->query("SELECT id FROM \"user\" WHERE lower(first_name) LIKE
'$strLetter%'");
$result_rows = $sql->get_table_hash();
```

После этого, как и раньше, в цикле осуществляется проход по данным, полученным в результате выполнения запроса. Однако сейчас вместо простого инстанцирования объекта `GenericObject` значение `id` добавляется в объект класса `GenericObjectCollection` следующим образом.

```
$dbc->AddTuple($result_rows[$i] ["id"]);
```

Объект класса `GenericObjectCollection` просто записывает эти данные в стек для дальнейшего использования.

Затем объекту `GenericObjectCollection` передается важная информация: размер страницы. В первом примере это было просто жестко задано в классе `UserHome`. В данном случае размер страницы используется из-за того, что вас вряд ли устроит получение сразу всей информации. Наверняка лучше получать ее по частям.

Может оказаться, что заданному критерию будут соответствовать тысячи объектов. Скорее всего, со всеми этими объектами вы не будете одновременно выполнять какие-либо действия. Поэтому, установив размер страницы (равный, например, 12 элементам, как в рассматриваемом примере), можно получить коллекцию объектов, размер которой соответствовал бы заданному значению. Это является не просто стремлением к аккуратности, а позволяет обеспечить требуемую производительность при работе с данными большого объема.

Описанная выше логика уровня класса достаточно точно отражает концепцию, которую в качестве основы можно использовать при реализации конкретного приложения.

```
$dbc->SetPageSize(this->items_per_page);
```

Все, что дальше нужно сделать, — это сообщить объекту `GenericObjectCollection` о необходимости предоставить в виде массива части найденных объектов. Используемый для этого метод в качестве параметра получает номер страницы, который по умолчанию равен 0.

```
$dbc->PopulateObjectArray($page_num);
```

В ответ объект `GenericObjectCollection` генерирует и выполняет запрос, как было показано выше.

```
SELECT * FROM "user" WHERE id IN (12091, 12092, 12093, 12094, 12095, 12096, 12097, 12098, 12099, 12100)
```

Этот простой список значений `id` создается с помощью внутреннего метода `_GetCommaSeparatedIDList()`. В качестве параметров этот метод получает начальное и конечное ограничения, которые вычисляются на основе номера текущей страницы. Заданным пределам и соответствует список возвращаемых значений `id`. Так, например, если в качестве параметров методу `_GetCommaSeparatedIDList()` были переданы значения 50 и 60, то в результирующий список будут включены значения `id` из этого диапазона.

Перед использованием результатов запроса сначала в цикле создается внутренний список объектов `user`. При этом используется функция `eval`, поскольку при начальном создании объекта класса `GenericObjectCollection` переданное имя класса было сохранено в виде строки. Метод `eval` позволяет сначала сконструировать, а затем выполнить PHP-код “на лету”. Это показано ниже.

```
$refObjArrayIndexObj = &$this->obj_array[$this_index];
$s = "\$refObjArrayIndexObj =
    new ".$this->class_name."(\".$this_db_row_id.\");";
eval($s);
```

Значение `$this_index` вычисляется с помощью внутреннего метода `_GetIndexFromTupleID()`, который находит соответствие между элементом полного массива подходящих объектов со значением `id` текущей проверяемой строки.

Возможно, вы заметили, что в предыдущем примере не используется класс коллекции из главы 5. Этот класс просто возвращает массив объектов. Это означает, что с его помощью нельзя воспользоваться итератором и выполнить цикл по всей коллекции. Для обеспечения простоты авторы все оставили без изменений. Однако при желании можно внести необходимые изменения.

Далее в цикле выполняется обращение к методу `SetField` объекта `GenericObject`, чтобы значения полей соответствовали данным, полученным из базы данных.

```
$refObjArrayIndexObj->ForceLoaded();
foreach ($this_row as $key =>$value) {
    if (!is_numeric($key)) {
        $refObjArrayIndexObj->SetField($key, $value);
    };
}
```

С помощью метода `ForceLoaded` можно “обмануть” объект `GenericObject` и “заставить” его думать о том, что он уже был загружен. Это идеальная ситуация, поскольку все поля этого класса были только что установлены с помощью объекта `GenericObjectCollection` и информации, извлеченной из базы данных.

Теперь можно вернуться к классу `UserHome` и получить массив инстанцированных объектов.

```
$objArray = $dbc->RetrievePopulatedObjects($page_num);
```

Столище раз подчеркнуть, что начальный и конечный пределы быстро вычисляются на основе номера страницы, а затем возвращается соответствующая часть объектов.

Теперь можно узнать общее число строк, соответствующих заданному критерию, и, как следствие, общее число полученных страниц.

```
$this->item_count = $dbc->GetItemCount();  
$this->page_count = $dbc->GetNumPages();
```

Эти значения могут быть получены с использованием простых методов класса `UserHome`.

Класс GenericObjectCollection: подведение итогов

На первый взгляд, класс `GenericObjectCollection` может показаться слишком сложным, поскольку для выполнения относительно простой задачи используется достаточно много исходного кода.

В самом деле, его исходный код противоречит простоте его первоначального назначения. Вначале массив объектов классов, расширяющих `GenericObject`, заполняется данными таким образом, что число SQL-запросов, необходимое для выполнения операций с полным набором, сводится к минимуму.

Такая иерархия имеет огромные преимущества в любых приложениях для администрирования, требующих отображения и редактирования табличной информации о сущностях системы.

Эксперимент — это ключ к пониманию. Почему бы не переписать некоторые из существующих приложений, чтобы они использовали классы `GenericObject` и `GenericObjectCollection`? Они будут активно использоваться далее в этой книге, так что лучше разобраться с ними сейчас.

Резюме

В этой главе речь шла о том, как классы `GenericObject` и `GenericObjectCollection` могут в корне изменить подход к разработке классов для представления сущностей из таблиц базы данных приложения.

Стоит сказать, что почти все подклассы `GenericObject` будут иметь и другие методы, которые не могут быть эффективно автоматизированы, так что необходимость в хорошем знании SQL все же остается. И все же, `GenericObject` предоставляет отличные средства для сокращения больших объемов довольно скучного кода с операторами `UPDATE`, `INSERT` и `DELETE`, которые в противном случае составят большую часть приложения.

Класс `GenericObjectCollection` демонстрирует также изъяны прямого объектного представления и позволяет понять, что полное соответствие модели реальному объекту может привести к снижению производительности приложения. Грамотно используя класс `GenericObjectCollection`, можно построить приложение, соответствующее принципам ООП и обладающее высокой производительностью при взаимодействии с базой данных.

В следующей главе будут рассмотрены базы данных и уровни их абстракции — важная технология, используемая для достижения максимальной независимости приложения от базы данных.

8

Уровни абстракции базы данных

Большинство корпоративных приложений, которые разрабатываются на языке PHP, каким-либо образом взаимодействует с реляционной базой данных, такой как MySQL, PostgreSQL, Oracle, Microsoft SQL Server. Некоторые компоненты из набора разработчика, рассматриваемые в этой книге, предоставляют наглядные способы интеграции с базой данных. К сожалению, стандартные функции взаимодействия с данными языка PHP сильно зависят от используемой платформы.

Например, для соединения с базой данных MySQL используется функция `mysql_connect()`, а для соединения с базой данных PostgreSQL – функция `pg_connect()`. Для работы с СУБД Oracle и интерфейсом ODBC аналогичные функции также имеют свои собственные имена. Для каждой платформы все эти функции являются достаточно схожими, однако совсем не идентичными. Как следствие, любые изменения системной конфигурации, на которой используется приложение, приведут к необходимости модификации вызовов всех функций обращения к базе данных. Этую проблему можно разрешить за счет применения других программных продуктов, как с открытым кодом, так и коммерческих, которые предоставляют унифицированный набор функций доступа к данным, способных работать с любой базой данных. Теоретически нужно иметь возможность простого перехода от СУБД MySQL к PostgreSQL или любой другой платформе.

На самом деле из-за различий между разными диалектами языка SQL, разными уровнями поддержки спецификации SQL и соответствующими ее расширениями, используемыми в большинстве систем управления базами данных, обеспечить реальную переносимость практически невозможно, а в больших корпоративных приложениях зачастую и не нужно. В то же время компоненты, обсуждаемые в данной книге, должны быть максимально унифицированными и повторно используемыми в самых различных приложениях. Так что одним из основных требований к этим компонентам является обеспечение переносимости между различными платформами баз данных.

Для того чтобы упростить возможность повторного использования набора компонентов в различных приложениях, функционирующих на разных платформах, в данной главе обсуждается уровень абстракции базы данных, а также рассматриваются некоторые из наиболее популярных пакетов, в которых подобные уровни абстракции реализованы.

Кроме того, вы узнаете также и о важном шаблоне проектирования Singleton. Этот шаблон позволяет создавать в точности один экземпляр определенного класса. Это оказывается полезным, если, например, установлено одно соединение с базой данных и нужно предотвратить создание приложением множества таких соединений.

Что такое уровень абстракции базы данных

Уровень абстракции базы данных — это фрагмент кода на языке PHP, позволяющий использовать одни и те же конструкции для взаимодействия с различными базами данных. Вместо того чтобы изучать достаточно широкий набор стандартных функций PHP для доступа к разным базам данных, можно воспользоваться одними и теми же способами вызова, одними и теми же входными параметрами, возвращаемыми значениями и сообщениями об ошибках.

При использовании уровня абстракции базы данных можно легко взять какой-либо компонент одного приложения, предназначенный, например, для работы с базой данных PostgreSQL, и повторно воспользоваться им в другом приложении для работы с базой данных Oracle. Хотя при этом и придется пересмотреть синтаксис операторов SQL, в сам код PHP никаких изменений вносить не потребуется.

Уровень абстракции существенно упрощает также реализацию поддержки регистрации событий и отладку взаимодействия с базой данных, поскольку в этом случае все функции используются централизовано. Вместо того чтобы применять в приложении функции pg_query() или mysql_query(), можно реализовать одну функцию, в которую будут передаваться все запросы. Тогда решение задачи перехвата, анализа и регистрация запросов в файле, а также отладка взаимодействия с базой данных существенно упростится.

Простая реализация

Самый простой способ реализации базового уровня абстракции заключается в создании класса-оболочки вокруг встроенных функций взаимодействия с определенной базой данных. Тогда для повторного использования этого компонента для работы с другой базой данных потребуется переписать код, в котором используются стандартные функции. Рассматриваемый ниже пример предназначен для работы с базой данных PostgreSQL. Ниже в этой главе будет рассмотрен и более сложный подход, однако этот простой пример поможет разобраться, как же функционирует уровень абстракции базы данных.

Поскольку в большинстве приложений не требуется модифицировать внутреннюю структуру базы данных, при реализации уровня абстракции мы ограничимся использованием операторов SELECT, INSERT, UPDATE и DELETE. Метод select() должен возвращать ассоциативный массив с полным набором записей. Функции insert(), update() и delete() должны возвращать количество записей, над которыми был выполнен соответствующий оператор SQL. Для простоты примем допущение, что последние три функции должны обеспечивать обработку любой требуемой строки.

Конфигурационный файл

Как правило, одновременно выполняется подключение только к одной базе данных. Поэтому можно создать конфигурационный файл, где будет храниться вся информация, которая требуется для установки соединения. В данном случае для создания такого файла был использован язык PHP, однако при желании можно воспользоваться и более удобным форматом. Этот вопрос более подробно будет рассматриваться в главе 17. Сохраните приведенный фрагмент в файле config.php.

```
<?php
$cfg['db']['host'] = 'localhost';
$cfg['db']['port'] = 5432; // Порт, используемый по умолчанию СУБД
PostgreSQL
$cfg['db']['user'] = 'postgresql';
$cfg['db']['password'] = 'mypass';
$cfg['db']['name'] = 'mydatabase';
?>
```

Установка соединения

Класс должен открывать соединение с базой данных в своем конструкторе, а закрывать его — в деструкторе. Стока, используемая для установки соединения, формируется на основе конфигурационного файла config.php. Создайте файл class.Database.php и введите в него следующий код.

```
<?php
require_once('config.php');

class Database {
    private $hConn;

    public function __construct() {
        global $cfg; // обеспечивает доступ этого метода
                     // к ассоциативному массиву $cfg,
                     // делая его глобальным

        $connString = ' host=' . $cfg['db']['host'];
        $connString .= ' user=' . $cfg['db']['user'];
        $connString .= ' password=' . $cfg['db']['password'];
        $connString .= ' port=' . $cfg['db']['port'];
        $connString .= ' dbname=' . $cfg['db']['name'];

        $this->hConn = @pg_connect($connString);

        if(! is_resource($this->hConn)) {
            throw new Exception("Нельзя соединиться с базой данных ".
                "с использованием параметров \"$connString\"", E_USER_ERROR);
        }
    }

    public function __destruct() {
        if(is_resource($this->hConn)) {
            @pg_close($this->hConn);
        }
    }
}
```

В конструкторе открывается соединение с сервером базы данных, а дескриптор этого соединения сохраняется в закрытой переменной \$hConn. Если с помощью заданных параметров соединение установить не удалось, генерируется исключение. Всегда следите за тем, чтобы все объекты класса Database создавались в блоке try...catch.

Для расширения функциональности класса Database нужно создать методы, которые позволяли бы использовать операторы SELECT, INSERT, UPDATE и DELETE. Эти методы рассматриваются в следующих разделах.

Выборка данных

Извлечь информацию из базы данных гораздо проще, чем из ассоциативного массива, в котором ключи соответствуют именам полей требуемой таблицы. Ниже приведен исходный код метода select() для базы данных PostgreSQL.

```
public function select($sql) {
    $hRes = @pg_query($this->hConn, $sql);
    if(! is_resource($hRes)) {
        $err = pg_last_error($this->hConn);
        throw new Exception($err);
    }

    $arReturn = array();
    while( ($row = pg_fetch_assoc($hRes)) ) {
        $arReturn[] = $row;
    }

    return $arReturn;
}
```

Для каждой строки набора данных, который был получен в результате обработки запроса \$sql, строится двумерный массив. Его первое измерение является числовым индексом, который определяет номер строки (начиная с нуля). Второе измерение определяет имя поля.

Такой подход оказывается не очень удачным при работе с большими наборами данных, поскольку каждая строка хранится в оперативной памяти. Ниже в этой главе будут рассмотрены и другие, более эффективные, способы реализации метода select().

Изменение информации

После того как появилась возможность извлечения информации из базы данных, нужно обеспечить механизм добавления, модификации и удаления записей. В следующем фрагменте кода показана одна из возможных реализаций методов insert(), update() и delete(). Первый метод в качестве параметров получает имя таблицы и ассоциативный массив с именами полей и их значениями. При этом оператор SQL конструируется внутри функции, обеспечивая корректное использование строковых значений.

```
public function insert($table, $arFieldValues) {
    $fields = array_keys($arFieldValues);
    $values = array_values($arFieldValues);

    // Создается массив значений, который используется
    // для формирования параметра VALUES оператора
    // SELECT.
    // Функция pg_escape_string используется для
```

```

// нечисловых значений.
$escVals = array();
foreach($values as $val) {
    if(! is_numeric($val)) {
        //обеспечение корректности строковых значений
        $val = '"' . pg_escape_string($val) . '"';
    }
    $escVals[] = $val;
}

//конструирование оператора SQL
$sql = " INSERT INTO $table (" .
$sql .= join(', ', $fields);
$sql .= ') VALUES(';
$sql .= join(', ', $escVals);
$sql .= ')';

$hRes = pg_query($sql);
if(! is_resource($hRes)) {
    $err = pg_last_error($this->hConn) . "\n" . $sql;
    throw new Exception($err);
}

return pg_affected_rows($hRes);
}

```

Метод `update()` в качестве параметров получает имя таблицы и ассоциативный массив с именами полей и их значениями. При этом ключи определяют поля, которые были модифицированы, а значения — новые значения этих полей. Массив `$arConditions` имеет аналогичную структуру и определяет поля и значения, которые будут использоваться в параметре WHERE.

```

public function update($table, $arFieldValues, $arConditions) {

    // создание массива для конструирования параметра SET
    $arUpdates = array();
    foreach($arFieldValues as $field => $val) {
        if(! is_numeric($val)) {
            //обеспечение корректности строковых значений
            $val = '"' . pg_escape_string($val) . '"';
        }
        $arUpdates[] = "$field = $val";
    }

    // создание массива для конструирования параметра WHERE
    $arWhere = array();
    foreach($arConditions as $field => $val) {
        if(! is_numeric($val)) {
            //обеспечение корректности строковых значений
            $val = '"' . pg_escape_string($val) . '"';
        }
        $arWhere[] = "$field = $val";
    }

    $sql = "UPDATE $table SET ";
    $sql .= join(', ', $arUpdates);
    $sql .= ' WHERE ' . join(' AND ', $arWhere);

    $hRes = pg_query($sql);
    if(! is_resource($hRes)) {
        $err = pg_last_error($this->hConn) . NL . $sql;
    }
}

```

```

        throw new Exception($err);
    }

    return pg_affected_rows($hRes);
}

```

Метод `delete()` в качестве параметров получает имя таблицы, из которой нужно удалить строки, и ассоциативный массив пар “имя поля–значение”, предназначенный для формирования параметра WHERE оператора DELETE.

```

function delete($table, $arConditions) {

    //создание массива для конструирования параметра WHERE
    $arWhere = array();
    foreach($arConditions as $field => $val) {
        if(! is_numeric($val)) {
            //обеспечение корректности строковых значений
            $val = '"' . pg_escape_string($val) . '"';
        }

        $arWhere[] = "$field = $val";
    }

    $sql = "DELETE FROM $table WHERE " . join(' AND ', $arWhere);

    $hRes = pg_query($sql);
    if(! is_resource($hRes)) {
        $err = pg_last_error($this->hConn) . NL . $sql;
        throw new Exception($err);
    }

    return pg_affected_rows($hRes);
}

```

В каждой из приведенных трех функций оператор SQL конструируется путем объединения элементов массивов `$arFieldValues` и `$arConditions`. Кроме того, в них обеспечивается корректное представление строковых значений, содержащихся в этих массивах, в частности символов “апостроф”.

После выполнения сгенерированных SQL-запросов каждая функция возвращает количество строк, над которыми запрос выполнялся. Обратите внимание, что значение 0 (которое интерпретируется как “ложь”) необязательно свидетельствует о неудачном завершении операции, а просто о том, что ничего не было модифицировано. В частности, для операторов UPDATE и DELETE именно это значение может оказаться правильным, в зависимости от входных параметров функции и содержимого таблицы. Оператор INSERT обычно применяется к одной записи. Однако если в таблице реализованы специализированные триггеры, то этот оператор может быть применен и к 0 записей, а также к 1 и более.

Использование класса Database

Класс `Database` использовать в приложениях гораздо проще и удобнее, чем применять функции `pg_[x]` напрямую. Создайте простую таблицу PostgreSQL с помощью следующего оператора SQL.

```

CREATE TABLE "mytable" (
    "id" SERIAL PRIMARY KEY NOT NULL,
    "myval" varchar(255)
);

```

После этого создайте файл DB_test.php со следующим кодом.

```
<?php
require_once('class.Database.php');

try {
    $objDB = new Database();
} catch (Exception $e) {
    echo $e->getMessage();
    exit(1);
}

try {
    $table = "mytable";
    $objDB->insert($table, array('myval' => 'foo') );
    $objDB->insert($table, array('myval' => 'bar') );
    $objDB->insert($table, array('myval' => 'blah') );
    $objDB->insert($table, array('myval' => 'mu') );
    $objDB->update($table, array('myval' => 'baz'), array('myval' => 'blah'));
    $objDB->delete($table, array('myval' => 'mu'));
    $data = $objDB->select("SELECT * FROM mytable");
    var_dump($data);
} catch (Exception $e) {
    echo "Обработка запроса завершилась неудачно" . NL;
    echo $e->getMessage();
}
?>
```

В приведенном примере для создания четырех строк в таблице mytable, в которых для поля myval заданы значения foo, bar, blah и mu соответственно, использовались четыре оператора INSERT. Далее с использованием оператора UPDATE в строках, в которых в поле myval содержалось значение blah, выполнена замена этого значения на baz (в данном случае это одна строка). И наконец, с помощью оператора DELETE была удалена строка со значением mu в поле myval.

Функция var_dump используется для отображения текущего содержимого таблицы и генерирует следующее.

```
array(3) {
    [0]=>
    array(2) {
        ["id"]=>
        string(2) "1"
        ["myval"]=>
        string(3) "foo"
    }
    [1]=>
    array(2) {
        ["id"]=>
        string(2) "2"
        ["myval"]=>
        string(3) "bar"
    }
    [2]=>
    array(2) {
        ["id"]=>
        string(2) "3"
        ["myval"]=>
        string(3) "baz"
    }
}
```

Следует заметить, что последующие запуски тестового сценария приведут к дальнейшей вставке строк в таблицу базы данных, поскольку в этом сценарии нет кода, обеспечивающего удаление всего содержимого таблицы. Поэтому при щелчке на кнопке Refresh (Обновить) в окне браузера можно получить гораздо более длинный результат.

Как можно увидеть из приведенного листинга, оператор SELECT возвращает весь набор записей в виде двумерного массива. Первое измерение обеспечивает числовое индексирование, при котором каждой строке соответствует свой индекс. Второе измерение представляет собой ассоциативный массив, в котором имя поля используется в качестве ключа.

Если код из приведенного примера потребуется использовать для взаимодействия с базой данных MySQL, а не PostgreSQL, то все функции pg_[x] придется заменить на их эквиваленты mysql_[x]. Эти изменения могут потребовать модификации и порядка следования входных параметров этих функций, а также способа их использования внутри класса. Однако внешний программный интерфейс при этом останется без изменений. Кроме того, при необходимости также придется немного модифицировать и сами операторы SQL. Вместе с тем код PHP останется без изменений.

Однако следует заметить, что рассмотренный выше пример класса Database для большинства приложений оказывается слишком простым. С его помощью нельзя эффективно обрабатывать большие наборы данных. Этот класс не позволяет использовать транзакции и поддерживает лишь самые простые механизмы обработки ошибок. Поэтому в оставшейся части этой главы будет рассмотрен набор более robustных функций уровня абстракции.

Уровень абстракции PEAR DB

Для того чтобы оценить всю мощь уровня абстракции базы данных, стоит познакомиться с классом DB библиотеки PEAR, которая входит в комплект поставки PHP. Этот уровень абстракции позволяет одновременно поддерживать соединения со многими типами баз данных, а также в полной мере использовать все преимущества интерфейса ODBC.

В отличие от встроенных функций PHP для работы с базами данных класс DB для всех соединений позволяет использовать один и тот же синтаксис. Для установки соединения с использованием класса DB используется следующий синтаксис.

```
[тип базы данных]://[пользователь]:[пароль]@[имя узла]/[имя базы данных]
```

Эта строка определяет *имя источника данных* (data source name – DSN). С помощью унифицированного синтаксиса, который используется для определения имени DSN, можно подключиться к любой базе данных, поддерживаемой классом DB.

```
// подключение к серверу MySQL с помощью учетной записи
// root и пароля passw0rd
// соединение с базой данных 'dbfoo'
$dsn = 'mysql://root:passw0rd@db01/dbfoo';
```

```
// подключение к серверу PostgreSQL на локальном компьютере
// с помощью учетной записи postgres без пароля
// соединение с базой данных 'mydb'
$dsn = 'pgsql://postgres@localhost/mydb';
```

Ниже приведен перечень баз данных и строк, определяющих их тип, которые поддерживаются в настоящее время.

Тип базы данных, поддерживаемой классом DB	База данных
ibase	InterBase
ifx	Informix
mssql	Microsoft SQL Server
mysql	MySQL (для версии 4.0 и ниже)
mysqli	MySQL (для версии 4.1 и выше)
oci8	Oracle 7/8/9
odbc	ODBC
pgsql	PostgreSQL
sqlite	SQLite
sybase	Sybase

Следует заметить, что поддержка требуемой базы данных должна быть обеспечена либо при компиляции модуля PHP, либо путем использования соответствующего динамического модуля. Как и в предыдущем примере, класс DB является оболочкой вокруг встроенных функций PHP. Поэтому если окажется, что требуемые функции оказались недоступными, то вы не сможете подключиться к базе данных соответствующего типа.

Подключение к базе данных с помощью класса DB

Для того чтобы подключиться к базе данных с помощью класса DB, нужно подсоединить файл с этим классом и задать корректное имя DSN. В следующем примере (`DB_connect.php`) устанавливается соединение с базой данных PostgreSQL, а затем данные извлекаются из таблицы и выводятся на экран с помощью функции `var_dump()`.

```
<?php
require_once('DB.php');
$dsn = 'pgsql://postgres@localhost/mydb';
$conn = DB::connect($dsn);

if(DB::isError($conn)) {
    //здесь можно добавить собственный код
    print("Нельзя подключиться к базе данных с использованием DSN $dsn");
    die($conn->getMessage());
}

//получение результата в виде ассоциативного массива
$conn->setFetchMode(DB_FETCHMODE_ASSOC);

$sql = "SELECT * FROM mytable";
$data = &$conn->getAll($sql); //Получение всех строк.
                                //Такой подход нужно использовать
                                //только для небольших наборов записей
```

```
// Всегда удостоверьтесь в отсутствии ошибок
if (DB::isError($data)) {
    print("Ошибка при выполнении запроса $sql");
    die ($data->getMessage());
}

var_dump($data);

$conn->disconnect(); //закрытие соединения
?>
```

Обратите внимание на то, что все функции класса DB могут вернуть объект класса DB_Error. Поэтому проверяйте все возвращаемые значения с помощью метода DB::isError(). При разработке собственного приложения вам может понадобиться выполнить более сложные действия, а не просто вызвать функцию die(). Кроме того, не забывайте о том, что на экран лучше никогда не выводить такую конфиденциальную информацию, как имя DSN или запрос SQL. Гораздо лучше сохранять подобную информацию в файле журнала. Вопрос разработки надежного механизма ведения журналов более подробно будет рассматриваться в главе 11.

Следует также заметить, что все методы класса DB являются статическими. По существу, класс DB является оболочкой для классов более низкого уровня, которые и обеспечивают поддержку всей необходимой функциональности. В классе DB содержится несколько полезных служебных методов, таких как connect(), disconnect(), isError() и т.д. Ни один из этих методов не зависит от состояния какого-либо объекта. Поэтому эти методы объявлены как статические.

Извлечение информации

Модуль DB предоставляет несколько различных способов извлечения информации из базы данных. Решение об использовании каждого из них нужно принимать в каждом конкретном случае. Еще раз вернемся к файлу DB_connect.php. Метод DB::connect() возвращает объект подкласса DB_common. Имя подкласса определяется на основе первой части имени DSN. В рассмотренном примере возвращается объект DB_pgsql. Класс DB_common предоставляет метод getAll(), который возвращает все записи, удовлетворяющие заданному запросу, в форме, которая была определена с помощью метода setFetchMode(). В примере использовался флаг DB_FETCHMODE_ASSOC, который задан по умолчанию. Это означает, что результатирующие записи по умолчанию возвращаются в виде ассоциативного массива, в качестве ключей которого используются имена полей базы данных.

К простым методам, которыми можно воспользоваться, относится метод getOne(). Этот метод возвращает значение первого поля из первой строки набора записей. Метод getOne() полезно использовать при извлечении из базы данных одного значения, такого как числовой идентификатор. Метод getRow() возвращает первую строку. Если необходимо получить массив из значений одного столбца, можно воспользоваться методом getCol(). В следующем фрагменте в качестве примера используется та же самая таблица mytable из предыдущего раздела. Вот как все вышеупомянутые методы можно использовать на практике.

```
$sql = "SELECT id, myval FROM mytable";

$arData = $conn->getAll($sql);
/*
```

```

* $arData = array (
*     0 => array('id' => 1, 'myval' => 'foo'),
*     1 => array('id' => 2, 'myval' => 'bar'),
*     2 => array('id' => 3, 'myval' => 'baz'),
*
*)
*/
$id = $conn->getOne($sql);
/*
* $id = 1;
*/
$arData = $conn->getRow($sql);
/*
* $arData = array('id' => 1, 'myval' => 'foo');
*/
$arData = $conn->getCol($sql);
/*
* $arData = array(1, 2, 3);
*/

```

Эффективная обработка данных

Все методы `get*`(), по существу, являются оболочками вокруг методов `fetch*`(). В большинстве случаев вполне достаточно воспользоваться методом `fetchRow()`. В отличие от функции `getAll()`, которая считывает в оперативную память весь набор записей, метод `fetchRow()` за один раз извлекает одну строку. Такой подход позволяет итеративно проходить по большому набору записей без использования большого объема памяти.

Методы `fetch*`() требуют, чтобы в качестве параметра им передавался объект `DB_result`. Это достигается за счет применения метода `query()`. В следующем фрагменте иллюстрируется, как все перечисленные методы можно использовать.

```

//получение результата в виде ассоциативного массива
$conn->setFetchMode(DB_FETCHMODE_ASSOC);

$sql = "SELECT * FROM mytable";
$result = & $conn->query($sql);

//проверка того, что запрос был обработан без ошибок
if (DB::isError($resulr)) {
    print("Ошибка при обработке запроса $sql");
    die($result->getMessage());
}

//вызов метода fetchRow() для получения одной строки
// за один раз
while($row = $result->fetchRow()) {
    print $row['myval'] . "<br>\n";
}

```

Как можно увидеть, в приведенном примере использовался синтаксис, аналогичный использованию стандартной функции `*_fetch_row()`. Каждый вызов метода `fetchRow()` приводит к увеличению внутреннего счетчика, который определяет, какая строка должна быть извлечена при следующем обращении к базе данных. При достижении конца набора записей этот метод возвращает значение `null`.

Другие полезные функции

Модуль DB предоставляет и другие полезные функции. Ниже кратко рассматриваются некоторые из них.

Функция query()

Метод `query()` позволяет выполнять операторы `SELECT`, а также операторы `UPDATE` и `DELETE`.

```
$sql = "INSERT INTO mytable (myvalue) VALUES('Foobar')"
$result = $conn->query($sql);
if(DB::isError($result)) {
    print("Нельзя выполнить оператор INSERT");
    die($result->getMessage());
}

$sql = "UPDATE mytable SET myvalue = 'foobar' WHERE id = 4";
$result = $conn->query($sql);
if(DB::isError($result)) {
    print("Нельзя выполнить оператор UPDATE");
    die($result->getMessage());
}
```

Не забывайте о том, что возвращаемые значения всех методов нужно проверять с помощью метода `DB::isError()`.

Функция nextID()

При использовании оператора `INSERT` иногда трудно найти способ извлечения данных, не зависящий от платформы. Многие системы управления реляционными базами данных поддерживают концепцию последовательности. Эта концепция позволяет использовать простой постоянный счетчик, в качестве начального значения которого выбирается некоторое предопределенное число (обычно 1). Путем увеличения значения счетчика из базы данных можно последовательно извлекать новые значения. Вышеупомянутая концепция используется в PostgreSQL, Oracle и других реляционных СУБД, в которых отсутствует встроенный механизм поддержки автоинкрементных полей. В СУБД MySQL поддержка последовательностей вообще отсутствует, поскольку в ее встроена поддержка автоинкрементных полей. Однако для обеспечения переносимости модуль DB эмулирует эту возможность путем использования таблицы с одним полем `AUTOINCREMENT`, которое называется `id`.

Позднее вы узнаете, насколько это полезно. Поскольку в данной книге в качестве примера используется система управления базами данных PostgreSQL, то таблицу можно создать с помощью следующих операторов SQL.

```
CREATE TABLE info (
    id SERIAL PRIMARY KEY NOT NULL,
    data varchar(255)
);
```

На самом деле будет создана следующая таблица.

```
CREATE SEQUENCE info_id_seq;
CREATE TABLE info (
    id integer NOT NULL PRIMARY KEY DEFAULT nextval('info_id_seq'),
    data varchar(255)
);
```

Отсюда видно, что путем создания объекта `info_id_seq` система управления базой данных PostgreSQL эмулирует реализацию автоинкрементного первичного ключа. Такое соглашение об именовании означает, что объект `info_id_seq` предназначен для управления столбцом `id` таблицы `info`. При этом для увеличения значения последовательности на единицу и получения нового значения используется функция `nextval()`. Конструкция `nextval('info_id_seq')` используется для получения значения по умолчанию для поля `id`. Тогда можно применять следующие операторы.

```
INSERT INTO info(data) VALUES('bla bla bla');
```

Однако получить реальное значение `id` для новой созданной записи нельзя, поскольку может существовать несколько записей со значением bla bla bla в поле `data`. Для извлечения информации о новой созданной строке таблицы в различных базах данных применяются разные подходы. Так, в Oracle используется псевдоним `ROWID`, в PostgreSQL – `OID`. Для того чтобы обойти все эти различия, лучше всего воспользоваться методом `nextID()` модуля DB. Этот метод позволяет получить новое значение заданной последовательности и неявно использовать его в операторах `INSERT`.

```
$newID = $conn->nextID('info_id', true);

if(DB::isError($newID)) {
    print("Ошибка получения нового ID!");
    die($newID->getMessage());
}

$sql = "INSERT INTO info(id, data) VALUES($newID, 'bla bla bla')";
$result = $conn->query($sql);

if(DB::isError($result)) {
    print("Обработка запроса $sql завершилась неудачно");
    die($result->getMessage());
} else {
    print("Создана запись с ID $newID");
}
```

Метод `nextID()` возвращает значение следующего элемента последовательности, имя которого формируется на основе значения его первого параметра и строки `_seq`. Второй параметр метода `nextID()` определяет, что при необходимости эта последовательность будет создана. Такая возможность может оказаться весьма полезной. Однако лучше всего оставить решение этой задачи системе управления базой данных. Тогда все операции по созданию корректной таблицы останутся в поле вашего зрения.

Функция `quoteSmart()`

В начале этой главы для обработки строковых значений в методах `insert()`, `update()` и `delete()` использовалась функция `pg_escape_string()`. Эта функция автоматически обрабатывает символы ' строковых значений, содержащихся в запросе. Модуль DB предоставляет простой метод `quoteSmart()`, который можно применять при работе с любой системой управления базой данных. В качестве параметра этот метод получает строковое значение, а на выходе возвращает строку, которая является корректной для базы данных, с которой устанавливается соединение.

```
function searchByName($name) {
    global $conn; //соединение уже должно быть установлено

    $sql = "SELECT * FROM users WHERE name = " . $conn->quoteSmart($name);
    $result = $conn->query($sql);
```

```
//для проверки отсутствия ошибок воспользуйтесь методом DB::isError
return $result;
}
```

Для краткости код проверки отсутствия ошибок в данном фрагменте не приведен. При использовании СУБД PostgreSQL для значения "O'Malley" параметра \$name метод \$conn->quoteSmart (\$name) вернет значение "'O\'Malley'". (Обратите внимание на одинарные кавычки вокруг имени и на символ апострофа.) Способ обработки этого символа может оказаться различным в зависимости от используемой базы данных.

Получение дополнительной информации

Выше были рассмотрены лишь несколько методов классов, определенных в модуле DB. Полную документацию с примерами можно найти по адресу <http://pear.php.net/manual/en/package.database.php>.

Завершенный уровень абстракции базы данных

В модуле PEAR DB содержится множество служебных функций, с помощью которых можно построить гибкий и легко настраиваемый уровень абстракции. Затем этот уровень можно применять совместно с любой системой управления базами данных. Рассматриваемый ниже класс представляет собой завершенное проектное решение, которое можно многократно использовать во всех разрабатываемых системах.

Создайте конфигурационный файл config.php со следующим именем DSN.

```
$cfg['db']['dsn'] = 'pgsql://postgres:password@localhost/mydb';
```

Создайте файл class.Database.php с классом Database. При этом в конструкторе нужно установить соединение с базой данных и при возникновении проблем сгенерировать сообщение об ошибке.

```
<?php

require_once('config.php');
require_once('DB.php');

class Database {
    private $conn;

    function __construct($dsn = null) {
        global $conn;

        //Если входной параметр отсутствует, используется значение из $cfg
        if($dsn == null) {
            $dsn = $cfg['db']['dsn'];
        }

        //Установка соединения с использованием переменной $dsn
        $this->conn = DB::connect($dsn);

        if(DB::isError($this->conn)) {
            //Соединение не установлено. Генерация исключения
            throw new Exception($this->conn->getMessage(), $this->conn->getCode());
        }
    }
}
```

```

    }

    //Данные всегда извлекаются в виде ассоциативного массива
    $this->conn->setFetchMode(DB_FETCHMODE_ASSOC);
}

function __destruct() {
    $this->conn->disconnect();
}
?>

```

Как правило, параметры источника данных можно извлекать из конфигурационного файла. В этом случае передавать параметр в конструктор не нужно. Однако иногда соединение необходимо установить с альтернативным источником данных. Именно для этого и предназначен параметр `$dsn`. Поскольку извлечение информации из базы данных в виде ассоциативного массива не является чрезмерно ресурсоемкой операцией, в конструктор была добавлена строка выбора формата значений, возвращаемых всеми функциями `get *()` и `fetch *()`. В приведенном фрагменте все возвращаемые значения этих функций являются ассоциативными массивами. Следует также заметить, что работать с ассоциативными массивами гораздо проще, чем со значениями других типов.

Обратите внимание, что при возникновении проблем в конструкторе генерируется исключение. Об этом не нужно забывать и при реализации всех остальных методов класса `Database`. В качестве параметров в конструктор класса `Exception` передается строковое сообщение и числовой код из объекта `DB_object`, возвращаемого методом `DB::connect()`.

Для того чтобы обеспечить закрытие соединения, нужно реализовать деструктор. Это является очень важным, поскольку без корректного закрытия соединения может оказаться, что каждый последующий запрос PHP-сценария будет передаваться другому резервному сокету сервера базы данных. В результате его ресурсы могут оказаться исчерпанными.

Теперь к классу `Database` можно добавить несколько методов, предназначенных для извлечения требуемой информации из базы данных, а именно, методы `select()`, `getAll()`, `getOne()` и `getCol()`. Метод `select()` должен возвращать объект `DB_result`, поскольку в результирующих наборах данных может содержаться очень много записей. В других методах просто вызываются методы класса `DB_common` с аналогичным именем.

```

//возвращает объект DB_result
function select($sql) {
    $result = $this->conn->query($sql);

    if(DB::isError($result)) {
        throw new Exception($result->getMessage(), $result->getCode());
    }

    return $result;
}

//возвращает двумерный ассоциативный массив
function getAll($sql) {
    $result = $this->conn->getAll($sql);

    if(DB::isError($result)) {

```

```

        throw new Exception($result->getMessage(), $result->getCode());
    }

    return $result;
}

//возвращает единственное скалярное значение
//из первого столбца первой записи
function getOne($sql) {
    $result = $this->conn->getOne($sql);

    if(DB::::isError($result)) {
        throw new Exception($result->getMessage(), $result->getCode());
    }

    return $result;
}

//возвращает нумерованный одномерный массив значений
//из первого столбца
function getCol($sql) {
    $result = $this->conn->getCol($sql);

    if(DB::::isError($result)) {
        throw new Exception($result->getMessage(), $result->getCode());
    }

    return $result;
}

```

При возникновении каких-либо проблем каждым из этих методов будет сгенерировано исключение. Использование исключений вместо возвращаемых значений методов объекта DB_common позволяет корректно инкапсулировать применение модуля DB. Тогда, если вам понадобится другой уровень абстракции (например, PEAR MDB), то вносить изменения за пределами класса Database не потребуется.

Теперь добавим метод update(), позволяющий легко манипулировать базой данных. В качестве параметров этот метод получает имя таблицы, ассоциативный массив имен полей (ключей) и значений, которые нужно обновить, а также строку для параметра WHERE оператора UPDATE. При конструировании параметра WHERE ассоциативный массив не используется, поскольку имеет смысл обеспечить возможность использования других операторов (например, =, LIKE, IN и т.д.), а также регулярных выражений (если они поддерживаются выбранной СУБД). Конструирование параметра WHERE, который был бы эффективным для различных платформ, является достаточно трудным делом. Поэтому соответствующую строку, передаваемую в качестве параметра, лучше всего сгенерировать за пределами класса Database.

```

function update($tableName, $arUpdates, $sWhere = null) {

    $arSet = array();
    foreach($arUpdates as $name => $value) {
        $arSet[] = $name . ' = ' . $this->conn->quoteSmart($value);
    }
    $sSet = implode(', ', $arSet);

    //убедитесь в корректности имени таблицы
    $tableName = $this->conn->quoteIdentifier($tableName);

    $sql = "UPDATE $tableName SET $sSet";
    if($sWhere) {

```

```

        $sql .= " WHERE $sWhere";
    }

$result = $this->conn->query($sql);

if(DB::isError($result)) {
    throw new Exception($result->getMessage(), $result->getCode());
}

//возвращается количество модифицированных строк
return $this->conn->affectedRows();
}

```

Метод `quoteSmart()` обеспечивает корректную обработку символов апострофа в значениях, передаваемых во втором параметре. Метод `quoteIdentifier()` предназначен для преобразования в корректную форму имени таблицы. Метод `update()` возвращает количество строк, которые были модифицированы.

Будьте внимательны! Если в функцию `update()` не была передана строка для параметра `WHERE`, будет модифицирована каждая строка таблицы.

Метод `insert()` аналогичен методу `update()` за исключением того, что в нем не используется оператор `WHERE`.

Кроме того, с помощью специального маркера метод `insert()` позволяет получить значения идентификаторов новых записей. Если значением массива `$arValues` является строка '`#id#`', то предполагается, что соответствующий столбец представляет собой первичный ключ данной таблицы. Это значение заменяется с помощью метода `nextID()`, вызываемого для последовательности `$tableName_id_seq`. Если был обнаружен специальный маркер, функция `insert()` вернет значение нового идентификатора. В противном случае возвращается количество записей, к которым эта функция была применена (обычно 1).

```

function insert($tableName, $arValues) {
    $id = null;

    $sFieldList = join(', ', array_keys($arValues));
    $arValueList = array();
    foreach($arValues as $value) {
        if(strtolower($value) == '#id#') {
            //получение следующего значения из последовательности
            //для данной таблицы
            $value = $id = $this->conn->nextID($tableName . "_id");
        }
        $arValueList[] = $this->conn->quoteSmart($value);
    }
    $sValueList = implode(', ', $arValueList);

    //проверка корректности имени таблицы
    $tableName = $this->conn->quoteIdentifier($tableName);

    $sql = "INSERT INTO $tableName ( $sFieldList ) VALUES ( $sValueList )";
    $result = $this->conn->query($sql);

    if(DB::isError($result)) {
        throw new Exception($result->getMessage(), $result->getCode());
    }
}

```

```
//возвращение идентификатора или количества записей
return $id ? $id : $this->conn->affectedRows();
}
```

Если одним из значений оказалась строка '#id#', то возвращается значение нового идентификатора. В противном случае возвращается количество строк, которые были задействованы при обработке запроса. При вызове этой функции необходимо точно знать, какого типа возвращаемое значение вы ожидаете, иначе возвращаемое значение 1 можно ошибочно принять за идентификатор, в то время как на самом деле оно может означать количество записей.

Этот класс гораздо более робастен, чем класс простого уровня абстракции, созданный в начале этой главы. Однако для полноты картины необходимо реализовать еще несколько свойств. Первое из них — поддержка транзакций.

Поддержка транзакций

Транзакции используются для группировки изменений базы данных и совместного принятия нескольких операций. Этот прием обычно используется в том случае, когда отдельные изменения не имеют смысла и при невыполнении любого из них отменяются все предложенные ранее модификации базы данных (в рамках одной транзакции).

С учетом этого при реализации операторов INSERT, UPDATE и DELETE в контексте поддержки транзакций запись в базу данных выполняется только после команды COMMIT. Если что-то не получилось, все изменения отменяются и никто из других пользователей базы данных даже не узнает о попытках ее обновления. Для поддержки транзакций в классе Database необходимо реализовать три метода: метод объявления начала транзакции, метод принятия предлагаемых изменений и метод прекращения транзакции и отмены всех сделанных в рамках нее изменений.

Метод startTransaction() задает контекст транзакции. Это означает информирование используемой базы данных о том, что все последующие операции будут выполняться в рамках одной транзакции. Если база данных не поддерживает транзакций, класс DB_common возвращает ошибку.

```
function startTransaction() {
    //функция autoCommit возвращает true/false
    //в зависимости от результата выполнения команды
    return $this->conn->autoCommit(false);
}
```

Обычно для большинства СУБД режим autoCommit по умолчанию включен. Это означает, что все операторы INSERT, UPDATE и DELETE автоматически обновляют информацию в базе данных. В таком состоянии нет необходимости явно подтверждать вносимые изменения. Поэтому для использования механизма обработки транзакций необходимо установить значение параметра функции autoCommit() равным false. Тогда выполнение всех операторов придется подтверждать явно.

Последовательность изменений в рамках транзакции можно завершить двумя способами: с помощью функции commit() для записи изменений в базу данных или с помощью функции abort() для их отмены и возврата в исходное состояние.

```
function commit() {
    $result = $this->conn->commit();

    if(DB::isError($result)) {
```

```

        throw new Exception($result->getMessage(), $result->getCode());
    }

    $this->conn->autoCommit(true);
    return true;
}

function abort() {
    $result = $this->conn->rollback();

    if(DB::isError($result)) {
        throw new Exception($result->getMessage(), $result->getCode());
    }

    return true;
}

```

Современные версии большинства платформ баз данных поддерживают транзакции. К ним относятся PostgreSQL, MySQL, Oracle, Microsoft SQL Server и некоторые другие. Если в рамках одной функции приходится вносить изменения в несколько таблиц базы данных, необходимо использовать транзакции, чтобы не нарушить целостность ссылок. Нарушение целостности ссылок означает наличие ссылок на сущности в других таблицах, которых на самом деле не существует (например, значение 235 столбца user_id, если в таблице user не существует пользователя с идентификатором id, равным 235). Более подробная информация о поддержке транзакций конкретной СУБД содержится в соответствующей документации.

Ранее упоминалось о необходимости реализации в классе Database нескольких важных свойств. Поддержка транзакций — одно из них. Второе касается способа открытия соединения с базой данных. В следующем разделе читатель познакомится с важным шаблоном проектирования, получившим название Singleton.

Шаблон проектирования Singleton

Шаблон проектирования Singleton обеспечивает возможность использования единственного экземпляра некоторого класса. То есть, для класса, разработанного в соответствии с этим шаблоном, может существовать лишь один экземпляр.

Этот шаблон используется в тех случаях, когда при инстанцировании класса необходимо выполнять множество вспомогательных операций либо когда существование нескольких объектов класса невозможно. В данном случае приложение работает лишь с одной базой данных, задаваемой в файле config.php в массиве \$cfg['db']['dsn']. Открытие соединения с базой данных обычно требует гораздо больше ресурсов, чем любые, даже самые сложные, запросы. Поскольку язык PHP не поддерживает многопоточность (сам интерпретатор PHP является многопоточным, но приложения, написанные на этом языке, — нет) и каждый запрос на страницу предполагает связь лишь с одной базой данных, нет нужды открывать несколько соединений.

Поэтому, чтобы избежать лишних операций, необходимо гарантировать единственность соединения. Это можно обеспечить следующим образом.

Во-первых, в класс Database добавим статический метод instance(). Он будет возвращать инстанцированный объект Database. Во-вторых, чтобы избежать случайного создания новых экземпляров класса, конструктор объявим в области private. В методе instance() необходимо создать статическую переменную, содержащую ссылку на инстанцированный объект Database. При первом вызове метод instance() значение этой переменной будет равно нулю, а при последующих она будет содержать ссылку на объект, созданный при первом вызове метода.

```

private function __construct($dsn = null) {
    global $cfg;

    if($dsn == null) $dsn = $cfg['db']['dsn'];
    println("DSN: $dsn");

    this->conn = DB::connect($dsn);

    if(DB::isError(this->conn)) {
        //Соединение не установлено. Генерируется исключение
        throw new Exception($this->conn->getMessage(), $this->conn->getCode());
    }

    //Данные всегда извлекаются в виде ассоциативного массива
    $this->conn->setFetchMode(DB_FETCHMODE_ASSOC);
}

static public function instance() {
    static $objDB;

    if(!isset($objDB)) {
        $objDB = new Database();
    }

    return $objDB;
}

```

Теперь конструктор стал закрытым, и любые попытки выполнения операторов вида `$db = new Database()` будут приводить к ошибке. Для получения объекта `Database` будет использоваться вызов `$db = Database::instance()`.

Напомним, что статические переменные сохраняют свое значение между вызовами функции.

Использование класса Database

Следующий фрагмент кода содержит хороший пример использования дополненного класса `Database`.

```

<?php

require_once('class.Database.php');

try {
    $db = Database::instance();
} catch (Exception $e) {
    // Продолжать не стоит...
    die("Нельзя соединиться с базой данных.");
}

$sql = "SELECT count(1) FROM mytable";
$count = $db->getOne($sql);
print "В таблице mytable имеется $count записей!<br>\n";

// начало транзакции
$db->startTransaction();

// выполнение SQL-запросов INSERT и UPDATE
try {

```

```

$arValues = array();
$arValues['id'] = '#id#';
$arValues['myval'] = 'бла бла бла';
$newID = $db->insert('mytable', $arValues);

print "Новая запись имеет ID $newID<br>\n";

// обновление только что созданной записи
$arUpdate = array();
$arUpdate['myval'] = 'foobar baz!';
$affected = $db->update('mytable', $arUpdate, "id = $newID");

print "Обновлено $affected записей<br>\n";

// запись изменений в базу данных
$db->commit();
} catch (Exception $e) {
    // при возникновении ошибок транзакция прерывается
    // и выводится сообщение об ошибке
    $db->abort();
    print "Возникла ошибка.<br>\n" . $e->getMessage();
}

?>

```

Внимательно ознакомьтесь с этим примером и разберитесь, как он работает. Не забывайте использовать блоки `try...catch`, чтобы перехватывать и корректно обрабатывать любые исключения, возникающие при установке соединения или выполнении запросов.

Резюме

Поскольку нижний уровень абстракции базы данных представляет достаточно гибкий и мощный пакет PEAR DB, разработанный класс `Database` можно использовать практически в любом проекте. Хотя не все запросы можно переносить на различные базы данных, возможность быстрого изменения типа базы данных существенно упрощает повторное использование компонентов в различных проектах.

В главе 11 будет построена рабочая система регистрации сообщений и отладки, работающая с классом `Database` и обеспечивающая средства для отладки проблем, возникающих при работе с базой данных, без вывода на экран детальной информации о запросе. Поскольку практически каждый запрос выполняется с помощью класса `Database`, можно обеспечить централизованный сбор данных о выполняемых запросах и вывод их в файл журнала или окно отладки.

Шаблон проектирования `Singleton` обеспечивает возможность использования единственного экземпляра класса. Этот прием позволяет снизить нагрузку на систему и повысить производительность приложения в тех случаях, если инстанцирование класса является ресурсоемкой операцией (например, установка соединения с базой данных) или пользователем класса является лишь один объект.

9

Интерфейс Factory

Из предыдущих глав вы узнали, что такое *шаблоны проектирования* и как их использовать при разработке приложений на PHP. В этой главе вы еще раз убедитесь, что шаблоны проектирования — не просто учебные упражнения. Их можно эффективно использовать в реальных приложениях. Иногда разработчики применяют шаблоны, даже не осознавая этого.

Эта глава познакомит читателя с *интерфейсом фабрики Factory* и типичными примерами его использования. Однако этими ситуациями его применение не ограничивается, он пригодится и при разработке ваших собственных приложений.

Для лучшего усвоения материала авторы приводят некоторые примеры кода, в которых используется интерфейс Factory. Мы также обсудим, как эти примеры можно улучшить благодаря новым возможностям PHP 5. По ходу изложения будут отмечены важные детали реализации.

Шаблон проектирования Factory

Шаблон проектирования *Factory* (см. книгу Крэга Лармана *Применение UML и шаблонов проектирования. Второе издание*, которая вышла в Издательском доме “Вильямс” в 2002 году) — это шаблон проектирования, решающий проблему создания (инстанцирования) сложных объектов. Но зачем усложнять себе жизнь, используя шаблон проектирования, если можно просто использовать оператор new, например в виде `$valiable = new MyClass();`? Существует две основных ситуации, в которых оправдана его применимость.

- **Гибкость выполнения программы.** Иногда невозможно заранее указать момент времени, когда должен быть инстанцирован конкретный объект. Выбор используемых объектов может зависеть от некоторых условий, реализуемых в процессе выполнения программы, в частности команд пользователя, параметров окружения (например, многоязычности) или даже поддержки программного обеспечения и аппаратных средств.

- **Абстракция.** Используя интерфейс Factory, обязанность создания экземпляра некоторого класса приложения можно делегировать его подклассам. Это означает, что приложение может базироваться на предположении, что родительские, *абстрактные*, классы будут использоваться как “посредники” для реальных действующих классов. Это позволяет разрабатывать приложение с учетом только абстрактных классов без обращения напрямую к представляющим их реальным классам.

Пример интерфейса Factory

Обеспечение доступа к базам данных для Web-приложений — довольно типичная задача. Как же ее решить, если необходимо обеспечить поддержку нескольких баз данных? Зачастую эту задачу нельзя решить наперед; многое зависит от возможностей или предпочтений клиента. Подобная проблема возникает при разработке приложений для широкого круга потребителей, в том числе при создании проектов с открытым кодом.

Предположим, приложение должно обеспечить доступ к базам данных MySQL, PostgreSQL и Interbase. Приведем упрощенные фрагменты кода, которые можно использовать для создания каждого из соединений.

- MySQL:

```
$query = mysql_query ($sql, $database_handle);
```

- Postgres:

```
$query = pg_query ($database_handle, $sql);
```

- Interbase:

```
$query = ibase_connect ($database_handle, $sql);
```

Старый школьный подход

Если переменная \$databasetype задает требуемый тип базы данных, то метод установки соединения можно определить с помощью оператора switch().

```
// запрос
switch(strtolower($databasetype)){
    case "mysql":
        $query = mysql_query ($sql, $database_handle);
        break;
    case "postgres":
        $query = pg_query ($database_handle, $sql);
        break;
    case "interbase":
        $query = ibase_connect ($database_handle, $sql);
        break;
    default:
        die ("Неизвестный тип базы данных:". $databasetype);
}
```

Этот код, без сомнения, будет работать. Но если позднее понадобится добавить поддержку SQL-сервера, его придется модифицировать. Нужно будет пройтись по всему коду и всюду, где необходимо, добавить следующий фрагмент.

```
case "mssql":
    $query = mssql_query ($sql, $database_handle);
```

Допустим, вам начали поступать жалобы, что программное обеспечение не работает с базой данных Interbase. Почему это происходит? База данных Interbase поддерживается. Внимательно изучив проблему, вы поняли, что заказчики подразумевают “Interbase”, а вводят firebird. Это действительно ошибка, поскольку продукт Interbase (от Borland) был так же выпущен, как программный продукт с открытым кодом под названием Firebird. Невзирая ни на что, придется вернуться к коду и добавить следующее.

```
case "interbase":
case "firebird":
    $query = ibase_query ($database_handle, $sql);
```

Хотя эти тривиальные изменения внести несложно, подобные модификации придется делать по всему коду, а не только в том фрагменте, где устанавливается соединение. Закрытие базы данных можно выполнить следующим образом.

```
// закрытие базы данных
switch(strtolower($databasetype)) {
    case "mysql":
        mysql_close ($database_handle);
        break;
    case "postgres":
        pg_close ($database_handle);
        break;
    case "interbase":
    case "firebird":
        ibase_close ($database_handle);
        break;
    case "mssql":
        mssql_close ($database_handle);
        break;
    default:
        die ("Неизвестный тип данных:". $databasetype);
}
```

Несложно заметить, что даже незначительное расширение функциональности требует существенных усилий — в самом деле, разработчику придется разыскать все места, куда необходимо внести изменения. Ситуация усложнится еще более, если вы вдруг передумаете закрывать соединение.

Хотя рассмотренный код синтаксически и технически корректен, он требует существенных модификаций при добавлении новых платформ баз данных. Например, если необходимо обеспечить поддержку базы данных SuperCoolDatabase, изменения придется внести во многие места основного кода. Поскольку разработчик — тоже человек, необходимость многократного изменения кода ведет к ошибкам и упущениям, которых лучше избежать. Возникшую проблему можно решить с использованием шаблона проектирования Factory.

Применение интерфейса Factory

В предыдущем примере показано, что разработчик зачастую не может заранее определить момент, когда понадобится поддержка новой базы данных. Как можно спланировать будущее расширение приложения и обеспечить удобство поддержки и “читабельность” кода?

Одним из возможных решений этой проблемы является использование шаблона проектирования Factory. Он обеспечивает абстракцию для создания объектов (т.е., экземпляров) на основе наследования. В шаблоне проектирования Factory определяется интерфейс, представляющий стандарт для инстанцирования нужных классов.

На рис. 9.1 показана диаграмма классов UML, иллюстрирующая реализацию шаблона Factory.

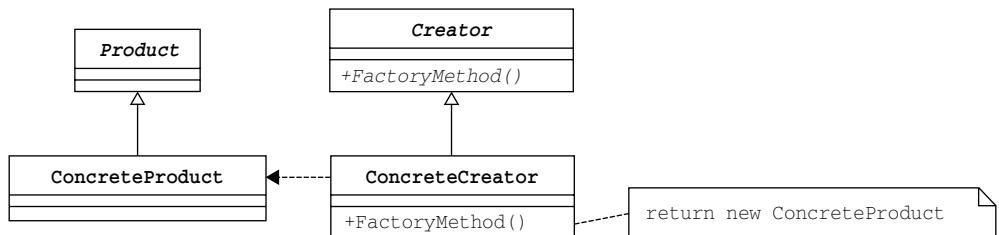


Рис. 9.1.

Главным компонентом этой диаграммы является класс *Creator*, в котором определяется интерфейс метода, реализующего фабрику — *FactoryMethod()*.

Реализация метода *FactoryMethod()* содержится в рабочем экземпляре класса *ConcreteCreator*. Классу *ConcreteCreator* (у которого может быть много разновидностей) делегируется обязанность с помощью метода *FactoryMethod()* инстанцировать экземпляр нужного класса *Product*, изображенного на диаграмме как *ConcreteProduct*. В классе *ConcreteCreator* можно определить и другие операции.

Использование интерфейса Factory на уровне абстракции базы данных

Сначала рассмотрим экземпляр класса, который нужно создавать в процессе работы приложения. Этот класс всегда должен быть в центре внимания разработчика системы. В следующем примере в зависимости от выбора базы данных генерируется специальное исключение. Поскольку невозможно точно определить, какую базу данных использует клиент, выбор типа исключения возлагается на интерфейс *Factory*.

Приведем краткий пример реализации базы данных PostgreSQL (рис. 9.2).

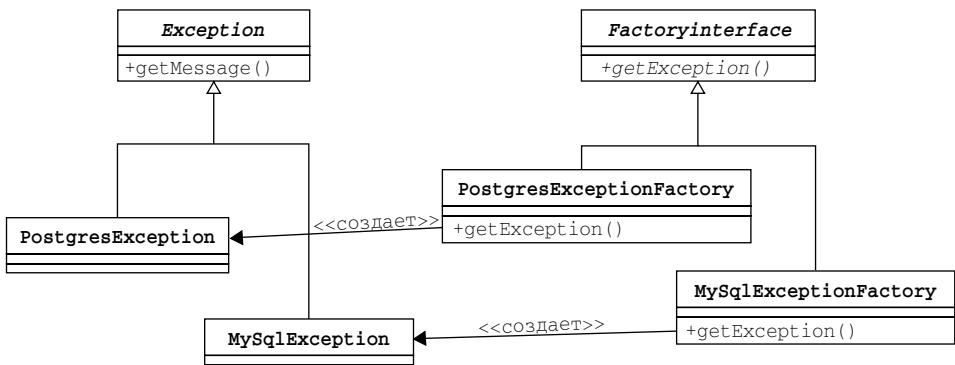


Рис. 9.2.

```

interface ExceptionFactory {
    public function getException($result=NULL, $message=NULL);
}

class PostgresExceptionFactory implements ExceptionFactory {
    public function getException($result=NULL, $message=NULL) {
        return new PostgresException($result, $message);
    }
}

class PostgresException extends Exception {

    function __construct($result=NULL, $message=NULL) {
        $code = 0;
        if ($message == NULL) {
            if ($result <> NULL) {
                $mess ge = pg_result_error($result);
                $code = pg_result_status($result);
            } else {
                $message = pg_last_error();
            }
        }
        parent::__construct($message, $code);
    }
}

class MySqlException|Factory implements ExceptionFactory {
    public function getException($result=NULL, $message=NULL) {

        return new MySqlException($result, $message);
    }
}

class MySqlException extends Exception {
    function __construct($result=NULL, $message=NULL) {
        if ($message == NULL) {
            $message = mysql_error();
        }
        $code = 0;
        if ($result <> NULL) {
            $code = mysql_errno($resource);
        }
        parent::__construct($message, $code);
    }
}

/*
 * Псевдокод
 */

// Создание фабрики
$factory = new PostgresExceptionFactory();
// Метод getException возвращает объект PostgresException
if (...что-то не так) {
    throw $factory->getException();
}

```

Давайте обсудим приведенную диаграмму и пример кода.

1. После того как клиент определится с используемой базой данных, ее поддержку можно обеспечить с помощью соответствующей реализации класса Exception-

Factory. Если такой реализации не существует, ее можно легко добавить в соответствии с требованиями интерфейса ExceptionFactory.

2. Если нужно сгенерировать исключение, это делается с помощью вызова метода getException() соответствующего класса ExceptionFactory.
3. В этом случае метод getException() представляет *интерфейс фабрики*. Так как этот метод реализован в конкретном классе, он возвращает корректный тип исключения для используемой базы данных.

Проектное решение представленного примера довольно прозрачно. Заметим, что интерфейс ExceptionFactory реализуется в подклассах PostgresExceptionFactory и MySQLExceptionFactory. Именно в этих классах инстанцируются конкретные исключения.

Многократное применение шаблона Factory

В реализацию рассматриваемого шаблона проектирования Factory можно добавить еще один интерфейс фабрики. Например, многие базы данных поддерживают концепцию *транзакций*, которые предполагают обработку целого блока взаимосвязанных запросов с обеспечением целостности используемой базы данных. В качестве такой базы данных может выступать Interbase.

Рис. 9.3 иллюстрирует поддержку транзакций с помощью реализации шаблона проектирования Factory.

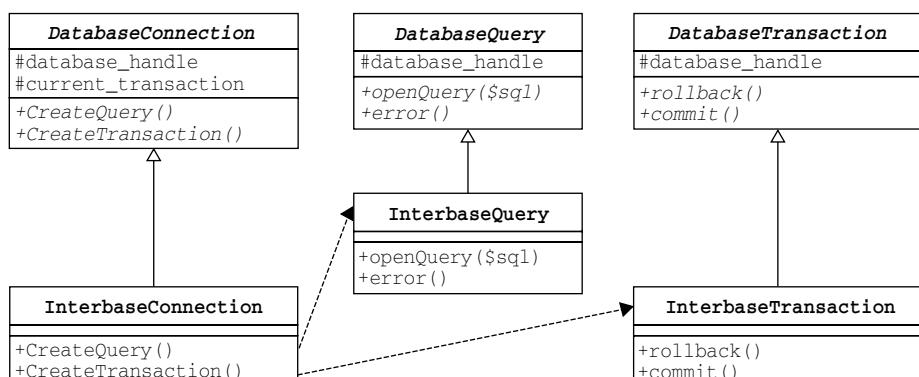


Рис. 9.3.

Класс InterbaseTransaction работает аналогично классу InterbaseQuery. Оба эти класса инстанцируются классом InterbaseConnection. Обработку транзакций можно обеспечить следующим образом.

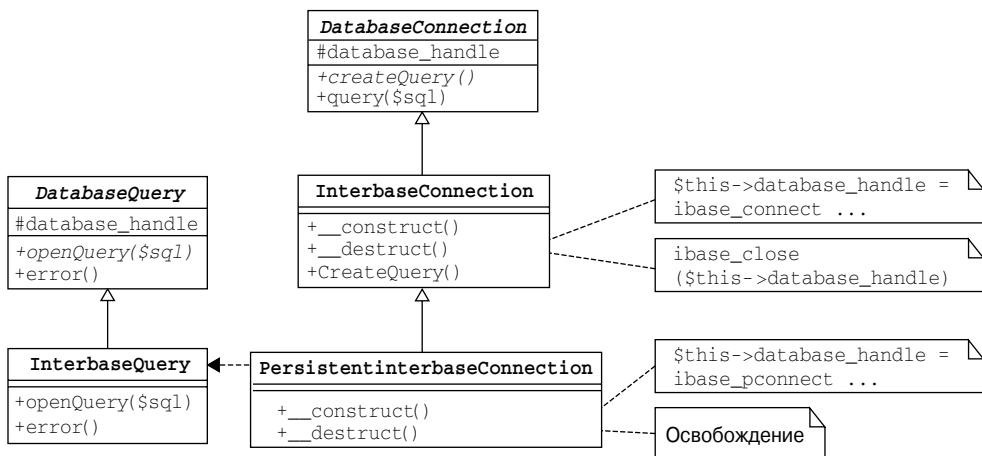
```

$con = new InterbaseConnection();
$strans = $con->reateTransaction();
$query = $con->query("delete from employee");
if ($query->error()) {
    $trans->rollback();
} else {
    $trans->commit();
}

```

Усовершенствование существующих классов

Наследование класса `InterbaseConnection` не только возможно, но и полезно. Это видно из рис. 9.4.



Puc. 9.4.

Обратите внимание на разницу между классами `InterbaseConnection` и `PersistentInterbaseConnection`. Во-первых, отметим, что методы `ibase_connect` или `ibase_pconnect` используются для установки реального соединения с базой данных сервера. Во-вторых, обратите внимание, что класс `PersistentInterbaseConnection` не требует закрытия физического соединения в деструкторе, потому что использование метода `pconnect` не требует вызова соответствующей функции `ibase_close()`.

Поскольку класс PersistentInterbaseConnection использует обычный класс запроса InterbaseQuery, а не специфический класс InterbasePersistentQuery, он, согласно шаблону Factory, полностью основывается на родовом интерфейсе InterbaseConnection.

В результате для замещения обычного соединения с базой данных Interbase постоянным соединением в классе `PersistentInterbaseConnection` перекрываются конструктор и деструктор класса `InterbaseConnection`. Это великолепный пример добавления функциональности за счет расширения существующей структуры класса, который одновременно является классическим примером использования шаблона проектирования Factory.

Резюме

В данной главе показано применение шаблона проектирования Factory, а также рассмотрены различные ситуации, в которых его следует применять.

Преимущества этого шаблона проектирования продемонстрированы на примере работы с базой данных. Показано, как код взаимодействия с конкретной базой данных можно легко локализовать в различных местах приложения. Это облегчает поддержку и модификацию программного продукта.

В следующей главе речь пойдет о программировании на основе событий. Вы узнаете, как реализовать обработку событий на основе ООП в PHP 5.

10

Управление событиями

Иногда приложение лучше рассматривать в терминах *событий* и управления ими, а не проектировать архитектуру с традиционной объектно-ориентированной точки зрения. Это не значит, что нужно отказаться от объектно-ориентированного подхода и полностью перейти на управление событиями. Правильнее будет сказать, что классы создаются на основе событий, существенных для приложения. Это более радикальный подход, но во многих прикладных задачах он предоставляет очень мощный механизм для разработки программной архитектуры всего проекта.

События происходят постоянно и довольно часто. Одними из лучших примеров программ, основанных на управлении событиями, являются приложения, которые мы используем каждый день для редактирования текстов и отправки электронной почты. Для конечного пользователя программ, написанных для операционной системы Windows, управление событиями — предмет первостепенной важности. Принцип работы большинства приложений Windows — соответствующим образом реагировать на действия пользователя. Для многих Web-приложений это, конечно, не верно. Но если в приложении важную роль играет графический интерфейс пользователя, такое приложение должно разрабатываться на основе принципов управления событиями.

Почти все действия, предпринимаемые пользователем и даже самими приложениями, можно рассматривать как события. Довольно сложным примером события, не инициированного пользователем, являются системные часы. Если вы хотите создать приложение, которое выполняло бы различные действия в зависимости от времени, необходимо определенные изменения времени фиксировать как события и привязывать их к коду.

Для любой проблемы существует множество вариантов решения, и в этом плане PHP 5 не является исключением. Для работы с событиями не существует никакого нового расширения или библиотеки, потому что при проектировании приложений на основе управления событиями меняется способ мышления, а не программная реализация. Другими словами, обработка событий не требует каких-то специальных базовых технологий. Давайте рассмотрим один из способов управления событиями в PHP.

Что такое события

Событием является любое действие, на которое может реагировать приложение. Скорее всего, вам уже приходилось разрабатывать приложения, которые в той или иной степени требовали обработки событий. Определение хода программы при выполнении некоторого условия — это некоторая форма принятия решения. Например, представьте себе, что вы создали базу данных, содержащую определенное количество записей, и хотите дать возможность клиентам работать с записями через Web-интерфейс. Пользователям нужно дать возможность хотя бы просматривать и изменять данные. Для этого понадобятся специальные функции.

Решение о том, какую функцию вызывать, целиком зависит от введенной пользователем информации. Соответственно, разрабатываемый интерфейс пользователя должен предоставлять различные варианты и возможность выбрать необходимые действия. После того как пользователь примет решение, приложение должно ответить вызовом соответствующих функций для обработки запроса.

Типичным примером является создание Web-страницы с двумя кнопками, где одна отвечает за просмотр записей, а другая позволяет их редактировать. В качестве механизма обработки событий подойдет следующий фрагмент кода.

```
switch($_GET['action']){
    case "редактирование":
        edit_record();
        break;
    case "просмотр":
        view_record();
        break;
}
```

В данном фрагменте интерпретируется информация из массива `$_GET` (другими словами, из пользовательского запроса), и на основе его содержания принимается решение о том, какие действия выполнять. Именно так организовано управление событием, которое происходит при нажатии кнопки.

Эта идея лежит в основе управления практически любым событием, какое только можно себе представить. Для тех, кто использует Outlook или другую программу-организатор, обычным делом являются события, генерируемые системой. Сообщения о встречах и напоминания появляются под управлением специальной системы запуска событий (например, показание системных часов в определенный момент времени).

Конечно, реализация кода обработки событий — это задача разработчика. Использование оператора `switch` не является единственным решением. Реагировать на события позволяет любая разновидность цикла, а в приведенном выше примере можно с успехом использовать оператор `if/else`.

“Что же это за ажиотаж вокруг событий?” — спросите вы. В конце концов, рассмотренный пример кажется довольно простым. Почему же нельзя просто оставить все, как есть? Рассмотренное выше проектное решение по обработке событий может привести к путанице в случае изменения или расширения приложения. Это происходит, если изначально простая совокупность действий, определяемая оператором `if`, разрастается до чудовищных размеров, затрудняя поддержку и модификацию кода. Такое часто происходит при усложнении приложения. Поэтому для обработки событий следует использовать простые методы, проверяющие только несколько очень простых условий и гарантирующие, что в будущем код не разрастется до размеров полноценного приложения уровня предприятия.

Использование ООП для управления событиями

Для реализации эффективного решения по управлению событиями в приложениях PHP лучше всего сделать шаг назад и на высоком уровне определить, как приложение должно себя вести. Более продуманный подход к реализации событий позволит легче управлять их поведением. Решением проблемы является применение объектно-ориентированного подхода для программирования на основе событий.

Использование классов и объектов не только поможет сохранить упорядоченность кода, но и позволит создать расширяемую программную архитектуру. А это уже плюс, потому что расширяемость приложений на сегодня является главным требованием. Есть и другие положительные моменты, которые будут обсуждаться детальнее далее в этой главе. Кроме того, применение объектно-ориентированного подхода наводит на мысль о существовании готового шаблона проектирования, который с успехом можно применять для управления событиями.

Давайте разберемся, что необходимо для управления событиями на основе объектно-ориентированного подхода. Тогда мы сможем понять, существует ли шаблон (или шаблоны), реализующий эту модель поведения.

Проектное решение по управлению событиями

При разработке приложения, основанного на управлении событиями, необходимо сделать три главных шага.

На первом необходимо определить, как фиксировать события, которыми необходимо управлять. Это сложно выразить на абстрактном уровне, потому что способ “перехвата” событий зависит непосредственно от их типа.

Затем необходимо определить, как обрабатывать перехваченные события. В зависимости от специфики рассматриваемого приложения необходимо поставить и решить определенное количество вопросов. Например, стоит определить, из скольких различных источников будут “приходить” события в управляющий модуль. Также потребуется решить, сколько классов событий будет обрабатывать приложение — один или несколько — и как обработчик будет их эффективно различать.

Наконец, необходимо спроектировать управляющие модули (обработчики) для каждого события. Количество типов откликов на каждое событие практически не ограничено. В качестве отклика может выступать любое действие — от направления пользователя на новую страницу, изменения записи до генерации новых событий.

По существу, мы описали работу класса-реактора, или *диспетчера* событий, который рассматривает событие и обеспечивает вызов соответствующего обработчика. Давайте рассмотрим эту идею детальнее.

Возвращаясь к первоначальному примеру, предположим, что пользователь хочет изменить либо просмотреть записи в той или иной форме. Результатом введенного пользователем запроса может быть следующий адрес URL.

`http://myserver/interface.php?event=edit`

Из этого адреса URL ясно, что запускаемое событие — `edit`. Это событие передается классу-диспетчеру, который определяет, какой из обработчиков событий необходимо вызвать (в данном случае, проверяя значения, связанные с массивами `$_GET` или `$_POST`). Собственно обработка запроса будет выполняться соответствующим модулем, а не диспетчером.

Обработчики сами по себе представляют элементы объектно-ориентированной программы (их можно разработать в качестве упражнения по ООП). В самом деле, одним из эффективных путей создания обработчиков является наследование общего родительского класса `handler`. Использование наследования упрощает создание новых управляющих модулей, так как функции, общие для всех обработчиков, определяются внутри родительского класса. Например, код для установления связи с базой данных будет доступен для тех подклассов класса `handler`, которым может понадобиться такое соединение.

Для начала построим диаграмму, определяющую последовательность действий для данного примера (рис. 10.1).

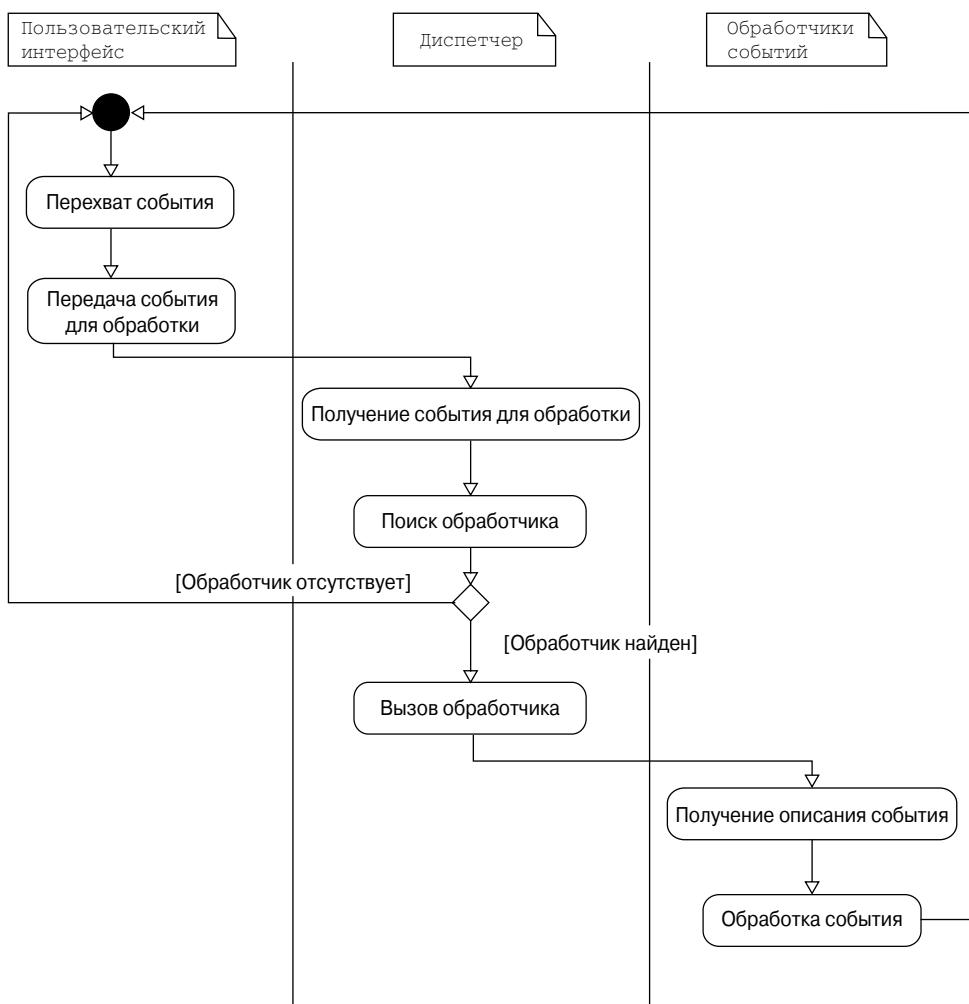


Рис. 10.1.

Это очень простое представление, но целью построения этой диаграммы является совсем не изящность реализации. Можно привести практически неограниченное

количество аргументов в пользу этого проектного решения. Среди них безопасность, поддержка баз данных, возможность работы с несколькими источниками событий. Это лишь некоторые из мотивов, которые необходимо принимать во внимание при выработке проектного решения на этом этапе.

Имея эту базовую модель, можно построить диаграмму классов, реализующую эту последовательность действий (рис. 10.2).

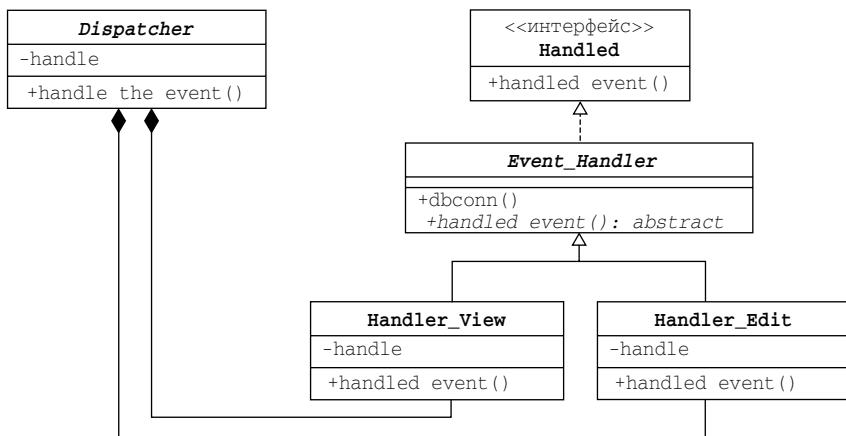


Рис. 10.2.

На данном этапе не нужно заботиться о программном коде пользовательского интерфейса. Требования к нему должен сформулировать конечный пользователь, а не архитектор программной системы. Сосредоточимся на диаграмме классов, отображающей взаимодействие *диспетчера* и *обработчиков*. Это приложение обеспечивает лишь базовую функциональность, достаточную для демонстрации возможностей и преимуществ парадигмы управления событиями.

Реализация решения

Приведенная выше диаграмма классов демонстрирует использование интерфейса для реализации каждого обработчика событий. Применение интерфейсов для обеспечения реализации определенных методов в каждом из классов-обработчиков является хорошим приемом, позволяющим стандартизировать способ построения обработчиков. Это очень важно с учетом возможности расширения приложения и развития его функциональности. Наличие интерфейса обеспечит реализацию концепции “*plug and play*” и сделает код легко расширяемым.

Кроме расширяемости, есть и другие преимущества. Ниже будет рассказано о том, как с использованием этой модели обеспечить безопасность на уровне отдельных элементов приложения. Но сначала рассмотрим программный код всех классов.

class.Dispatcher.php

Начнем с класса *Dispatcher*, код которого выглядит следующим образом.

```

<?php
require_once('class.Event_Handler.php');
require_once('class.Handler_View.php');
require_once('class.Handler_Edit.php');
  
```

```

class Dispatcher
{
    private $handle;

    function __construct($event_handle) {
        $this->handle=$event_handle;
    }
    function handle_the_event() {
        $name = "handler_($this->handle)";
        if (class_exists("$name")){
            $handler_obj=new $name($this->handle);
            $response=$handler_obj->handled_event();
            return $response;
        } else {
            echo "Невозможно обработать событие!";
        }
    }
}
?>
```

Конструктор отвечает за получение дескриптора события и сохранение его внутри класса для последующего использования. Дескриптор события может быть использован в процессе обработки, поэтому его значение передается всем возможным обработчикам. Конечно, обработчикам может также потребоваться дополнительная информация.

При обращении к базе данных пользователь может запросить записи из определенного диапазона. В таком случае ограничения диапазона можно перехватывать как часть события. Это демонстрирует следующий адрес URL.

<http://myserver/interface.php?event=edit&first=2&last=7>

Значения параметров можно хранить в некотором массиве (например, `$event_parameters`), заполняемом в конструкторе. В дальнейшем эти значения можно передать обработчику, который будет возвращать записи из определенного диапазона. Мы будем иметь дело с событиями, характерными для рассматриваемого примера, но вы получите общее представление о принципах обработки событий.

Вид функции `handle_the_event()` (а также ее имя) зависит от приложения и разработчика (и, соответственно, от конкретного соглашения по именованию классов-обработчиков). Если некто захочет добавить новую функциональность в ваше приложение, ему придется именовать обработчики следующим образом.

```

class Handler_(Unique_Event_Handle)
{
    //Обработка события с дескриптором Unique_Event_Handle
}
```

В рассматриваемом случае требуется только два обработчика: `Handler_View` и `Handler_Edit`. Метод `handle_the_event()` в диспетчерском классе проверяет, существует ли соответствующий обработчик в пространстве имен PHP, и если да — создает объект, соответствующий данному обработчику. Если нужного обработчика не существует, выдается сообщение.

`echo "Невозможно обработать событие!" ;`

Обратите внимание, что при инстанцировании экземпляра обработчика конструктору передается дескриптор события. Таким же способом обработчику можно передать массив значений.

`$handler_obj=new $name($this->handle);`

Не будем беспокоиться о действиях обработчика. Диспетчер должен передать соответствующему обработчику всю накопленную информацию о событии, не заботясь при этом, как она будет использована.

Следующий шаг состоит в вызове метода `handled_event()` для получения отклика от обработчика.

```
$response=$handler_obj->handled_event();
```

Как видно из диаграммы классов, существование и точный формат функции `handled_event()` обеспечиваются интерфейсом. Как следствие, диспетчер может без риска вызвать метод `handled_event()` для используемого в данный момент обработчика, чтобы получить ответ, даже если этот метод просто возвращает значение `NULL`. После получения ответа от обработчика работа диспетчера завершается.

Теперь обратимся к обработчикам событий.

interface.Handled.php

Этот интерфейс гарантирует реализацию функции `handled_event()` в каждом создаваемом классе-обработчике. Как было отмечено ранее, это важно с точки зрения диспетчера. Если рассматривать ситуацию более глобально, то таким способом можно закрепить существование любых других важных методов, имеющих отношение к обработчикам событий. Такие методы можно реализовать в родительском обработчике и при необходимости переопределить (перекрыть) в дочернем, а можно просто объявить абстрактными, выполняя их реализацию лишь в дочерних классах. Как можно заметить из приведенного ниже кода интерфейса, в нашем примере предполагается лишь существование метода `handled_event()`.

```
<?php
interface Handled
{
    abstract function handled_event();
}
?>
```

Метод `handled_event()` должен быть реализован либо в родительском классе обработчика событий, либо в его потомках. Сначала рассмотрим родительский класс.

Class.Event_Handler.php

Родительский класс на самом деле является утилитой класса. В нем указывается, как обрабатывать низкоуровневые события, которые могут происходить в приложении. В данном случае он создает соединение с базой данных. Файл `common_db.inc` содержит функцию `db_connect()`, которая создает соединение с базой данных. Читателям предлагается самостоятельно реализовать этот метод. Как бы то ни было, если вы создаете серьезное приложение, лучшим решением проблемы будет использование уровней абстракции баз данных, которые описаны в главе 8 **“Уровни абстракции базы данных”**.

В зависимости от решаемой проблемы в этом классе можно реализовать любое количество соответствующих методов. Приведем код для простого класса `Class.Event_Handler.php`.

```
<?php
require_once("interface.Handled.php");
require_once('common_db.inc');
```

```

abstract class Event_Handler
{
    function dbconn(){
        $link_id=db_connect('sample_db');
        return $link_id;
    }
    abstract function handled_event();
}
?>

```

Поскольку не известно, какие действия должно повлечь за собой каждое событие, функция `handled_event()` объявлена абстрактной, а ее реализация отложена до момента создания конкретных классов-обработчиков. Это можно сделать, поскольку сам класс объявлен абстрактным. Это не влияет на само приложение, потому что вам никогда не придется создавать объекты этого класса — диспетчер определит, к объекту какого класса-обработчика обратиться напрямую, и выполнит эту операцию.

`class.Handler_View.php`

Этот дочерний класс демонстрирует функциональность, присущую типичному обработчику. Обратите внимание на то, что в этом классе необходимо реализовать функцию `handled_event()`, так как она была объявлена абстрактным методом в родительском классе. В данном случае будет показано, как просмотреть несколько записей из базы данных.

```

<?php
require_once('class.Event_Handler.php');

class handler_View extends Event_Handler
{
    private $handle;

    function __construct($event_handle){
        $this->handle=$event_handle;
    }

    function handled_event(){
        echo "Событие, $this->handle, обработано.
              <BR> Это все, что нужно!<BR><BR>Записи следующие: <BR><BR>";

        $id=parent::dbconn();
        $result=pg_query($id, "SELECT * FROM user");
        while($query_data=pg_fetch_row($result)){
            echo "'", $query_data[1], ", - ", $query_data4, "<br>";
        }
    }
}
?>

```

Конечно, этот код нельзя использовать для решения практической задачи — это всего лишь пример. Если вы хотите проверить данный код, просто добавьте его к какой-либо программе, работающей с базой данных, и измените по своему усмотрению SQL-запрос.

Обратите внимание, что переданный в конструктор дескриптор события сохраняется в локальной переменной, являющейся закрытым членом класса. Эта переменная на самом деле не используется в данном примере. Здесь лишь наглядно продемонстрировано, как при необходимости передавать параметры обработчикам.

Class.Handler_Edit.php

Теперь познакомимся с другим дочерним обработчиком. Этот пример обработчика в ответ на запрос по редактированию просто возвращает пользователю сообщение.

```
<?php
require_once('class.Event_Handler.php');

class handler_Edit extends Event_Handler
{
    private $handle;

    function __construct($event_handle) {
        $this->handle=$event_handle;
    }

    function handled_event () {
        echo "Обработано действительно событие $this->handle ! <BR>";
    }
}
?>
```

Как обычно, конструктор получает дескриптор события и сохраняет его локально. В данном случае это значение действительно используется в процессе обработки событий. Правда, реализация метода `handled_event()` не представляет собой ничего особенного; просто возвращается сообщение, подтверждающее, что событие было обработано.

Вот и все. Теперь рассмотрим, как модифицировать приложение и получить хорошие результаты ценой минимальных усилий.

Обеспечение безопасности

Допустим, необходимо ограничить функциональность приложения в зависимости от прав доступа пользователя (довольно типичное требование). К примеру, определенным доверенным пользователям нужно позволить вносить любые изменения, а остальных ограничить возможностью чтения без внесения изменений в содержание.

После успешной аутентификации пользователя приложение создает новый сеанс. Затем переменным сеанса можно присвоить значения, соответствующие статусу пользователя. В данном случае мы использовали переменную сеанса, содержащую имя пользователя. Это означает, что информация о пользователе доступна практически во всем приложении, в том числе и в обработчиках событий.

Добавьте код следующего метода в дочерние классы обработчиков событий, которые вы хотите использовать для ограничения доступа. Как видите, метод просто проверяет имя пользователя в текущем сеансе, и если оно не является приемлемым, метод считает пользователя неуполномоченным выполнять данные операции.

```
function secure_handler(){
    if($_SESSION['name'] == "Давид") {
        $this->handled_event();
    } else {
        echo "Извините, $_SESSION['name'], Вы не авторизованы!";
    }
}
```

Чтобы гарантировать применение безопасного обработчика, нужно изменить одну строку в диспетчере. Откройте файл и измените строку

```
$response=$handler_obj->handled_event();
```

на следующую:

```
$response=$handler_obj->secure_handler();
```

В соответствии с установленными принципами проектирования существование этого метода в классах-потомках необходимо гарантировать с помощью интерфейса. Это означает, что обработчик-предок должен как минимум содержать объявление абстрактной функции `secure_handler()`.

Добавьте в интерфейс следующую строку.

```
abstract function secure_handler();
```

Невозможно точно знать, права какого уровня требуются для каждого события; они могут изменяться в зависимости от ситуации. Соответственно обработчик-родитель не-посредственно не следит за обеспечением безопасности. Он делегирует обязанность реализации безопасности классам-потомкам с помощью абстрактного метода.

```
abstract function secure_handler();
```

Интерфейс определяет, что в каждом обработчике событий должны быть реализованы оба метода — `handled_event()` и `secure_handler()`. Чтобы определить, разрешать ли обработку события, каждый метод `secure_handler()` может считывать требуемые ему параметры из файла, поддерживаемого администратором. Конечно, этот файл может быть зашифрован или храниться в безопасном месте. Права на его редактирование должен иметь только администратор.

Вот небольшой пример пользовательского интерфейса, при работе с которым может использоваться рассмотренное выше безопасное приложение, основанное на обработке событий.

```
<?php
require_once('class.Dispatcher.php');
?>
<HTML>
    <HEAD>
        <TITLE>Безопасное приложение, основанное на обработке событий</TITLE>
    </HEAD>
    <BODY>
        <FORM method="GET" ACTION=<?=$_SERVER['PHP_SELF']?>>
            <INPUT type="submit" name="event" value="View">
            <INPUT type="submit" name="event" value="Edit">
        </FORM>
    </BODY>
</HTML>

<?php
function handle(){
    $event = $_GET['event'];
    $disp = new dispatcher($event);
    $disp->handle_the_event();
}

session_start();
$_SESSION['name']="Горацио";

handle();
?>
```

Переменные сеанса, как правило, устанавливаются во время регистрации пользователя, но для краткости мы предположим, что доступ к приложению хочет получить

пользователь Горацио (это жестко зафиксировано в программном коде). Проверим, сможет ли Горацио использовать функцию редактирования. Если Горацио щелкнет на кнопке **Edit**, в браузере появится следующее сообщение (рис. 10.3).

Как видите, попытки Горацио редактировать записи сразу же пресекаются, потому что изменения может вносить только Давид.

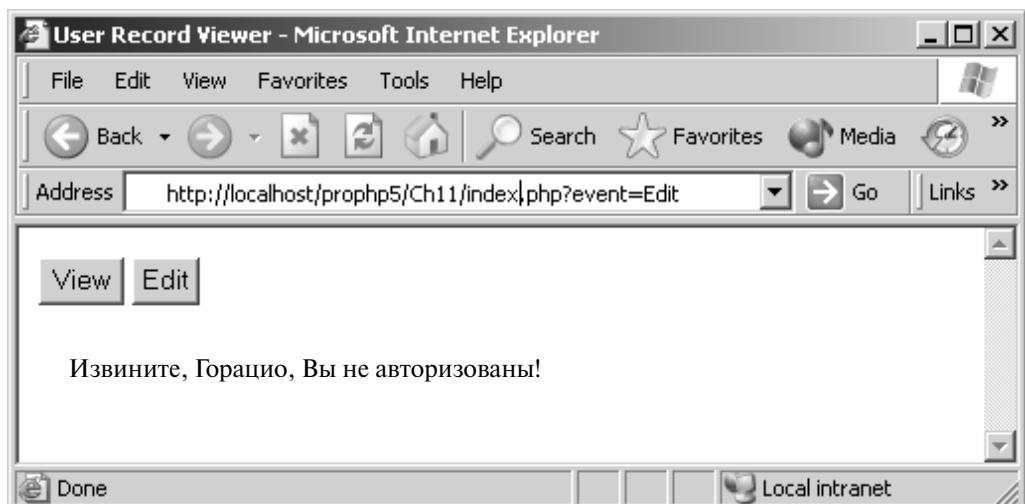


Рис. 10.3.

Теперь читателю понятны преимущества такого подхода. На примере обеспечения безопасности было показано, насколько просто модифицировать приложение в соответствии с любыми возникающими требованиями.

Остановитесь и подумайте

При разработке грамотного корпоративного приложения на основе обработки событий необходимо решить множество проблем, например вопрос регистрации обработчиков. При подключении пользователя к узлу (подключение в этом случае может быть рассмотрено как событие, перехватываемое приложением), приложение автоматически должно проверить, какие обработчики доступны данному пользователю. Это достигается путем использования обработчика, доступного по умолчанию. Он находит в базе данных список допустимых для пользователя обработчиков и возвращает эту информацию на уровень представления.

Вооруженный этой информацией, уровень представления будет “знать”, какие кнопки представлять в распоряжение пользователя. Администратору будет доступно много кнопок, в то время как пользователь сможет увидеть только кнопку **View**. Если доступ ограничен только кнопкой просмотра, пользователь, не обладающий соответствующими правами, не сможет злонамеренно изменить документ.

Вторая проблема, которую следует решить перед использованием этого метода в крупных разработках (хотя и тесно взаимосвязанная с первой), состоит в создании диспетчера, включающего в проект соответствующие файлы PHP. Ввиду того, что диспетчер создает объекты классов, соответствующих обработчикам, в проект необходимо

включить соответствующие классы. В рассмотренном примере мы вручную включили в проект все необходимые классы-обработчики. Для больших приложений такой подход является нерациональным, потому что можно включить сотню классов, а использовать только один из них.

Один из способов решения этой проблемы состоит в выделении определенных обработчиков как доступных по умолчанию и подключении остальных в зависимости от статуса пользователя. Каждому пользователю в базе данных среди других свойств соответствует поле, содержащее список доступных ему обработчиков. Эту информацию можно использовать для автоматического включения соответствующих обработчиков в файл диспетчера при добавлении соответствующих кнопок в пользовательский интерфейс.

Наконец, различные события могут генерироваться разными сущностями, поэтому стоит уделить внимание добавлению функциональности по определению источника (генератора) данного события и возвращению соответствующего ответа. Это позволит определить, в каком виде возвращать ответ: в машинно-ориентированном формате XML или ориентированном на человека формате HTML, в зависимости от того, какая сущность сгенерировала событие.

Для реализации описанного подхода используется шаблон Reactor. Он обеспечивает обработку событий от многих источников и заслуживает внимания читателя, если вы серьезно настроены использовать события. В начале главы упоминалось о существовании подобного шаблона проектирования. Этот шаблон очень близок к описанному в данной главе подходу, но обладает некоторыми дополнительными возможностями, позволяющими обрабатывать события от различных источников.

Резюме

Эта глава служит введением в область программирования на языке PHP на основе обработки событий. В результате ее изучения у читателя должно сложиться общее представление о том, как применять принципы управления событиями, используя объектно-ориентированный подход.

Создание приложений, основанных на обработке событий, — это проблемно-ориентированная область, требующая внимания к среде, в которой будет работать система. Обработка событий применяется во многих областях, но наилучшим примером являются программы с графическим интерфейсом.

В данной главе было показано, что мощный и изящный механизм наследования, присущий объектно-ориентированному программированию, можно эффективно использовать при разработке приложений на основе управления событиями.

В следующей главе мы рассмотрим реализацию эффективных механизмов регистрации событий (журналов) и отладки, оказывающих существенную помощь разработчикам на стадиях проектирования и тестирования проекта.

11

Регистрация событий и отладка

Надежные механизмы регистрации событий и отладки позволяют сэкономить массу времени, необходимого на выявление и устранение проблем, возникающих при разработке каждого приложения.

Журналы событий помогают анализировать работу приложения и те аспекты поведения, которые не фиксирует журнал Web-сервера, например нагрузку, выполнение конкретных операторов SQL или сообщения, характерные для данного приложения.

Механизм отладки позволяет проверять значения переменных, отслеживать выполнение условий циклов, определять состояние приложения во время его работы без использования операторов `print`, которые приходится удалять перед передачей системы заказчику. Проработав код данной главы, читатель получит в свое распоряжение набор классов, которые окажутся полезными в каждом проекте.

Создание механизма регистрации событий

Главной целью регистрации событий является получение возможности отследить работу приложения и его эффективность за некоторый период времени. Хорошо написанный журнал событий позволяет проводить ретроспективный анализ поведения системы и определять необходимость модификации кода. Для получения этой информации используются три основных механизма.

Запись в файл

Простейший способ регистрации событий — заносить информацию в файл. Обычно пользователь, для которого запущен Web-сервер, не обладает правом записи в файловую систему. Как правило, ему доступен только каталог временных файлов (для большинства UNIX-систем это `/tmp`), а это не лучшее место для хранения файлов журнала. Чтобы дать возможность пользователю Web-сервера записывать информацию в файл, создайте специальный каталог для хранения файлов журнала.

Создайте папку logs того же уровня, что и каталог документов приложения, и удостоверьтесь, что пользователь Web-сервера обладает правами записи в эту папку.

```
/www
|
+---./mysite
|
+---htdocs
|   |
|   +---images
|   |
|   +---css
|
+---logs
```

Пример записи в файл

Для инициализации записи в файл в PHP используется функция `fopen()`, параметрами которой являются дескриптор файла и “цель” открытия файла (в данном случае – для записи). Непосредственно для записи данных в файл используется функция `fwrite()`, а для закрытия файла – функция `fclose()`. Следующий пример кода демонстрирует создание простого метода `logMessage()`, который можно использовать для записи информации в заранее известный файл журнала.

```
<?php

function logMessage($message) {
    $LOGDIR = '/www/mysite/logs/';
    $logFile = $LOGDIR . 'mysite.log';
    $hFile = fopen($logFile, 'a+'); //открытие для добавления и
                                    //создания файла, если
                                    //он не существует
    if(! is_resource($hFile)) {
        printf("Нельзя открыть %s для записи. Проверьте разрешения.", $logFile);
        return false;
    }
    fwrite($hFile, $message);
    fclose($hFile);

    return true;
}

logMessage("Привет, журнал!\n");
?>
```

Если правильно установить все права для каталога `$LOGDIR` и запустить этот код, в файле журнала `mysite.log` появится сообщение “Привет, журнал!”. Для удовлетворения элементарных потребностей этого будет достаточно, но для реальных приложений придется отслеживать гораздо больше информации. Файлы журнала предназначены для автоматического анализа специальными приложениями. В них необходимо отслеживать время и дату каждого сообщения, отмечать важность сообщения и фиксировать, какой модуль его сгенерировал.

Класс Logger

В следующем примере приводится класс, реализующий механизм ООП, для генерации более детализированного файла журнала, чем функция из предыдущего раздела. Создайте файл с именем class.Logger.php. В нем будет содержаться класс Logger, позволяющий записывать информацию в табличном формате в текстовый файл на сервере. При этом отслеживается время генерации сообщения, его уровень, само сообщение и имя модуля (при желании). В качестве имени модуля может выступать любая строка, помогающая идентифицировать, в какой части приложения было сгенерировано сообщение. Приведем код класса Logger.

```
<?php

//Уровни регистрации. Чем выше номер, тем меньше критичность сообщения
//Диапазоны заданы так, чтобы позднее можно было
//добавить новые уровни
define('LOGGER_DEBUG', 100);
define('LOGGER_INFO', 75);
define('LOGGER_NOTICE', 50);
define('LOGGER_WARNING', 25);
define('LOGGER_ERROR', 10);
define('LOGGER_CRITICAL', 5);

class Logger {

    private $hLogFile;
    private $logLevel;

    //Закрытый конструктор.
    //При создании класса использован шаблон Singleton
    private function __construct() {
        global $cfg; //массив данных о конфигурации системы из внешнего файла

        $this->logLevel = $cfg['LOGGER_LEVEL'];
        $logFilePath = $cfg['LOGGER_FILE'];

        if(! strlen($logFilePath)) {
            throw new Exception('Не задан путь к файлу журнала ' .
                'в конфигурации системы.');
        }

        //Создаем дескриптор файла журнала.
        //Запрещаем вывод сообщений об ошибках PHP.
        //Будем выводить собственные сообщения, генерируемые исключениями.
        $this->hLogFile = @fopen($logFilePath, 'a+');
        if(! is_resource($this->hLogFile)) {
            throw new Exception("Указанный файл журнала $logFilePath " .
                'нельзя открыть или создать' .
                'для записи. Проверьте права доступа.');
        }
    }

    public function __destruct() {
        if(is_resource($this->hLogFile)) {
            fclose($this->hLogFile);
        }
    }

    public static function getInstance() {
        static $objLog;
```

```

if(!isset($objLog)) {
    $objLog = new Logger();
}

return $objLog;
}

public function logMessage($msg, $logLevel = LOGGER_INFO, $module = null) {
    if($logLevel <= $this->logLevel) {
        $time = strftime('%x %X', time());
        $msg = str_replace("\t", '    ', $msg);
        $msg = str_replace("\n", ' ', $msg);

        $strLogLevel = $this->levelToString($logLevel);

        if(isset($module)) {
            $module = str_replace("\t", '    ', $module);
            $module = str_replace("\n", ' ', $module);
        }

        //В журнале время, уровень, текст сообщения и имя модуля
        //разделяются символом табуляции,
        //сообщения начинаются с новой строки
        $logLine = "$time\t$strLogLevel\t$msg\t$module\n";
        fwrite($this->hLogFile, $logLine);
    }
}

public static function levelToString($logLevel) {
    switch ($logLevel) {
        case LOGGER_DEBUG:
            return 'LOGGER_DEBUG';
            break;
        case LOGGER_INFO:
            return 'LOGGER_INFO';
            break;
        case LOGGER_NOTICE:
            return 'LOGGER_NOTICE';
            break;
        case LOGGER_WARNING:
            return 'LOGGER_WARNING';
            break;
        case LOGGER_ERROR:
            return 'LOGGER_ERROR';
            break;
        case LOGGER_CRITICAL:
            return 'LOGGER_CRITICAL';
        default:
            return '[unknown]';
    }
}
}

?>

```

Код класса довольно ясен. Обратите внимание, что конструктор объявлен в закрытой области класса, чтобы предотвратить создание нескольких файлов журнала во время выполнения одного запроса. Единственный экземпляр дескриптора файла создается в методе getInstance(). Такое проектное решение соответствует шаблону Singleton, о котором шла речь в главе 8 “Уровни абстракции базы данных”.

В первых нескольких строках кода задаются константы, определяющие уровни сообщений. С их помощью можно управлять выводом информации в журнал. На первых этапах разработки приложения в журнале целесообразно регистрировать детальную информацию, а на последних стадиях — только некоторые данные (в целях экономии системных ресурсов). Уменьшение объемов записываемой информации повышает значение критерия “сигнал/шум”: в фиксированном числе строк содержится большее количество полезных сведений. Для полноценного работающего приложения отладочная информация не нужна, она только затрудняет поиск нужных данных.

В конструкторе из глобального массива конфигурационных параметров извлекаются два элемента: имя файла журнала и текущий уровень регистрации. В данном примере предполагается, что для каждого приложения существует только один файл журнала. Ниже в этой главе будет показано, как расширить этот класс и обеспечить большую гибкость и функциональность при выборе места записи отладочной информации.

Уровень регистрации хранится в закрытой переменной, которая в дальнейшем будет использоваться для определения того, какую информацию записывать в файл журнала. Затем проверяется, действительно ли существует файл по указанному адресу и имеет ли данный пользователь Web-сервера право записи в файл журнала. Параметр `a` функции `fopen()` означает, что файл открывается с целью добавления информации, а если такого файла не существует, он создается. Дескриптор файла сохраняется в другой закрытой переменной. Если по какой-то причине не удается создать дескриптор файла, конструктор генерирует исключение.

Любой вызов метода `Logger::getInstance()` нужно помещать в блок `try...catch`, поскольку при его выполнении возможны различные непредвиденные обстоятельства, связанные с отсутствием прав на запись в файл журнала или нехваткой дискового пространства. Это не должно помешать работе всего приложения. Подобные исключения необходимо “отлавливать” и принимать соответствующие меры, например, отправляя системному администратору электронные сообщения. В главе 14 описывается рабочий класс для электронной почты.

В этом классе также есть деструктор (строки 41–45), закрывающий открытый файл. Если путь к файлу был указан некорректно, дескриптор файла будет ссылаться на не существующий ресурс. Даже в случае генерации исключения в конструкторе деструктор вызывается для удаления объекта из памяти. Поэтому необходимо удостовериться, что при закрытии файла не возникнет новая ошибка, связанная с попыткой закрытия несуществующего файла.

Как и класс `Database`, описанный в главе 8, метод `getInstance()` класса `Logger` содержит статическую переменную, в которой хранится экземпляр класса между вызовами метода `getInstance()`. При первом вызове этого метода переменная принимает значение `null`, что инициирует создание экземпляра класса `Logger`. Если указан неверный путь к файлу журнала, будет сгенерировано исключение. При последующих вызовах метода в этой статической переменной будет храниться ссылка на экземпляр, созданный при первом вызове. Именно этот исходный экземпляр и будет возвращать метод `getInstance()`. Применение шаблона `Singleton` позволяет сэкономить системные ресурсы, требуемые для открытия файла.

Всю основную работу выполняет метод `logMessage()`. Ему передаются следующие параметры: текст сообщения, уровень регистрации и имя модуля (необязательный параметр). Если значение уровня регистрации, указанное среди конфигурационных параметров системы (и присвоенное в конструкторе закрытой переменной-члену `$logLevel`), не ниже значения уровня регистрации, передаваемого в качестве второго

параметра в эту функцию, создается запись в файле журнала. Если текущий уровень регистрации для данного приложения ниже значения второго параметра функции, запись не создается. Этот механизм позволяет определить, какую информацию заносить в журнал регистрации. Каждому сообщению, передаваемому методу `logMessage()`, должен соответствовать свой уровень регистрации, определяющий степень важности сообщения. Например, если в целях отладки в журнал должно заноситься значение некоторой переменной, второй параметр метода `logMessage()` должен принимать значение `LOGGER_DEBUG`. В процессе промышленной эксплуатации системы на сервере параметр `$cfg['LOGGER_LEVEL']` должен принимать значение `LOGGER_WARNING` или выше. При этом в файл журнала не будут выводиться отладочные сообщения. Регистрироваться будут только ошибки или предупреждения.

В строке 60 функция `logMessage()` генерирует временную метку с использованием функции PHP `strftime()`. В качестве первого параметра функции `strftime()` используется строка, определяющая формат вывода времени, а второй параметр задает системное время UNIX. В строке формата в данном примере используются стандартные краткие представления даты (%x) и времени (%X). На сервере, расположеннном в США, функция `strftime('%x %X', time())` возвращает значение "03/17/2006 03:55:25", что означает "17 марта 2006 года, 3 часа 55 минут и 25 секунд". Если компьютер расположен в другой части мира, представление даты будет другим (вероятнее всего, 17/03/2006). Более подробная информация о форматах даты и времени содержится в документации по использованию функции `strftime()`, которую можно найти по адресу <http://www.php.net strftime>. Для вывода времени с точностью до миллисекунд нужно использовать функцию `microtime()`.

После создания текстового представления даты и времени в функции `logMessage()` необходимо отформатировать это сообщение. Поскольку файл журнала предназначен для машинной обработки, поля в каждой строке разделяются символом табуляции \t. Записи в файле отделяются друг от друга символом новой строки \n. Для удобства машинной обработки эти символы нужно удалить, заменив символ табуляции четырьмя пробелами (именно столько символов в большинстве текстовых редакторов по умолчанию заменяет символ табуляции), а символ новой строки — пробелом. Эти же подстановки нужно выполнить и для параметра `$module`, если он задан.

Здесь же выполняется преобразование числовой константы уровня регистрации в строку. Поскольку PHP оперирует реальными числовыми значениями констант, нет простого способа преобразовать имя константы в строковое представление. Это строковое представление жестко задается в методе `levelToString()`. Оператор `switch` "просматривает" константы и возвращает имя нужной. Хотя оператор `break` в данном случае не является технически необходимым, он добавляется для ясности. Объявление этого метода статическим тоже не является обязательным. Но поскольку в нем не используются переменные-члены, а значение уровня регистрации может понадобиться за пределами класса, объявление метода статическим облегчает написание операторов вида `echo Logger::levelToString($cfg["LOGGER_LEVEL"]);`.

Следующий пример кода демонстрирует использование класса `Logger` в реальном приложении.

```
<?php
require_once('class.Logger.php');

$cfg['LOGGER_FILE'] = '/var/log/myapplication.log';
$cfg['LOGGER_LEVEL'] = LOGGER_INFO;
$log = Logger::getInstance();
```

```

if(isset($_GET['id'])) {
    //данные не записываются в журнал - уровень
    //регистрации слишком высок
    $log->logMessage('Значение id', LOGGER_DEBUG);

    //LOG_INFO – значение по умолчанию, поэтому данные выводятся
    $log->logMessage('Значение id равно' . $_GET['id']);

} else {

    //Эти данные тоже выводятся, включая имя модуля
    $log->logMessage('Значение id не передается из ' . $_SERVER['HTTP_REFERER'],
        LOGGER_CRITICAL,
        "Пробный модуль");

    throw new Exception('Не задан идентификатор id!');
}
?>

```

Если значение `$_GET['id']` задано, метод `logMessage()` вызывается дважды, но в файл журнала записывается только одно сообщение. Это связано со значением уровня регистрации, заданным в файле конфигурации. Содержимое журнала будет выглядеть примерно так.

03/17/06 03:58:42 LOGGER_INFO Значение id равно 25

Здесь выводится время и дата, уровень регистрации сообщения и само сообщение. Если значение `$_GET['id']` не задано, в файле журнала появится следующая запись.

03/17/06 03:58:42 LOGGER_CRITICAL Значение id не передается из
<http://localhost/testLogger.php> Пробный модуль

В этом случае страница `testLogger()` не может передать значение `id`, поэтому в файл записывается ошибка `LOGGER_CRITICAL`.

Расширение класса Logger

Запись в текстовый файл — чрезвычайно удобный способ ведения журнала событий. Для анализа и синтаксического разбора таких файлов существует множество утилит, начиная от простых программ `sed` и `grep` для UNIX и заканчивая развитым коммерческим программным обеспечением, специально предназначенным для анализа журналов, стоимостью в десятки тысяч долларов. Но иногда текстового файла бывает недостаточно. Для интеграции журналов приложений с системными журналами и централизации регистрации информации может понадобиться реляционная база данных. Разработанный выше класс `Logger` позволяет записывать информацию для всего приложения только в один файл. Однако иногда данные, поступающие от различных частей приложения или разных выполняемых задач, желательно записывать в разные файлы. В этом разделе класс `Logger` будет расширен с целью поддержки нескольких журналов для одного приложения и обеспечения возможности интеграции с любыми средствами хранения данных.

В главе 8 упоминался класс `PEAR DB`, позволяющий с помощью изменения единственной строки подключаться к совершенно различным базам данных. Используя строку `mysql://root@localhost/mydb`, с помощью имени пользователя `root` можно подключиться к базе данных MySQL `mydb`, работающей на локальном компьютере. Для соединения с базой данных PostgreSQL эту строку нужно заменить следующей `pgsql://postgres@localhost/yourdb`. При этом произойдет соединение с кластером

базы данных PostgreSQL на локальной машине, и пользователь, аутентифицированный как `postgres`, сможет получить доступ к базе данных `yourdb`. После установки соединения с помощью этой однотипной строки весь код использования класса DB будет одинаков, независимо от того, с какой базой установлено соединение. Поэтому для взаимодействия класса `Logger` с любым видом носителя можно использовать очень схожую конструкцию.

Целью модификации класса `Logger` является обеспечение однотипного синтаксиса при связи с различными носителями для записи журнала данных. Нужно также обеспечить возможность ведения различных журналов для разных частей приложения. Новый класс `Logger` должен поддерживать некий “реестр” соединений с журналами. Например, в один журнал можно заносить сообщения об ошибках, а в другой — информацию об SQL-запросах. Из следующего фрагмента кода видно, какие операции можно выполнять с обновленным классом.

```
<?php

Logger::register('errors', 'file:///var/log/error.log');
Logger::register('app',
'pgsql://postgres@db/errors?table=applog&timestamp=dtlog' .
'&msg=smsg&level=slevel&module=smod');

$objQLog = Logger::getInstance('queries');

$sql = "SELECT * FROM foo";
$objQLog->logMessage("Выбор всех записей");

try {
    Database->getInstance()->select($sql);
} catch (DBQueryException $e) {
    $objErrLog = Logger::getInstance('errors');
    $objErrLog->logMessage($e->getMessage(), LOGGER_CRITICAL);
}
?>
```

Поскольку обещанный класс пока не описан, пример его использования может несколько обескуражить читателя. Однако иногда четкое представление об использовании класса помогает в его проектировании. В реальном приложении первые две строки приведенного кода необходимо поместить в подключаемый файл. В этих строках задаются два различных журнала: `app` и `errors`. Сообщения журнала `app` будут храниться в таблице `applog` базы данных PostgreSQL. В этой таблице определены поля для хранения даты/времени, сообщения, уровня регистрации и модуля. Журнал ошибок будет храниться в текстовом файле, расположенным по адресу `/var/log/error.log`. Большая часть сообщений, поступающих от приложения, будет храниться в журнале `app`, предназначенному для последующего анализа работы приложения. Ошибки будут заноситься в журнал `errors`, представляющий собой текстовый файл. Такой формат журнала ошибок определяется тем, что одним из типов ошибок может быть невозможность установить соединение с базой данных.

Синтаксический разбор строки соединения

Первым шагом при создании нового класса `Logger` является анализ строки вида `scheme://user:password@host:port/path?query#fragment` (схема://пользователь:пароль@хост:порт/путь?запрос#фрагмент). К счастью, в PHP есть очень удобная функция для выполнения этой операции — `parse_url()`. В качестве параметра функция получает строку общего вида, а возвращает массив

элементов адреса URL, если таковой существует (ключами массива являются имена, используемые в приведенном выше примере запроса). Если один или несколько разделов в адресе URL не указан, соответствующий элемент массива не создается (не нужно думать, что данному ключу соответствует значение null). Например:

```
$url = "ftp://anonymous@ftp.gnu.org:21/pub/gnu/gcc"
$arParts = parse_url($url);
var_dump($arParts);

//вывод на печать
Array (
    [scheme] => ftp
    [user] =>anonymous
    [host] => ftp.gnu.org
    [port] => 21
    [path] => /pub/gnu/gcc
)
```

Обратите внимание, что в этом массиве отсутствуют элементы с ключами password, query и fragment.

Перепроектирование класса Logger для использования строки соединения

Как и для класса PEAR DB, класс Logger будет использовать элемент scheme сформированного массива для определения типа носителя для файла журнала. Для установки нового соединения будет применяться метод register(), параметрами которого являются имя журнала и адрес URL. Поскольку в приведенном выше фрагменте кода метод register() вызывается как статический метод класса, для взаимодействия методов register() и getInstance() с общим блоком информации понадобится некая промежуточная функция. В следующем примере кода приведен вновь созданный метод register(), обновленный метод getInstance() и модифицированный конструктор. По существу, это абсолютно новый класс Logger. Добавленные фрагменты выделены серым цветом.

```
<?php
```

```
//Уровни регистрации. Чем выше номер, тем меньше критичность сообщения
//Диапазоны заданы так, чтобы позднее можно было
//добавить новые уровни
define('LOGGER_DEBUG', 100);
define('LOGGER_INFO', 75);
define('LOGGER_NOTICE', 50);
define('LOGGER_WARNING', 25);
define('LOGGER_ERROR', 10);
define('LOGGER_CRITICAL', 5);
```

```
class Logger {
```

```
    private $LogFile;
    private $logLevel;
```

```
//Обратите внимание: закрытый конструктор.
//Класс создан на основе шаблона Singleton
private function __construct() {
}
```

```

public static function register($logName, $connectionString) {
    $urlData = parse_url($connectionString);

    if(! isset($urlData['scheme'])) {
        throw new Exception("Некорректная строка соединения $connectionString");
    }

    include_once('Logger/class.' . $urlData['scheme'] . 'LoggerBackend.php');

    $className = $urlData['scheme'] . 'LoggerBackend';
    if(!class_exists($className)) {
        throw new Exception('Для $urlData['scheme'] ' .
                            'нет базы данных');
    }

    $objBack = new $className($urlData);
    Logger::manageBackends($logName, $objBack);
}

public static function getInstance($name) {
    Logger::manageBackends($name);
}

private static function manageBackends($name, LoggerBackend $objBack = null) {
    static $backEnds;

    if(!isset($backEnds)) {
        $backEnds = array();
    }

    if(!isset($objBack)) {
        //
        if(isset($backEnds[$name])) {
            return $backEnds[$name];
        } else {
            throw new Exception("Заданная база данных $name" .
                                "в журнале не зарегистрирована");
        }
    } else {
        $backEnds[$name] = $objBack;
    }
}

public static function levelToString($logLevel) {
    switch ($logLevel) {
        case LOGGER_DEBUG:
            return 'LOGGER_DEBUG';
            break;
        case LOGGER_INFO:
            return 'LOGGER_INFO';
            break;
        case LOGGER_NOTICE:
            return 'LOGGER_NOTICE';
            break;
        case LOGGER_WARNING:
            return 'LOGGER_WARNING';
            break;
        case LOGGER_ERROR:
            return 'LOGGER_ERROR';
            break;
        case LOGGER_CRITICAL:
            return 'LOGGER_CRITICAL';
    }
}

```

```

        default:
            return '[unknown]';
    }
}
?>

```

В этом фрагменте кода класс `Logger` записывает информацию в файл, определяемый абстрактным классом `LoggerBackend`, который служит базовым для классов, реально записывающих данные в журнал по определенному механизму. Конструктор класса `Logger` теперь пуст, но по-прежнему объявлен в закрытой области, чтобы предотвратить возможность инстанцирования за пределами метода `getInstance()`.

Новый статический метод `register()` отвечает за создание объекта `LoggerBackend` на основе части `scheme` адреса URL, заданного в качестве второго параметра функции `register()`. Для определения схемы вызывается функция `parse_url()`. Если в адресе URL схема не задана, метод `register()` генерирует исключение. Если схема определена, предпринимается попытка подключения файла `Logger/class.[схема]LoggerBackend.php` с помощью директивы `include_once`. Содержащийся в этом файле класс инстанцируется с использованием массива `$urlData`.

Если класс `LoggerBackend` инстанцирован корректно, его экземпляр вместе с каноническим именем, определенным в первом параметре функции `register()`, передается закрытой функции `manageBackends()`. Эта функция вводится потому, что все открытые методы класса `Logger` теперь являются статическими. Следовательно, в классе `Logger` теперь нельзя использовать переменные-члены для хранения инстанцированных объектов. Функция `manageBackend()` содержит статическую переменную, играющую роль члена класса. Если функции передаются два параметра, объект `LoggerBackend` хранится в массиве `$backEnds`, а для доступа к нему используется ключ, передаваемый с помощью параметра `$name`. Если задан только один параметр (а именно, `$name`) функция `manageBackends()` возвращает объект `LoggerBackend`, который хранится в переменной `$name` (если он существует).

Обновленный метод `getInstance()` теперь возвращает не объект `Logger`, а инстанцированный объект `LoggerBackend`. Это делается с помощью вызова метода `manageBackends()` с одним параметром, определяющим каноническое имя конкретного запрашиваемого журнала. Если указанный журнал ранее не был зарегистрирован или его не удается инстанцировать в методе `register()`, функция `manageBackends()` генерирует исключение, которое “всплывает” в функции `getInstance()`.

Класс `LoggerBackend`

Методы `manageBackends()`, `register()` и `getInstance()` взаимодействуют с объектами `LoggerBackend`. В этом классе содержится конкретный конструктор (который определен и вызывается) и абстрактный метод `logMessage()`, который присутствовал еще в исходном классе `Logger`. Этот метод остался неизменным — количество и типы его параметров не отличаются от первого примера кода класса `Logger`. Однако в классе `LoggerBackend` метод `logMessage()` объявлен абстрактным — реального тела функции не существует. В этом классе приводится лишь объявление функции, которая должна быть реализована в классах-наследниках класса `LoggerBackend`. Параметром конструктора этого класса является массив, возвращаемый функцией `parse_url()`. В конструкторе этот массив записывается в закрытую переменную-член класса `LoggerBackend`. Никакие другие методы в классе `LoggerBackend` не определены. Сохраните следующий фрагмент кода в файле `Logger/class.LoggerBackend.php`.

```
<?php
abstract class LoggerBackend {
    protected $urlData;

    public function __construct($urlData) {
        $this->urlData = $urlData;
    }

    abstract function logMessage($message, $logLevel = LOGGER_INFO, $module);
}
?>
```

Сам класс необходимо объявить абстрактным, поскольку он содержит лишь один абстрактный метод, а экземпляры класса `LoggerBackend` на практике не используются. Реальную работу должны выполнять подклассы класса `LoggerBackend`.

Производные классы `LoggerBackend`

Чтобы обеспечить функционирование обновленного класса `Logger`, необходимо создать хотя бы один производный класс для класса `LoggerBackend`. Поскольку выше уже приводился пример кода для записи информации в файл, проще будет сначала создать подкласс `LoggerBackend`, выполняющий именно эту операцию.

Создайте файл `class.fileLoggerBackend.php` (обратите внимание на заглавные буквы в имени файла) и введите в него следующий код.

```
<?php
require_once('Logger/class.LoggerBackend.php');

class fileLoggerBackend extends LoggerBackend {

    private $logLevel;
    private $hLogFile;

    public function __construct($urlData) {
        global $cfg; //массив информации о конфигурации системы
                    //из внешнего файла

        parent::__construct($urlData);

        $this->logLevel = $cfg['LOGGER_LEVEL'];

        $logFilePath = $this->urlData['path'];
        if(! strlen($logFilePath)) {
            throw new Exception('Не задан путь к файлу журнала ' .
                                'в строке соединения.');
        }

        //Открываем файл журнала. Отменяем вывод сообщений об ошибках PHP.
        //Обработка ошибок будет выполняться с помощью генерации исключений.
        $this->hLogFile = @fopen($logFilePath, 'a+');
        if(! is_resource($this->hLogFile)) {
            throw new Exception("Указанный файл журнала $logFilePath " .
                                'нельзя открыть или создать для' .
                                'записи. Проверьте разрешения.');
        }
    }

    public function logMessage($msg, $logLevel = LOGGER_INFO, $module = null) {
        if($logLevel <= $this->logLevel) {
            $time = strftime('%x %X', time());
            $msg = str_replace("\t", ' ', $msg);
            fwrite($this->hLogFile, $time . ' ' . $logLevel . ' ' . $module . ' ' . $msg);
        }
    }
}
```

```

$msg = str_replace("\n", ' ', $msg);
$strLogLevel = Logger::levelToString($logLevel);
if(isset($module)) {
    $module = str_replace("\t", ' ', $module);
    $module = str_replace("\n", ' ', $module);
}
//logs: дата, время, уровень сообщений и имя модуля
//разделяются символом табуляции, записи
//отделяются друг от друга символом разрыва строки
$logLine = "$time\t$strLogLevel\t$msg\t$module\n";
fwrite($this->hLogFile, $logLine);
}
}
?

```

Большая часть этого кода уже знакома читателю. Этот фрагмент очень напоминает исходный класс `Logger`. Класс `fileLoggerBackend` будет отвечать за журнал, зарегистрированный со схемой `file://`. Он записывает журнал в файл, указанный в элементе `path` массива, возвращаемого функцией `parse_url()`. Чтобы зарегистрировать журнал этого типа, адрес URL должен иметь примерно следующий вид `file:///var/log/app.log`. Обратите внимание, что этот адрес URL состоит из следующих частей: `file://` (завершается двумя косыми) и `/var/log/app.log` (начинается одной косой), таким образом в схеме `file` используется три косые черты подряд.

Чтобы использовать журнал этого типа, введите следующий код (напоминающий пример использования журнала, приведенный выше).

```
<?php
Logger::register('app', 'file:///var/log/applog.log');
$log = Logger::getInstance('app');
$log = logMessage('Это новое сообщение для журнала!', LOGGER_CRITICAL, 'test');
?>
```

В этом фрагменте журнал типа `fileLoggerBackend` регистрируется с помощью класса `Logger` под именем `app`. Запись будет вестись в файл `/var/log/applog.log`.

Запись журнала в таблицу базы данных

Аналогичный процесс наследования класса `LoggerBackend` можно использовать для создания механизма записи в любое хранилище. В следующем примере в качестве носителя журнала используется база данных PostgreSQL. Однако эти же принципы можно применять для работы с любой платформой базы данных.

В этом классе конструктор существенно объемнее. В строке соединения должны быть указаны параметры соединения и имена полей базы данных, в которые будет заноситься информация. Поэтому в коде перед установкой соединения проверяется наличие необходимых значений. Весь этот код не представляет большого интереса, но авторы включили его в книгу, чтобы показать, как проводится синтаксический анализ строки соединения для обеспечения гибкого механизма регистрации событий в базе данных PostgreSQL. Этот код необходимо сохранить в файле `Logger/classpgsqlLoggerBackend.php`.

```
<?php
require_once('Logger/class.LoggerBackend.php');

class pgsqlLoggerBackend extends LoggerBackend {

```

```

private $logLevel;
private $hConn;

private $table = 'logdata';
private $messageField = 'message';
private $logLevelField = 'loglevel';
private $timestampField = 'logdate';
private $moduleField = 'module';

public function __construct($urlData) {
    global $cfg; //массив данных о конфигурации системы
    //из внешнего файла

    parent::__construct($urlData);

    $this->logLevel = $cfg['LOGGER_LEVEL'];

    $host = $urlData['host'];
    $port = $urlData['port'];
    $user = $urlData['user'];
    $password = $urlData['password'];
    $arPath = explode('/', $urlData['path']);
    $database = $arPath[1];

    if(!strlen($database)) {
        throw new Exception('pgsqlLoggerBackend: Некорректная строка соединения'.
            ' Не задано имя базы данных');
    }

    $connStr = '';
    if($host) {
        $connStr .= "host=$host ";
    }

    if($port) {
        $connStr .= "port=$port ";
    }

    if($user) {
        $connStr .= "user=$user ";
    }

    if($password) {
        $connStr .= "password=$password ";
    }

    $connStr .= "dbname=$database";

    //Ошибки будут обрабатываться с помощью исключений
    $this->hConn = pg_connect($connStr);

    if(! is_resource($this->hConn)) {
        throw new Exception("Нельзя соединиться с базой данных с помощью $connStr");
    }

    //Возьмите строку запроса var=foo&bar=blah
    //и преобразуйте ее в массив вида
    //array('var' => 'foo', 'bar' => 'blah')
    //Не забудьте конвертировать элементы url
    $queryData = $urlData['query'];
    if(strlen($queryData)) {
        $arTmpQuery = explode('&', $queryData);

        $arQuery = array();
        foreach($arTmpQuery as $queryItem) {
            $arQueryItem = explode('=', $queryItem);
            $arQuery[urldecode($arQueryItem[0])] = urldecode($arQueryItem[1]);
        }
    }
}

```

```

//Ни один из этих элементов не является обязательным.
//Значения по умолчанию устанавливаются в закрытой
//области объявлений в верхней части класса.
//Эти переменные задают имена таблицы и полей,
//в которых хранятся различные элементы записи журнала.
if(isset($arQuery['table'])) {
    $this->table = $arQuery['table'];
}

if(isset($arQuery['messageField'])) {
    $this->messageField = $arQuery['messageField'];
}

if(isset($arQuery['logLevelField'])) {
    $this->logLevelField = $arQuery['logLevelField'];
}

if(isset($arQuery['timestampField'])) {
    $this->timestampField = $arQuery['timestampField'];
}

if(isset($arQuery['moduleField'])) {
    $this->logLevelField = $arQuery['moduleField'];
}

}

public function logMessage($msg, $logLevel = LOGGER_INFO, $module = null) {
    if($logLevel <= $this->logLevel) {
        $time = strftime('%x %X', time());
        $strLogLevel = Logger::levelToString($logLevel);
        $msg = pg_escape_string($msg);

        if(isset($module)) {
            $module = '"' . pg_escape_string($module) . '"';
        } else {
            $module = 'NULL';
        }

        $arFields = array();
        $arFields[$this->messageField] = '"' . $msg . '"';
        $arFields[$this->logLevelField] = $logLevel;
        $arFields[$this->timestampField] = '"' . strftime('%x %X', time()) . '"';
        $arFields[$this->moduleField] = $module;

        $sql = 'INSERT INTO ' . $this->table;
        $sql .= ' (' . join(', ', array_keys($arFields)) . ')';
        $sql .= ' VALUES (' . join(', ', array_values($arFields)) . ')';

        pg_exec($this->hConn, $sql);
    }
}
?>

```

В конструкторе класс анализирует запрос, содержащийся в адресе URL, и определяет имена таблицы и полей, в которых хранится информация для журнала. Предполагается, что таблица будет создана с помощью оператора SQL, пример которого показан ниже. Имена полей не имеют значения, поскольку они настраиваются в строке соединения, но типы данных в таблице и запросе должны совпадать (или быть совместимыми).

```
create table logdata (
    message text,
    loglevel smallint,
    logdate timestamp,
    module varchar(255)
);
```

Код использования нового журнала, работающего с базой данных PostgreSQL, должен иметь примерно следующий вид.

```
<?php

$cfg['LOGGER_LEVEL'] = LOGGER_INFO;

Logger::register('app', 'pgsql://steve@localhost/mydb?table=logdata');

$log = Logger::getInstance('app');

if(isset($_GET['fooid'])) {

    //не записывается в журнал - уровень регистрации слишком высок
    $log->logMessage('Идентификатор fooid ', LOGGER_DEBUG);

    //По умолчанию используется значение LOG_INFO, поэтому сообщение записывается в журнал
    $log->logMessage('Значение fooid равно ' . $_GET['fooid']);

} else {

    //Это сообщение записывается с указанием имени модуля
    $log->logMessage('Идентификатор fooid не передается из ' .
$_SERVER['HTTP_REFERER'],
                    LOGGER_CRITICAL,
                    "Тестовый модуль");

    throw new Exception('Не найден идентификатор!');
}
?>
```

В строке соединения, передаваемой методу `Logger::register()`, задается имя журнала `app`, подключаемого к базе данных PostgreSQL, работающей на компьютере `localhost`. Объект `LoggerBackend` подключается как пользователь `steve` к базе данных `mydb`. Данные журнала записываются в таблицу `logdata`. Если эти параметры не указаны, по умолчанию используются переменные, объявленные в закрытой области класса. Этую базу данных можно легко заменить любой другой, для которой существует соответствующий модуль PHP, например MySQL или SQL Server.

Создание механизма отладки

Хотя разработанный выше механизм регистрации данных обладает практически неограниченной гибкостью, иногда гораздо более удобно выводить отладочные сообщения в окно браузера, а не в файл журнала. В классе `Logger` мы постарались избежать многочисленных операторов `print`, которые обычно приходится вручную удалять перед развертыванием приложения. Специальный отладочный механизм позволяет выводить сообщения на экран таким образом, что их легко отключить при необходимости.

В классе `Debugger` отладочные сообщения хранятся в рамках сеанса и выводятся в нижней части экрана с запрашиваемой страницей. При этом отладочные данные концентрируются в одном месте на экране, что упрощает их поиск. Такой подход

также позволяет выводить большие объемы данных, не влияя на структуру интерфейса пользователя.

Отладочные данные хранятся в рамках сеанса, поскольку они генерируются не при каждом запросе на страницу.

Класс Debugger содержит две основные функции. Функция debug() записывает код в отладочный массив в рамках сеанса с дополнительным ключом и уровнем отладки. Уровни отладки аналогичны уровням регистрации в классе Logger. Чем ниже значение уровня регистрации, тем выше его приоритет. Вторая функция называется debugPrint(). Она выводит информацию в нижней части страницы. В ней выбирается массив из сеанса и генерируется HTML-таблица, содержащая данные этого массива. После завершения вывода информация удаляется из данных сеанса. Благодаря этому сохраняется отладочная информация для запросов, не выводящих данные на экран.

Код этого класса необходимо сохранить в файле class.Debugger.php.

```
<?php

define('DEBUG_INFO', 100);
define('DEBUG_SQL', 75);
define('DEBUG_WARNING', 50);
define('DEBUG_ERROR', 25);
define('DEBUG_CRITICAL', 10);

class Debugger {

    public static function debug($data, $key = null, $debugLevel = DEBUG_INFO) {
        global $cfg;

        if(! isset($_SESSION['debugData'])) {
            $_SESSION['debugData'] = array();
        }

        if($debugLevel <= $cfg['DEBUG_LEVEL']) {
            $_SESSION['debugData'][$key] = $data;
        }
    }

    public static function debugPrint() {
        $arDebugData = $_SESSION['debugData'];
        print Debugger::printArray($arDebugData);

        $_SESSION['debugData'] = array();
    }

    function printArray($var, $title = true) {
        $string = '<table border="1">';
        if ($title) {
            $string .= "<tr><td><b>Ключ</b></td><td><b>Значение</b></td></tr>\n";
        }

        if (is_array($var)) {
            foreach($var as $key => $value) {

                $string .= "<tr>\n" ;
                $string .= "<td><b>$key</b></td><td>";

                if (is_array($value)) {
                    $string .= Debugger::printArray($value, false);
                } elseif(gettype($value) == 'object') {

```

```

        $string .= "Объект класса " . get_class($value);
    } else {
        $string .= "$value" ;
    }

    $string .= "</td></tr>\n";
}

$string .= "</table>\n";
return $string;
}
?>

```

Пять констант, определенных в верхней части файла, задают уровни отладки, аналогичные уровням регистрации в классе `class.Logger.php`. Метод `debug()` считывает массив `$cfg` в глобальном пространстве имен и выбирает элемент `DEBUG_LEVEL`, определяющий текущий уровень отладки для всего приложения.

По умолчанию уровень отладки принимает значение `DEBUG_INFO`, хотя его можно изменить с помощью третьего параметра функции `debug()`. Таким образом можно регулировать детальность отладочной информации.

Параметр `$key` — это дополнительная метка отладочного массива. Хотя в списке параметров он значится вторым, его значение выводится в самом левом столбце таблицы, формируемой функцией `debugPrint()`.

Функция `debugPrint()` просто передает отладочную информацию из данных сеанса методу `printArray()`, который выполняет всю реальную работу. После вывода отладочной информации функция `debugPrint()` очищает массив сеансов для новой отладочной информации, генерируемой следующим запросом на страницу.

Функция `printArray()` принимает массив (как минимум одномерный) и выводит таблицу HTML, отображающую содержимое массива. Многомерные массивы обрабатываются корректно. Для объектов выводится только имя класса.

Следующий фрагмент кода иллюстрирует использование этого класса.

```

<?php

require_once('class.Debugger.php');

$cfg = array();
$cfg['DEBUG_LEVEL'] = DEBUG_INFO;

$myData = array();
$myData[] = 'Привет';
$myData[] = array('name' => 'Боб',
                 'colors' => array('красный', 'зеленый', 'синий'));

Debugger::debug($myData, 'my data');

$x = 5 + 8;
Debugger::debug($x, 'x');

Debugger::debugPrint();

?>

```

Этот пример легче понять с помощью рис. 11.1, на котором показаны результаты вывода информации на экран с помощью функций `debugPrint()`.

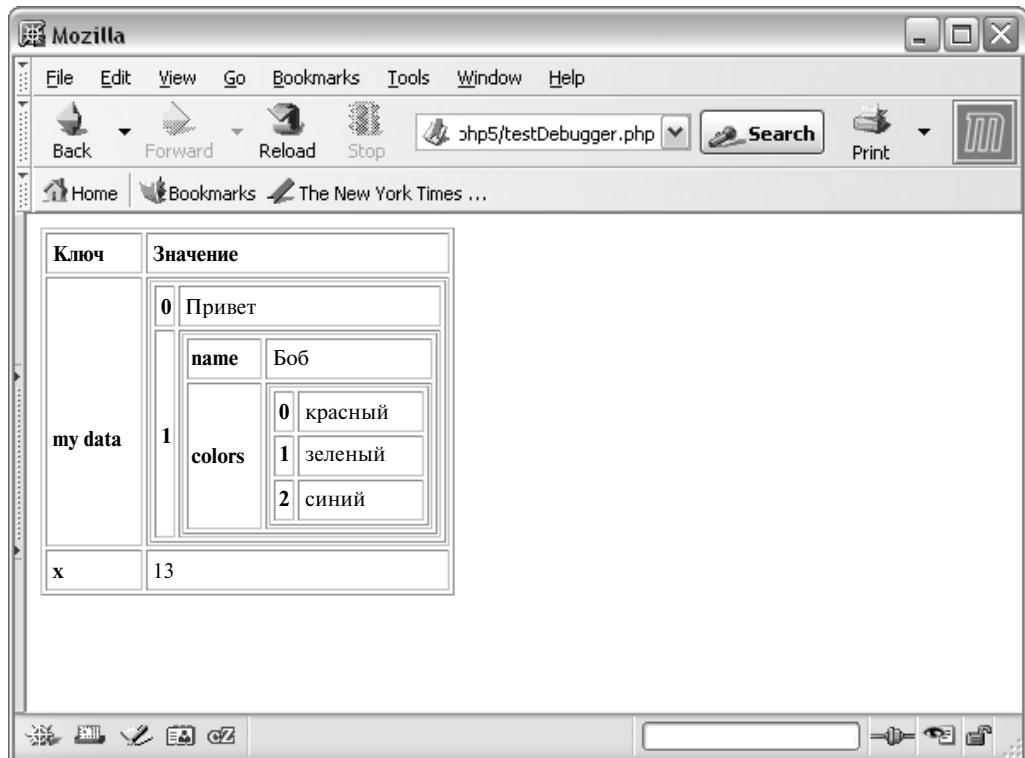


Рис. 11.1.

Резюме

Класс `Logger` можно расширить, обеспечив поддержку вывода журнальных данных в любое хранилище. Для этого нужно лишь создать и использовать нужный класс журнала. Для большинства приложений удобно пользоваться классом `fileLoggerBackend`, обеспечивающим запись в файл, но иногда полезно записывать информацию журнала в базу данных.

Класс `Debugger` функционально напоминает класс `Logger`, но позволяет выводить информацию на экран и хранить ее в данных сеанса, даже если исходный запрос не предполагает вывода данных.

В следующей главе читатель познакомится с протоколом SOAP (Simple Object Access Protocol) и узнает о возможностях PHP 5 по созданию клиентов и серверов, поддерживающих этот протокол.

12

Протокол SOAP

В настоящее время становится все более очевидным, что приложения больше не могут функционировать в вакууме, а требуют взаимодействия с другими системами. На суперкомпьютере вы не найдете ни одного фрагмента кода, который позволял бы решать все задачи любого пользователя. Такие принципы, как "слабая связанность" и "повторное использование кода" сейчас стали очень популярными, поскольку он позволяет повысить эффективность применения результатов работы других разработчиков или, как минимум, отделить и изолировать друг от друга различные функции приложения, тем самым упростив их модификацию или поддержку.

Повторное использование кода подразумевает, что каждый конкретный программный модуль должен разрабатываться лишь один раз. Если этот модуль потребовалось включить в другой проект, об этом можно просто попросить его разработчиков. Эффект от использования слабой связанности и повторного использования кода можно легко оценить, проанализировав множество Web-служб, которые без особого труда можно найти в Интернет.

Конечно, к повторно используемой части кода относятся и разработанные ранее системы, с которыми нужно взаимодействовать. Следует помнить о том, что для обеспечения удобного взаимодействия между старыми и новыми системами также потребуется выполнить много работы.

В результате такой распределенности функций и слабо связанной архитектуры на первый план перемещается вопрос реализации взаимодействия. Какой бы тип связи ни потребовался при реализации современной информационной системы, можно с уверенностью утверждать, что промежуточным транспортным звеном наверняка станет язык XML. Сейчас это еще не совсем так, однако информационные технологии развиваются именно в этом направлении.

Язык XML — это огромная тема для обсуждения, поэтому в данной книге он не будет подробно рассматриваться. Авторы надеются, что читатель немного знаком с технологией XML и сможет разобраться в материале этой главы. Если же вы не знакомы с языком XML, но хотите заняться его изучением, можно обратиться к книге *Beginning XML, 3rd Edition* издательства Wiley. Авторы книги являются приверженцами использования XML в качестве механизма взаимодействия между различным приложениями.

Поэтому в данной главе будет рассмотрен вопрос о том, как протокол SOAP (Simple Object Access Protocol – простой протокол доступа к объектам) можно применять для реализации взаимодействия на языке PHP 5, основанного на XML.

SOAP и PHP 5

SOAP – это протокол передачи сообщений, который не зависит от используемой платформы и удачно применяется совместно с различными Интернет-протоколами, такими как HTTP, SMTP и даже MIME. Как уже упоминалось выше, протокол SOAP обеспечивает возможность использования языка XML в процессе информационного обмена. Следует также отметить, что язык WSDL (Web Service Description Language – язык описания Web-служб) также существенно упрощает нашу жизнь, позволяя описать, как можно использовать службы, доступ к которым можно получить с помощью протокола SOAP. Более подробно язык WSDL будет обсуждаться в данной главе ниже.

Однако что же конкретно позволяет делать протокол SOAP? Самый простой ответ на этот вопрос заключается в том, что он позволяет применять структурированные и хорошо типизированные данные в децентрализованной и распределенной среде. Возможно, только что сформулированный ответ требует дополнительного пояснения. С помощью протокола SOAP можно передать информацию с одной платформы на другую и таким образом эффективно обходить ограничения, налагаемые на взаимодействие между различными компонентами в гетерогенной информационной среде. И это является чрезвычайно важным.

Язык PHP 5 позволяет использовать несколько различных библиотек поддержки протокола SOAP. Эти библиотеки перечислены ниже.

- ❑ SOAP-расширение PHP 5 : входит в комплект поставки PHP 5 .
- ❑ PEAR::SOAP: <http://pear.php.net/package/SOAP>.
- ❑ ezSOAP: <http://ez.no>.
- ❑ NuSOAP: <http://dietrich.ganx4.com/nusoap>.

Для работы можно воспользоваться любой из перечисленных библиотек. Однако в данной главе будет рассматриваться расширение SOAP языка PHP 5 , которое входит в комплект поставки.

Расширение SOAP PHP 5

В момент написания данной книги SOAP-расширение нужно было подключать вручную. Для этого в конфигурационном файле `php.ini` нужно было добавить строку `extension=php_soap.dll` для системы Windows или выполнить компиляцию с ключом `--enable-soap` в Linux. Будьте внимательны и следите за документацией, поскольку в последующих версиях эти рекомендации могут измениться.

Существует много аспектов, на которые нужно обращать внимание при настройке расширения SOAP в PHP 5 . Один из наибольших недостатков протокола SOAP заключается в том, что при его использовании не стоит рассчитывать на высокую скорость соединения, поскольку SOAP-клиент сначала генерирует запрос на получение документа WSDL и определяет, как необходимо вызвать функцию требуемой службы. Вполне очевидно, что с точки зрения времени выполнения какой-либо задачи загрузка файла через медленное соединение или, что еще важнее, подключение с большой задержкой может оказаться самым узким местом.

Решение этой проблемы заключается в специальной настройке модуля PHP. С помощью соответствующих конфигурационных параметров можно указать, нужно ли кешировать WSDL-страницы, местоположение их хранения, а также время использования этого хранилища. Это означает, что время выполнения приложения будет достаточно длительным только в первый раз. После загрузки требуемых документов будут использоваться их кешированные версии.

Если вы решили воспользоваться приведенными выше рекомендациями и повысить производительность приложения, добавьте в файл `php.ini` следующие строки.

```
;SOAP

;Параметр, который определяет, будет ли выполняться кеширование
soap.wsdl_cache_enabled = "1"

;Каталог хранения кешированных файлов
soap.wsdl_cache_dir="usr/local/php/wsdlcache"

;Время использования кешированной копии в секундах
soap.wsdl_cache_ttl = "1000"
```

Функции PHP 5 для поддержки протокола SOAP можно разделить на три категории. К двум основным относятся функции `SoapClient` и `SoapServer`, а к третьей — различные управляющие функции. Кратко эти функции будут рассматриваться ниже, однако более подробную информацию можно найти по адресу <http://www.php.net/manual/en/ref.soap.php>.

Функции `SoapClient`

Как принято при использовании объектно-ориентированного подхода, доступ к SOAP-функциям можно получить через соответствующие SOAP-объекты. Объект `SoapClient` инстанцируется следующим образом.

```
$client = new SoapClient (mixed wsdl [,array параметры]);
```

Первый параметр может принимать либо значение `null` (если описание WSDL не используется), либо определять идентификатор URI (универсальный идентификатор ресурса) документа WSDL. Остальные параметры передаются в виде массива и могут использоваться для определения свойств клиента SOAP. Например, клиент SOAP, использующий зашифрованные сообщения SOAP и стиль RPC, можно записать следующим образом.

```
$client = new SoapClient (null,
    array ('location' => "http://localhost/my_soap.php",
        'uri' => "http://my_url/",
        'style' => SOAP_PRC, 'use' => SOAP_ENCODED));
```

После инстанцирования клиента SOAP можно воспользоваться различными его методами. Эти методы приведены ниже.

`_call`. Позволяет обращаться к протоколу SOAP напрямую. В большинстве случаев функции SOAP можно вызывать как методы объекта `SoapClient`. Однако данный метод может пригодиться для передачи, например, заголовков SOAP.

- ❑ `_getFunctions`. Возвращает функции, предоставляемые данной Web-службой.
- ❑ `_getLastRequest`. Как следует из имени метода, при необходимости можно извлечь предыдущий запрос SOAP-клиента. Однако это можно осуществить

лишь в том случае, если экземпляр SoapClient был создан со значением `true` параметра `trace`.

- ❑ `__getLastResponse`. Позволяет получить предыдущий ответ, полученный клиентом SOAP. Для вызова этого метода нужно, чтобы был включен режим `trace`.
- ❑ `__getTypes`. Возвращает список типов SOAP (т.е. структур и объектов, определенных в Web-службе). Этот метод можно использовать в том случае, когда объект SoapClient был инстанцирован в режиме WSDL (т.е. если в качестве первого параметра был задан идентификатор URI).

Вопросы создания и использования клиентов SOAP будут рассматриваться чуть ниже, а сейчас имеет смысл познакомиться с другими возможностями PHP 5, связанными с поддержкой протокола SOAP.

Функции SoapServer

Конечно, обеспечить получение информации от клиента SOAP — это не единственная задача разработчиков. При реализации Web-службы одной из наиболее важных задач является создание SOAP-сервера, и язык PHP 5 предоставляет для этого ряд возможностей.

Как и при создании объекта SoapClient, для создания объекта SoapServer используется конструктор.

```
$server = new SoapServer (mixed wsdl [, array параметры])
```

С помощью первого параметра можно задать требуемый WSDL-документ или указать значение `null`. Если первым параметром конструктора является значение `null`, то в массиве параметров нужно указать параметр `uri` точно так же, как и при создании объекта SoapClient. После создания объекта-сервера можно использовать следующие его методы.

- ❑ `addFunction`. Позволяет добавить функцию для обработки SOAP-запросов.
- ❑ `getFunction`. Возвращает список всех доступных функций сервера.
- ❑ `handle`. Извлекает значения из параметра `soap_request` или, при его отсутствии, из глобальной переменной `$HTTP_RAW_POST_DATA`; обрабатывает данный запрос и возвращает ответ клиенту.
- ❑ `setClass`. Позволяет добавить к серверу методы PHP (и целые классы), которые будут использоваться для обработки запросов SOAP.
- ❑ `setPersistence`. Если для расширения функциональности сервера была использована функция `setClass`, с помощью этого метода можно обеспечить сохранение данных между запросами в пределах одного сеанса.

Другие функции

Несколько других функций SOAP предназначены для выполнения различных действий, наиболее важным из которых является обработка исключений. Можно создать объект SoapFault, который обеспечивал бы возврат с сервера сообщений об ошибках.

```
$fault = new SoapFault(string код-ошибки, string
строка- ошибки [,string
объект- ошибки [, mixed подробности [, string имя- ошибки [,mixed
заголовок]]]]);
```

Если при создании объекта SoapClient параметр exceptions имеет значение 0, то запрашиваемый метод будет возвращать объект SoapFault при возникновении любой ошибки. С учетом этого можно воспользоваться следующей функцией (не относящейся ни к какому классу) и проверить, успешным ли был вызов функции.

```
is_soap_fault ($methodresult)
```

Эта функция возвращает значение true, если переданный ей параметр является экземпляром класса SoapFault.

Кроме средств обработки исключений, в расширении SOAP PHP 5 реализован также объект SoapVar, который можно использовать для задания свойств типа, если режим WSDL не используется. Этот объект инстанцируется следующим образом.

```
$type = new SoapVar (mixed данные, int шифрование [, string имя_типа
[, string пространство-имен-типа [,string имя_узла
[, string пространство-имен-узла]]]]);
```

Как следует из имени, объект SoapHeader позволяет передавать SOAP-заголовки между клиентом и сервером. Для этого можно воспользоваться функциями __call() или handle().

```
client->__call("myFunction", null, null,
new SoapHeader('http://thenamespace.org/mynamespace/', 'myExample', 'an
example'));
```

И наконец, для передачи пар “имя-значение” в качестве параметров SOAP от клиента, который не находится в режиме WSDL, можно использовать объект SoapParam. Например, при вызове метода SOAP-сервера параметры можно передать следующим образом.

```
$client->__call ( "getSoapMethod", array (new SoapParam($value, "name")) );
```

Выше вы кратко ознакомились с возможностями расширения SOAP языка PHP. Полученные знания можно использовать для создания клиента и сервера SOAP. Именно этим мы и займемся в этой главе ниже.

Создание SOAP-клиента

К счастью, в Интернет можно найти различные примеры бесплатных серверов, предоставляющих Web-службы. Для того чтобы воспользоваться каким-либо из них, достаточно обладать некоторой информацией и реализовать соответствующий SOAP-клиент.

Итак, начнем. Web-сервер, с которым будет интересно познакомиться, находится по адресу <http://www.xmethods.com>. Загрузите эту страницу в браузер и посмотрите на ее нижнюю часть (рис. 12.1).

Конечно, если оказалось, что примеры служб были изменены или перенесены в другое место, то в приведенный в этой главе программный код придется внести определенные изменения. Однако самое главное – это методы, которые обеспечивают взаимодействие с сервером, а не сам сервер.

Прежде чем двигаться дальше, обратите внимание на следующие условия использования примеров Web-служб. Они были скопированы с Web-узла без изменений.

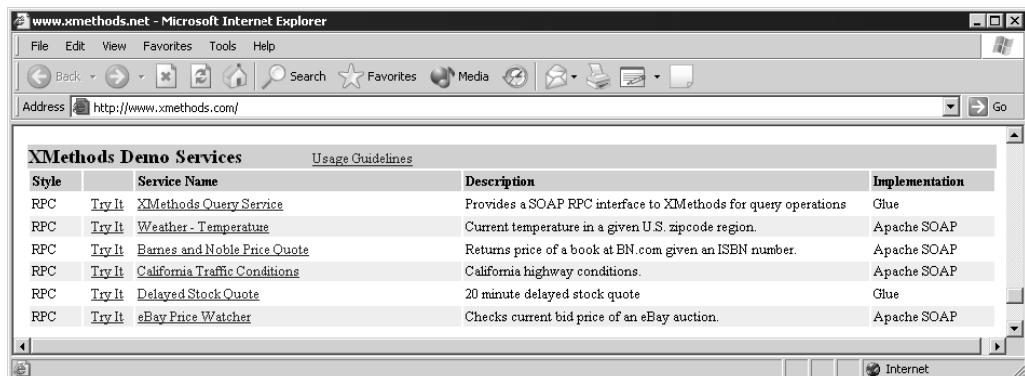


Рис. 12.1.

На Web-узле “XMethods Demo Services” содержатся примеры Web-служб, которые призваны помочь разработчикам. Эти службы полностью бесплатны. Тем не менее, обратите внимание на следующее.

1. Наша машинальные ресурсы и полоса пропускания являются ограниченными. Пожалуйста, не злоупотребляйте службами. Любое их некорректное использование может привести к блокировке вашего IP-адреса без каких бы то ни было предупреждений.
2. Любая из доступных служб может быть отключена без предупреждения.
3. Делается все возможное для обеспечения работоспособности служб, однако их функционирование через некоторое время не гарантируется.
4. Мы не несем ответственности за достоверность информации, предоставляемой нашими службами. Используйте их на свой страх и риск.
5. Учитывая все вышесказанное, мы не рекомендуем использовать наши службы для решения реальных задач, например разработки порталов.

По счастливому стечению обстоятельств, разработчики ресурса XMethods реализовали базирующуюся на протоколе SOAP Web-службу, которая позволяет получить цены на книги, предоставляемые компанией Barnes & Noble. Давайте воспользуемся этой службой и получим цену данной книги. Щелкните на ссылке Barnes and Noble Price Quote главной страницы. Обратите внимание на то, что разработчики максимально упростили решение поставленной задачи, поскольку предоставили идентификатор URI (см. следующий фрагмент кода) соответствующего документа WSDL. Вспомните, что этот документ определяет способ использования службы SOAP-клиентом.

Как уже упоминалось выше, при создании объекта SoapClient нужно принять решение о том, будет ли использоваться режим WSDL. Поскольку в данном случае у нас имеется ссылка на WSDL-документ, воспользуемся ею.

```
<?php
$client = new SoapClient("http://www.xmethods.net/sd/2001/
BNQuoteService.wsdl");
?>
```

Нам очень повезло, поскольку при отсутствии требуемого документа WSDL пришлось бы вручную формировать массив параметров, который содержал бы идентификатор URI, стиль и т.д.

Если предположить, что служба является работоспособной, на данный момент работу можно считать практически выполненной. На Web-узле можно щелкнуть на ссылке **Analyze WSDL** и просмотреть описание службы. Однако вместо этого давайте воспользуемся некоторыми из встроенных в язык PHP 5 функций и определим, как же эту службу можно использовать.

Лучше всего вывести на экран результат вызова функции `__getFunctions()`.

```
<?php
$client = new SoapClient("http://www.xmethods.net/sd/2001/
BNQuoteService.wsdl");

var_dump($client->__getFunctions());
```

?>

Сохраните этот код в файле `soap_functions.php` и обратитесь к нему через браузер. В результате вы увидите примерно следующую информацию (рис. 12.2).

Как можно увидеть, служба предоставляет единственный метод `getPrice()`, который в качестве параметра получает строку с ISBN-номером необходимой книги и возвращает значение с плавающей точкой. Это все, что необходимо знать. Теперь к серверу можно обратиться с запросом на получение цены книги *PHP 5 для профессионалов* (правда, на английском языке).

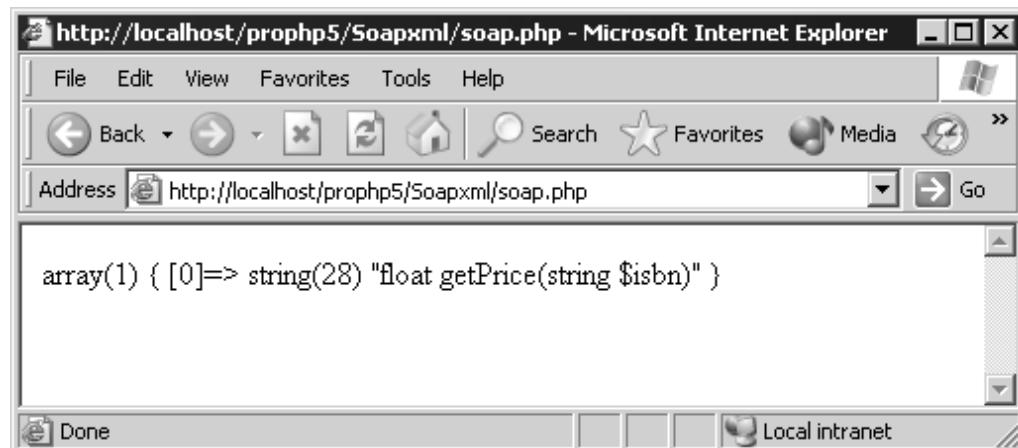


Рис. 12.2.

```
<?php
$client = new SoapClient("http://www.xmethods.net/sd/2001/
BNQuoteService.wsdl");

echo "Цена запрашиваемой книги: <BR><BR>$";
print $client->__call("getPrice", array("0764572822"));

?>
```

Сохраните этот код в файле BNQuote.php и обратитесь к нему через браузер. Если все пройдет успешно, вы получите искомое значение (рис. 12.3).

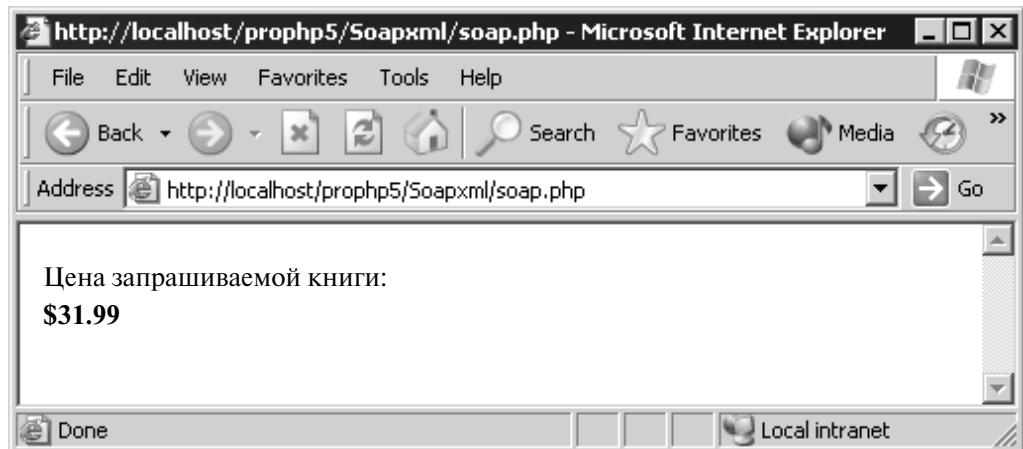


Рис. 12.3.

Конечно, показанный на рисунке результат не является реальной ценой данной книги. Это лишь демонстрация, которая никак не отражает цену компании Barnes & Nobles. Так что не расстраивайтесь, если вам пришлось заплатить за эту книгу больше!

Для того чтобы получить доступ к методу сервера, вовсе необязательно использовать метод __call(). Требуемый метод можно вызвать и напрямую, как показано ниже.

```
print $client->getPrice("0764572822");
```

Как видно из рассмотренного примера, можно без особых проблем получить информацию о любой книге Barnes & Nobles. Для этого достаточно всего нескольких строк кода. Конечно, за кулисами гораздо сложнее. Давайте разберемся, что же там происходит.

За кулисами

Несмотря на то, что программный интерфейс расширения SOAP языка PHP 5 очень прост в использовании, проверка корректности взаимодействия все же требует определенных усилий. В этом процессе документы WSDL играют очень важную роль, поскольку они определяют, как должно выполняться взаимодействие клиента и сервера. По существу документ WSDL можно рассматривать как расширяемый и основанный на XML способ определения Web-служб как набора конечных точек, управляемых сообщениями.

В этом разделе детально будет рассмотрен документ WSDL из предыдущего примера, а также сообщения с запросами и ответами SOAP, которые передаются между клиентом и сервером для получения цены книги.

Документ WSDL

Документ WSDL содержит различные элементы, которые определяют обязательные правила взаимодействия клиентов и серверов данной Web-службы. Документы WSDL хорошо структурированы и содержат некоторые основные элементы, а также несколько менее важных. Все элементы одновременно использовать необязательно. Ниже приведен перечень элементов с их кратким описанием.

- ❑ **definitions.** Корневой элемент, определяющий имя и целевое пространство имен.
- ❑ **types.** Описывает используемые типы данных и является необязательным. Если типы не определены, то по умолчанию используется спецификация XML Schema консорциума W3C.
- ❑ **message.** Описывает получаемое или отправляемое сообщение.
- ❑ **portType.** Задает или описывает операцию, которая обычно связана как с получаемым, так и с отправляемым сообщением.
- ❑ **binding/operation.** Описывает, как сообщения передаются. Содержит элементы, которые определяют способ связывания и стиль кодирования сообщения.
- ❑ **service.** Как следует из названия, этот элемент определяет адрес службы.
- ❑ **documentation.** Позволяет поставщику службы предоставить ее описание.
- ❑ **import.** Позволяет разделить WSDL-документ на различные элементы, чтобы можно было хранить в разных документах и импортировать их при необходимости.

Мы не будем тратить много времени на обсуждение языка WSDL. Более подробную информацию о нем можно найти по адресу <http://www.w3.org/TR/wsdl>.

В приведенном ниже фрагменте определяется версия XML (1.0), а затем задается элемент **definitions**. Этот элемент всегда является корневым и задает имя Web-службы (в данном случае BNQuoteService), а также целевые пространства имен, используемые в документе.

```
<?xml version = "1.0"?>
<definitions name="BNQuoteService"
  targetNamespace="http://www.xmethods.net/sd/BNQuoteService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://www.xmethods.net/sd/BNQuoteService.wsdl"
```

Далее следуют элементы **message**. В этом разделе содержатся имена запросов и ответов и определяются соответствующие типы (перечень всех типов, определенных в расширении SOAP, можно найти по адресу <http://www.php.net/manual/en/res.soap.php>).

```
<message name="getPriceRequest">
  <part name="isbn" type="xsd:string" />
</message>
<message name="getPriceResponse">
  <part name="return" type="xsd:float" />
</message>
```

В разделе **portType** полностью описывается операция **getPrice**, с которой связано отправляемое **getPriceRequest** и получаемое **getPriceResponse** сообщения.

```
<portType name="BNQuotePortType">
  <operation name="getPrice">
    <input message="tns:getPriceRequest" name="getPrice" />
    <output message="tns:getPriceResponse" name="getPriceResponse" />
  </operation>
</portType>
```

В следующем фрагменте кода определяется имя связки и ее тип BNQuotePortType. Это означает, что на самом деле связка определяется для операции portType, которая была описана выше. Далее определяется стиль RPC, а также транспортный механизм. Поскольку используется протокол HTTP, то используется транспорт SOAP HTTP. Обратите внимание, что в этом разделе также определяется, что тело сообщения SOAP будет кодироваться, хотя в общем случае возможны и другие варианты.

```
<binding name="BNQuoteBinding" type="tns:BNQuotePortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="getPrice">
    <soap:operation soapAction="" />
    <input name="getPriceRequest">
      <soap:body use="encoded" namespace="urn:xmethods-BNPriceCheck"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
    <output name="getPriceResponse">
      <soap:body use="encoded" namespace="urn:xmethods-BNPriceCheck"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </output>
  </operation>
</binding>
```

И наконец, с помощью элемента port определяется одна граничная точка с адресом реальной службы, в данном случае `http://services.xmethods.net:80/soap/servlet/rpcrouter`. Обратите внимание, что разработчики этой службы предоставили также описание службы, что упрощает ее использование. Это их характеризует с хорошей стороны.

```
<service name="BNQuoteService">
  <documentation>
    Возвращает цену книги с узла BN.com для данного номера ISBN
  </documentation>
  <port name="BNQuotePort" binding="tns:BNQuoteBinding">
    <soap:address
      location="http://services.xmethods.net:80/soap/servlet/rpcrouter" />
  </port>
</service>
</definitions>
```

Авторы надеются, что проведенный краткий анализ будет способствовать созданию вашего собственного WSDL-документа. Как уже упоминалось, много дополнительной информации о языке WSDL можно найти по адресу <http://www.w3.org/TR/wsdl>. Если вы планируете разработать свою собственную службу, то нужно начать с изучения именно этого ресурса.

Теперь давайте познакомимся с тем, как формируются запросы и ответы.

Оболочки запросов и ответов

Как вы уже знаете, язык PHP 5 предоставляет множество функций, которые можно использовать для работы с запросами и ответами. Откройте ранее созданный файл `BNQuote.php` и модифицируйте его следующим образом.

```

<?php

$client = new
SoapClient("http://www.xmethods.net/sd/2001/BNQuoteService.wsdl",
array("trace" => 1));

echo "Цена запрашиваемой книги: <BR><B>$";
print $client->__call("getPrice", array("0764572822"));

echo "</B><BR> Запрос: <B>";
print "<BR>" . ($client->__getLastRequest ()) . "</BR>";
echo "</B> Ответ: <B>";
print "<BR>" . ($client->__getLastResponse ()) . "</B></BR>";

?>

```

Обращение к обновленному файлу BNQuote.php из броузера приведет к получению следующей информации (рис. 12.4).

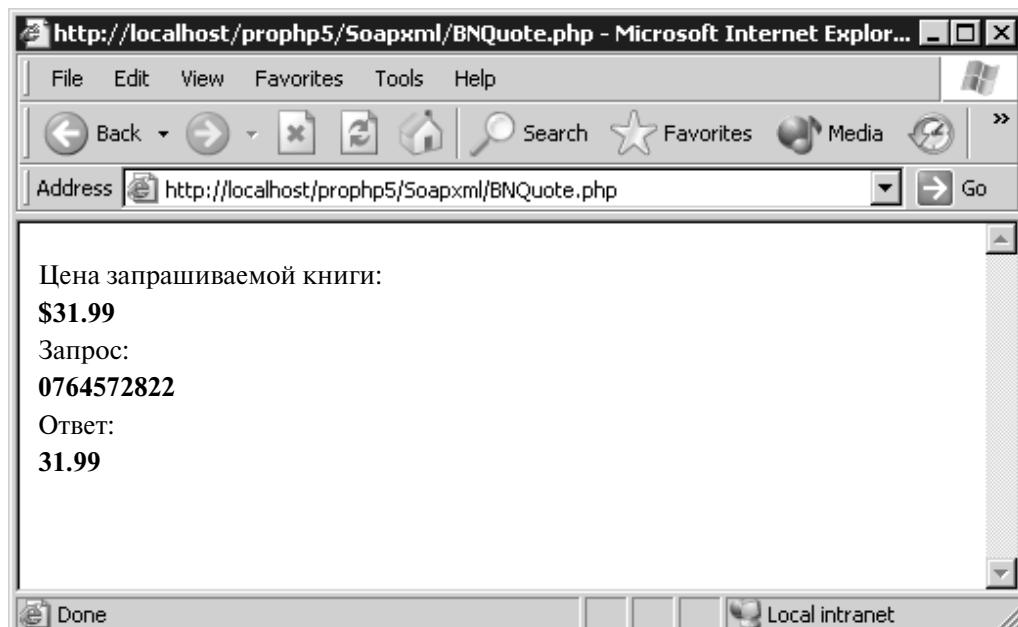


Рис. 12.4.

Никакой новой информации мы не получили. Выберите команду View Source (Просмотр HTML-кода) и посмотрите на исходный код, загруженный в браузер. Ниже приведен код оболочки запроса, по которому можно легко разобраться со способом передачи сообщения, который был определен на языке WSDL. Например, в данном случае запрос был отправлен как строка.

```

<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="urn:xmethods-BNPriceCheck"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"

```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
SOAP-ENV:encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
  <ns1:getPrice>
    <isbn xsi:type="xsd:string">0764572822</isbn>
  </ns1:getPrice>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Далее приведена оболочка ответа, возвращаемая сервером Web-службы. И в этом случае ее очень легко связать с документом WSDL. В частности, можно заметить, что возвращается значение с плавающей точкой.

```

<?xml version='1.0' encoding='UTF-8' ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  >
  <SOAP-ENV:Body>
    <ns1:getPriceResponse xmlns:ns1="urn:xmethods-BNPriceCheck"
      SOAP-ENV: encodingStyle= "http://schemas.xmlsoap.org/soap/enoding/">
      <return xsi:type="xsd:float">31.99</return>
    </ns1:getPriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

К сожалению, приведенный выше код является не совсем идеальным. Например, если в сценарии указать несуществующий ISBN-номер, вы не получите никаких полезных сообщений. В реальных приложениях абсолютно необходимо выполнять проверку на ошибки и обработку исключений.

На первый взгляд, может показаться, что способ подобной проверки лучше выбрать на уровне всего приложения. Однако давайте посмотрим, как для достижения этой цели можно применить возможности расширения SOAP языка PHP и тем самым добиться повышения надежности.

Обработка исключений в клиенте SOAP

Как уже упоминалось выше, расширение SOAP предоставляет объект SoapFault, который можно использовать для обработки исключений. Откройте созданный ранее файл soap.php и внесите изменения, показанные ниже. (Новый файл назовите soapfaults.php.)

```

<?php

$client = new
SoapClient("http://www.xmethods.net/sd/2001/BNQuoteService.wsdl",
array("exceptions" => 0));

```

```

echo "Цена запрашиваемой книги: <BR><B>$";
print $client->__call("getPrice", array("0764572822"));
?>

```

Значение 0 элемента exceptions массива параметров позволяет, чтобы объект SoapClient автоматически возвращал объект SoapFault при возникновении каких-либо ошибок. Подобные ситуации можно перехватить в блоке try...catch или с помощью метода is_soap_fault(). Этот метод позволяет определить, был ли возвращен объект SoapFault. Модифицируйте файл soapfaults.php следующим образом.

```

<?php
$client = new
SoapClient("http://www.xmethods.net/sd/2001/BNQuoteService.wsdl",
array("exceptions" => 0));
$methodresult = $client->__call("getPriceRequest", array("0764572822"));
if (is_soap_fault($methodresult)){
    echo "Возникла проблема: <BR>";
    var_dump($methodresult);
}
?>

```

Если нужно просмотреть скрытые переменные-члены объекта, вместо функции `var_dump` можно воспользоваться функцией `var_export`. Однако самое главное заключается в том, чтобы генерировать осмысленное сообщение с описанием возникшей проблемы. Например, при попытке обращения к неверному методу может быть получена следующая информация (рис. 12.5).

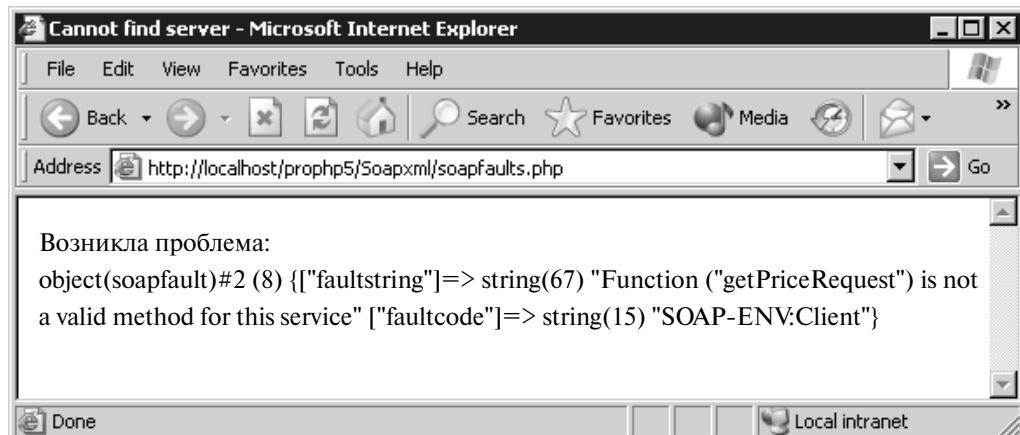


Рис. 12.5.

Конечно, существуют различные способы реализации обработки исключений, и разработчик может выбрать один из них, наиболее подходящий для конкретного приложения. В этом разделе лишь подчеркивается, что это вполне посильная задача.

Создание сервера SOAP

Зачастую нужно реализовать сервер, который предоставлял бы своим клиентам полезные услуги. Эти услуги могут быть связаны с чем угодно, начиная с поиска цен на книги и заканчивая верификацией данных на сервере имен, введенных служащими какой-либо компании. В данном разделе будет создан сервер, к которому клиент сможет подключиться и получить короткое сообщение. Кроме того, для описания взаимодействия двух частей SOAP-приложения будет использоваться небольшой WSDL-документ, описывающий взаимодействие двух частей нашего приложения. (Выше реальное использование WSDL-документов уже рассматривалось, поэтому сейчас мы не будем на этом подробно останавливаться.)

В рассматриваемом примере клиентом передается строка name, а в ответ возвращается сообщение. Создайте новый файл soapserver.php со следующим кодом.

```
<?php
function sayHello($name) {
    $salutation = "$name, вы будете рады узнать, что я работаю!";
    return $salutation;
}

$server = new SoapServer ("greetings.wsdl");
$server->addFunction("sayHello");
$server->handle();
?>
```

Этот сценарий, по существу, представляет собой простой пример вывода приветствия Здравствуй, мир!. Для разработки чего-то более полезного требуются большие усилия. Если бы понадобилось создать Web-службу, которая возвращала бы информацию из базы данных, то пришлось бы реализовать несколько функций, возвращающих результаты серверу; добавить функции к серверу, а также изменять описание WSDL для учета всех этих изменений и т.д.

В приведенном выше фрагменте кода реализована единственная функция sayHello(), которая предоставляется службой. Затем создается сервер, ссылающийся на документ greetings.wsdl (который будет определен чуть ниже), при помощи метода addFunction() добавляется функция sayHello(), а затем функция handle() выполняет всю работу по обработке запроса и ответа.

Для того чтобы обеспечить возможность использования службы, нужно сообщить ожидаемым клиентам о том, как с помощью документа WSDL можно связаться с сервером. Вот пример WSDL-файла, решающего эту задачу.

```
<?xml version ='1.0' encoding ='UTF-8' ?>
<definitions name='greetings'
  targetNamespace='http://myserver.co.za/sayHello'
  xmlns:tns='http://myserver.co.za/sayHello'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
  xmlns='http://schemas.xmlsoap.org/wsdl/'>

  <message name='sayHelloRequest'>
    <part name='name' type='xsd:string'/>
  </message>
  <message name='sayHelloResponse">
    <part name='salutation' type='xsd:string' />
  </message>

  <portType name='sayHelloPortType'>
    <operation name='sayHello'>
      <input message='tns:sayHelloRequest' />
      <output message='tns:sayHelloResponse' />
    </operation>
  </portType>

  <binding name='sayHelloBinding' type='tns:sayHelloPortType'>
    <soap:binding style='rpc'
      transport='http://schemas.xmlsoap.org/soap/http' />
    <operation name='sayHello'>
      <soap:operation soapAction=''/>
      <input>
```

```

<soap:body use='encoded' namespace=''
    encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
</input>
<output>
    <soap:body use='encoded' namespace=''
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
    </output>
</operation>
</binding>
<documentation>Это пример SOAP-сервера</documentation>
<service name='sayHelloService'>
    <port name='sayHelloPort' binding='sayHelloBinding'>
        <soap:address
location='http://localhost/prophp5/Soapxml/soapserver.php' />
    </port>
</service>
</definitions>

```

В этом фрагменте содержится все, что нужно. Однако не забывайте о том, что для корректной работы требуется задать корректный идентификатор URI.

Осталось воспользоваться новой службой. Это позволяет сделать следующий файл soapclient.php.

```

<?php
    $client = new SoapClient("greetings.wsdl");
    print_r($client->sayHello("Дэвид"));
?>

```

Обратившись к этому файлу в браузере, вы получите примерно следующий результат (в зависимости от введенного имени) (рис. 12.6).

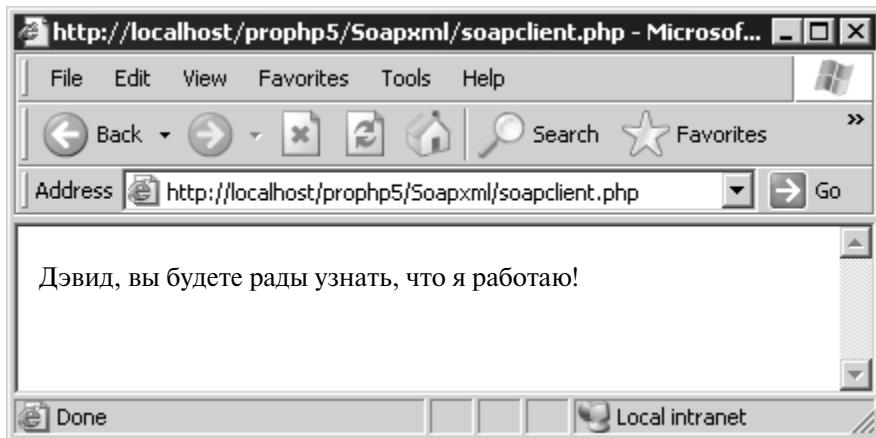


Рис. 12.6.

Резюме

В качестве отправной точки данную главу могут использовать те разработчики, которые хотят приступить к разработке мощных приложений на базе протокола SOAP. Этот протокол, а также связанные с ним технологии, составляют огромный пласт

знаний в области информационных технологий, поэтому рассказать обо всем в одной главе просто не представляется возможным. Однако не забывайте, что информация о том, как создавать и использовать Web-службы, может оказаться мощным орудием в арсенале разработчиков на языке PHP, особенно с учетом все возрастающего интереса к приложениям с распределенной архитектурой.

Из этой главы вы узнали, как язык PHP поддерживает протокол SOAP и какие программные средства позволяют их совместно использовать. При этом основное внимание было уделено новому расширению SOAP PHP 5 и его возможностям. Вся мощь Web-служб была продемонстрирована на простом примере получения цены книги по ее ISBN-номеру. Часть представленного материала была посвящена и документам WSDL, предназначенным для описания правил взаимодействия SOAP-сервера и SOAP-клиента.

И наконец, было продемонстрировано, как можно создать простой SOAP-сервер и использовать его для обработки простого запроса.

Часть III

Разработка повторно используемого набора объектов: сложные (но не слишком) служебные классы

В ЭТОЙ ЧАСТИ...

Глава 13. Модель, вид, контроллер

Глава 14. Общение с пользователями

Глава 15. Сеансы и аутентификация

Глава 16. Каркас для модульного тестирования

Глава 17. Конечные автоматы и файлы конфигурации

13

Модель, вид, контроллер

Первое появление языка PHP было очень похоже на небольшую революцию. Интерактивные Web-приложения с возможностями, которые однажды были реализованы разработчиком-профессионалом, стали появляться повсюду. В самых необычных местах вдруг стало появляться динамичное содержимое, которое выглядело очень сложным и пришло совсем не из мира CGI-сценариев для традиционной гостевой книги.

Причиной подобной революции была чрезвычайная понятность средств PHP. Эта новая технология позволяла реализовать не только программу “Здравствуй, мир!”. Даже традиционно сложные процедуры, такие как интеграция баз данных, стали вполне реализуемыми всего лишь в нескольких строках кода.

Однако основная проблема заключается в том, что такая доступность стала быстро использоваться не по назначению. С одной стороны, это приводило к тому, что сложные приложения разрабатывались неопытными командами разработчиков. А с другой стороны, опытные профессионалы сознательно отступали от своих принципов и не могли устоять перед соблазном воспользоваться беспрецедентной простотой языка PHP.

Риск, связанный с подобными подходами к разработке, вполне очевиден. Такое “применение” языка PHP приводит к появлению приложений, которые очень неудобно поддерживать. Для таких “шедевров” хорошо документированы случаи неустойчивого и ненадежного функционирования и даже изъяны, связанные с нарушением безопасности.

Многие языки программирования, такие как Java и C++, вынуждают использовать правильные подходы к программированию в процессе выполнения проектов, однако для использования языка PHP от разработчиков требуется самодисциплина. Опытный PHP-профессионал отличается от кодера на диване одним важным качеством: самодисциплиной. Одним из наиболее значимых примеров подобной самоорганизации является методология, предлагающая использование шаблона “модель–вид–контроллер” (Model–View–Controller (MVC)).

В данной главе вы узнаете, что собой представляет модель MVC и почему она считается методологией, которую стоит применять при выполнении ваших собственных проектов. Кроме того, будет построен полезный набор инструментов, который позволяет

использовать некоторые наиболее важные принципы, положенные в основу модели MVC. После прочтения главы вы сможете использовать им в своих приложениях.

И наконец, вы познакомитесь с двумя эффективными способами реализации шаблонов и научитесь их использовать параллельно с разработанными в этой главе инструментальными средствами.

Знакомство с архитектурой MVC

Буква *M* означает “модель” (model), *V* – “вид” (view) и *C* – “контроллер” (controller). Однако какой же смысл вкладывается в эти термины?

Модель MVC лучше всего описывается с помощью хорошо известного понятия “шаблон проектирования”. *Шаблон проектирования* (design pattern) – это описание многократно используемого решения повторяющейся проблемы в определенном контексте. Шаблоны проектирования предназначены для непротиворечивого решения проблем, которые возникают при проектировании крупномасштабных приложений. Наилучшего практического результата можно достигнуть лишь при использовании архитектуры, построенной на основе использования апробированных принципов.

Основная проблема заключается в том, как отделить друг от друга пользовательское управление программой (контроллер), ее внешний вид (вид), а также внутреннюю обработку и механизм принятия решений (модель). При этом нужно обеспечить их представление в виде трех определенных, отделимых друг от друга, компонентов. Как легко догадаться, решением этой проблемы является модель MVC. Однако зачем это нужно?

Во-первых, разделение влечет за собой возможность замены. Реализовав три отдельных компонента, можно обеспечить их непротиворечивое взаимодействие друг с другом. Кроме того, при необходимости любой компонент можно заменить. Например, если понадобилось, чтобы Web-приложение работало с карманными компьютерами, а не с Web-браузерами, можно заменить компонент “вид” на другой компонент, с помощью которого можно было бы представлять содержимое в формате карманного компьютера, а не Web-браузера. Если вы захотите обеспечить голосовой ввод данных, а не заставлять пользователей щелкать мышкой, можно заменить компонент “контроллер” на другой компонент, основанный на средствах VoiceXML.

Во-вторых, поиск трудноуловимых ошибок в коде, а также поддержка программы после ее передачи заказчику существенно упрощаются, если обеспечено рациональное разделение ее логики. При таком подходе оказывается полезным и объектно-ориентированное программирование, основы которого рассматривались в начале этой книги. Гораздо лучше иметь множество небольших компонентов, которые взаимодействуют друг с другом, чем небольшой набор больших, громоздких файлов. При таком разделении компонентов сложная архитектура программной системы становится гораздо более простой.

И наконец, для одной модели можно реализовать несколько контроллеров и компонентов представления. Вернемся к предыдущему примеру с карманными компьютерами. При использовании архитектуры MVC можно реализовать отдельные контроллеры и компоненты представления для карманных компьютеров, мобильных телефонов и даже устаревших браузеров. Все эти компоненты могут взаимодействовать с одной и той же моделью. Без архитектуры MVC будет невозможно повторно использовать разработанный код, и вы не сможете избежать дополнительных усилий и дублирования кода.

Как можно видеть, модель MVC — это очень полезный шаблон проектирования, который нужно стараться использовать везде, где только можно. В следующих разделах будут подробно рассмотрены все ее компоненты, а также способы их взаимодействия друг с другом. Что еще более важно, вы узнаете, как модель MVC может применяться при разработке архитектуры Web-приложений на языке PHP.

Модель

Модель является “сердцем” любого приложения. Вообще говоря, с моделью связан набор классов, разработанных для управления процессами, составляющими поведение программной системы. Это поведение связано с извлечением данных, являющихся основой для выходной информации приложения, а также с манипулированием данными, представляющими собой результат пользовательского ввода. Модель напрямую связана как с контроллером, так и с компонентом представления. Контроллер передает модели требуемые инструкции, а компонент представления (вид) преобразует данные, полученные от модели, в дружественный для пользователя формат.

Вид

Этот компонент модели MVC имеет самое непосредственное отношение к пользователям. Он обеспечивает представление данных, запрашиваемых пользователем и полученных от модели, или другой информации, которую модель “считает” нужным ему предоставить. Компонент вида напрямую соединен с моделью. Модель напрямую предоставляет данные компоненту представления для их отображения на конкретном устройстве. Можно сказать, что компонент представления связан также с контроллером в том смысле, что он обеспечивает физическое представление необходимых управляющих элементов, используемых для ввода данных.

Контроллер

Контроллер представляет собой пользовательский интерфейс к модели. Он предоставляет необходимые управляющие элементы, с помощью которых пользователи могут вводить в приложение данные и запросы на их получение. Контроллер связан с моделью, которой передает данные и запросы. После этого модель пытается удовлетворить поступивший запрос либо сохраняет полученные данные в хранилище.

Инфраструктура

Инфраструктура (*infrastructure*) — это, по существу, четвертый компонент модели MVC. Попросту говоря, основная задача инфраструктуры заключается в обеспечении совместной работы модели, контроллера и вида. Реализация инфраструктуры сильно зависит от языка программирования. В данной главе вы узнаете, как это можно реализовать на языке PHP.

MVC в Web-приложениях

Модель MVC не является новой концепцией. Она представляет собой определенное развитие модели IPO (Input, Processing, Output — вход, обработка выход), которая раньше успешно применялась при разработке простых текстовых приложений. В контексте Web использовать модель MVC относительно просто.

Если при разработке Web-приложения вы используете объектно-ориентированный подход, то модель представляется набором классов. При использовании любого языка Web-программирования эти классы должны обеспечивать базовое взаимодействие с любыми внешними источниками данных, принятие важных бизнес-решений, а также выполнять анализ и обработку входных и выходных данных.

Компонент “вид” представляется Web-браузером, а точнее, теми элементами, которые в нем отображаются. При передаче запроса, будь-то простой запрос на страницу, либо инструкция на обновление базы данных или извлечение из нее информации, предоставляемые моделью выходные данные отображаются в Web-браузере.

Возможно, это может сбить с толку, но контроллер также представляется Web-браузером, а точнее действиями, которые в нем выполняют пользователи. При заполнении полей сложной формы или щелчках на гиперссылках генерируемые браузером запросы GET и POST являются представлением усилий пользователей, направленных на передачу исходных данных в модель.

MVC в языке PHP

Как уже упоминалось выше, простота и доступность языка PHP зачастую обуславливает его некорректное использование. Это приводит к разработке приложений, которые очень трудно поддерживать. В частности, в контексте модели MVC это приводит к тому, что компоненты “модель”, “вид” и “контроллер” размещаются в одном сценарии. Именно при таком подходе можно говорить о *сценарии*. В свою очередь, при корректной реализации шаблона MVC сценариев в приложении не “существует”, а есть только компоненты.

Как не нужно делать

Предложите любому неопытному PHP-разработчику создать приложение гостевой книги, и он наверняка реализует его в виде единственного сценария. Этот файл может называться `guestbook.php` и выполнять как отображение существующих записей книги, так и добавление в базу данных новых записей. Его код будет иметь следующие особенности.

- ❑ Номер отображаемой страницы передается с помощью параметра метода GET.
- ❑ Если номер страницы не указан, отображается страница 1.
- ❑ Выполняется проверка наличия параметра `NewGuestBookEntry`. При его наличии выполняется проверка соответствия ограничениям (по длине и содержимому) и данные заносятся в базу. При его отсутствии выводится сообщение об ошибке. Сценарий завершает работу.
- ❑ Из базы данных извлекаются записи, соответствующие данному номеру страницы.
- ❑ Эти данные заносятся в таблицу HTML (описанную прямо в файле сценария).
- ❑ Пользователю выводится форма HTML, позволяющая ввести новую запись.

Этот подход содержит множество проблем, связанных не только с использованием метода GET для внесения изменения в базу данных. Если посмотреть на такую реализацию с точки зрения шаблона MVC, то окажется, что многочисленные компоненты контроллера и вида переплетаются друг с другом, создавая множество сложностей.

Поэтому рассмотрим более строгий подход, соответствующий шаблону проектирования MVC.

Расставим все по местам: подход MVC

Первое правило MVC — разбивайте сценарий на отдельные файлы. Зачастую для представления различных компонентов шаблона MVC используются файлы с различными расширениями, которые объединяются с помощью директивы `require_once`.

При этом не нужно следовать привычной практике именования файлов, при которой включаемым файлам присваивается расширение `.inc`. Это расширение ничего не говорит о содержимом файла и не позволяет различать их роли.

Лучше воспользоваться следующими расширениями.

Расширение	Компонент	Роль
<code>.php</code>	Инфраструктура	Логика поведения — инфраструктура взаимодействия между моделью, видом и контроллером
<code>.phpm</code>	Модель	Классы PHP — “сердце” приложения
<code>.phtml</code>	Вид/контроллер	Обработка результатов ввода пользователя и пользовательский интерфейс

Может показаться странным, что контроллер и вид объединены в один файл, но это соответствует физическому прототипу. Поскольку Web-браузер пользователя выступает и в роли интерфейса (контроллера), и в качестве средства отображения данных (вида), то резонно объединить эти компоненты в едином файле.

При делении сценария на модули следует руководствоваться такими правилами.

- Файл `.php` (управляющая страница) не должен содержать SQL-запросов или кода HTML.
- Файл `.phpm` (классы) не должен содержать HTML-код.
- Файл `.phtml` (шаблоны) не должен содержать SQL-запросов, а только базовые операторы PHP (`for`, `if`, `while`).

Такое разделение обеспечивает очень примитивную форму *естественных шаблонов* (native templating). В такой постановке интерпретатор PHP обрабатывает выходные данные приложения в отдельном файле шаблонов (в данном случае в файле `.phtml`). Выход определяется входными данным пользователя с помощью обработки в файлах логики (`.php`) и файлах классов (`.phpm`).

На практике этот подход реализуется следующим образом.

- Файл `.php` получает запросы GET или POST (поэтому не приходится реконфигурировать сервер Apache для поддержки новых расширений файлов).
- В файле `.php` содержатся все операторы `require`, необходимые для импортирования требуемых классов (каждый из которых хранится в файле `.phpm`).
- Файл `.php` проверяет входные параметры, передаваемые с помощью метода GET или POST, и определяет ход дальнейших действий. Он передает полученные данные (либо в виде строки, либо в обработанном виде) методам классов, включенным на предыдущем этапе.
- Классы анализируют входные данные, обрабатывают их, выполняют запросы к базе данных или другим источникам и возвращают данные файлу `.php`.

- ❑ Файл .php проверяет полученные результаты и принимает решение о дальнейшем использовании данных. Передача этих данных всегда выполняется с помощью построения единой суперпеременной (обычно хеш-таблицы), содержащей неформатированные данные, которые должны отображаться для пользователя.
- ❑ В файле .php с помощью директивы include подключается соответствующий файл .phtml.
- ❑ Файл .phtml анализирует эту суперпеременную и отображает ее пользователю в соответствующем виде на Web-странице.

Этот процесс может показаться сложнее, чем есть на самом деле. Ниже в этой главе будут приведены примеры, позволяющие оценить простоту данного подхода. А сейчас рассмотрим несколько классов, которые полезны при реализации шаблона проектирования MVC.

Потом вы узнаете, как этот набор классов интегрировать в описанный шаблон проектирования.

Минимальный набор классов для реализации MVC

Предлагаемый набор существенно облегчает реализацию шаблона проектирования MVC. Он связан только с реализацией инфраструктуры MVC. Позднее в этой главе речь пойдет о компоненте Smarty из пакета PEAR, обеспечивающем эффективную реализацию компонентов модели и вида.

Знакомство с набором

Этот минимальный набор обеспечивает взаимосвязь между контроллером и моделью. Основу данного подхода составляет объект запроса, обеспечивающий объектно-ориентированное представление вводимых пользователем данных. Он поддерживает применение ограничений для каждого параметра.

Концепция ограничений не только позволяет выявлять возможные ошибки ввода, но и отделить контроллер от модели. Проверки корректности данных в рамках модели не выполняются, поскольку модель перекладывает эту задачу на плечи инфраструктуры шаблона MVC.

Это довольно логично. Проверка корректности не относится к функциональности модели. Возлагая обязанность проверки корректности данных на инфраструктуру, вы можете применять различные правила для разных контроллеров и видов. Во многом в этом состоит философия шаблона MVC.

Вы можете расширить и модифицировать предлагаемые классы, которые обеспечивают только реализацию базовых концепций. Например, можно добавить методы автоматического сохранения переменных GET или POST с помощью скрытых параметров формы без участия переменных сеанса. Это полезно для обеспечения переходов между страницами, поддержки критерии поиска и т.п. Эту функциональность можно легко интегрировать в сам объект запроса.

Перечисленные здесь типы ограничений не охватывают всех возможных ситуаций. Поэтому при необходимости вы можете добавить свои ограничения.

Константы

Для совместной работы классов необходимо определить несколько констант, относящихся к трем типам параметров (GET, POST и COOKIES), а также различным типам ограничений.

Рассмотрим следующий код, содержащийся в файле constants.phpm.

```
<?php
// Константы

define(VERB_METHOD_COOKIE, 1);
define(VERB_METHOD_GET, 2);
define(VERB_METHOD_POST, 4);

define(CT_MINLENGTH, 1);
define(CT_MAXLENGTH, 2);
define(CT_PERMITTEDCHARACTERS, 3);
define(CT_NONPERMITTEDCHARACTERS, 4);
define(CT_LESSTHAN, 5);
define(CT_EQUALTO, 6);
define(CT_MORETHAN, 7);
define(CT_NOTEQUALTO, 8);
define(CT_MUSTMATCHREGEXP, 9);
define(CT_MUSTNOTMATCHREGEXP, 10);
?>
```

При определении этих констант используется две схемы нумерации. Экспоненциальная последовательность применяется для определения трех параметров HTTP-запроса, а обычная последовательность — для определения типов ограничений.

Эти две схемы именования связаны с выполнением побитовых операций. Поскольку этот вопрос еще не затрагивался, кратко остановимся на нем.

Рассмотрим следующий фрагмент кода.

```
$var3 = ($var1 && $var2);
```

Переменная \$var3 принимает значение true только в том случае, если истинны обе переменные в правой части выражения, \$var1 и \$var2 (в соответствии с булевой логикой операции &&). Но существует и другая форма логики сравнения, основанная на использовании базовых операторов AND и OR.

Рассмотрим следующее выражение.

```
$var3 = ($var1 & $var2);
```

Обратите внимание на использование одного амперсанда в правой части выражения. Это означает, что при выполнении операции сравнения используется не булева логика, а побитовая операция. Если переменные \$var1 и \$var2 являются целыми числами, то они представляются в виде двоичной последовательности, к каждому элементу которой применяются правила булевой логики сравнения. Если переменная \$var1 равна 43, а \$var2 — 11, то будет получен следующий результат.

34	=	00100010				
11	=	00001011				
34 & 11 =		00000010	=	2		

Результат выполнения этой операции равен 2, поскольку в двоичном представлении операндов совпадает и равен 1 только второй разряд.

Но какое отношение все это имеет к константам?

Это позволяет объединять константы в единое значение, содержащее арифметическую сумму соответствующих констант, а затем на основе этой суммы легко определять ее составляющие.

Если константа представлена степенью 2, то на основе суммы таких чисел легко определить сами слагаемые.

```
CONST_A =      1      =
CONST_B =      2      =
CONST_C =      4      =
CONST_D =      8      =
CONST_E =     16      =

```

Допустим, необходимо определить параметр, содержащий константы B, D и E. Складывая числа 2, 8 и 16, получим значение 26. Рассмотрим двоичные представления слагаемых и их суммы.

```
CONST_B =      2      =
CONST_D =      8      =
CONST_E =     16      =
CONST_B+D+E =  26      =

```

Обратите внимание, что из двоичного представления числа 26 легко определить, что оно составляет сумму чисел B, D и E.

Но как использовать это свойство в коде приложения?

Ведите и запустите следующий код из командной строки (а не в Web-браузере).

```
define('CONST_A', 1);
define('CONST_B', 2);
define('CONST_C', 4);
define('CONST_D', 8);
define('CONST_E', 16);

$myCombinedConstant = CONST_B + CONST_D + CONST_E;

if ($myCombinedConstant & CONST_A) {
    print "Результирующая константа содержит A\n";
}
if ($myCombinedConstant & CONST_B) {
    print "Результирующая константа содержит B\n";
}

if ($myCombinedConstant & CONST_C) {
    print "Результирующая константа содержит C\n";
}

if ($myCombinedConstant & CONST_D) {
    print "Результирующая константа содержит D\n";
}

if ($myCombinedConstant & CONST_E) {
    print "Результирующая константа содержит E\n";
}
```

Результат будет выглядеть следующим образом.

```
Результирующая константа содержит B
Результирующая константа содержит D
Результирующая константа содержит E
```

Как видно из этого примера, для определения слагаемых суммы можно просто применить побитовый оператор AND к сумме и исходным константам. Ненулевой (истинный) результат будет получен только при совпадении единичных битов в двух операндах.

Этот прием будет использован для работы с константами методов, а не с константами ограничений. Дело в том, что в одном параметре метода может быть использована комбинация методов запроса, а каждому типу ограничения соответствует единственный объект. Для комбинации ограничений необходимо комбинировать эти объекты, а не соответствующие константы. Поэтому для констант ограничений используется обычная система нумерации (1, 2, 3, 4, 5) — эти значения никогда не придется складывать.

Класс запроса

Объект `request` представляет запрос пользователя на генерацию страницы. Подобный класс, объединяющий параметры доступа, используется и в ASP.

В отличие от ASP, в PHP объект `request` приходится создавать самостоятельно, но его конструктор не содержит никаких параметров. Он проверяет существующие хеш-значения `$_REQUEST`, `$_COOKIE`, `$_POST` и `$_GET` и заполняет соответствующие переменные-члены. Объект `request` можно использовать сразу же после создания. Однако для этого придется воспользоваться методами проверки синтаксиса, обеспечивающими классами `Constraint` и `ConstraintFailure`.

Рассмотрим следующий код, содержащийся в файле `request.phpm`.

```
<?
require_once("constants.phpm");
require_once("constraint.phpm");
require_once("constraintfailure.phpm");
class request {
    private $_arGetVars;
    private $_arPostVars;
    private $_arCookieVars;
    private $_arRequestVars;
    private $_objOriginalRequestObject;

    private $_blIsRedirectFollowingConstraintFailure;
    private $_blRedirectOnConstraintFailure;
    private $_strConstraintFailureRedirectTargetURL;
    private $_strConstraintFailureDefaultRedirectTargetURL;

    private $_arObjParameterMethodConstraintHash;
    private $_arObjConstraintFailure;
    private $_hasRunConstraintTests;

    function __construct($check_for_cookie = true) {
        // Импорт переменных
        global $_REQUEST;
        global $_GET;
        global $_POST;
        global $_COOKIE;
        $this->_arGetVars = $_GET;
        $this->_arPostVars = $_POST;
        $this->_arCookieVars = $_COOKIE;
        $this->_arRequestVars = $_REQUEST;
        if ($check_for_cookie) {
            if ($this->_arCookieVars["phprqcOriginalRequestObject"]) {
                $cookieVal = $this->_arRequestVars["phprqcOriginalRequestObject"];
                $this->_blIsRedirectFollowingConstraintFailure = true;
                if (strlen($cookieVal) > 0) {
                    $strResult = setcookie ("phprqcOriginalRequestObject", "", time() - 3600, "/");
                    $origObj = unserialize(stripslashes($cookieVal));
                    $this->_objOriginalRequestObject = &$origObj;
                }
            }
        }
    }
}
```

```

        $this->_arRequestVars["phprqcOriginalRequestObject"] = "";
        $this->_arGetVars["phprqcOriginalRequestObject"] = "";
        $this->_arPostVars["phprqcOriginalRequestObject"] = "";
    };
    $this->_blIsRedirectOnConstraintFailure = true;
} else {
    $this->_blIsRedirectOnConstraintFailure = false;
};
} else {
    $this->_blIsRedirectOnConstraintFailure = false;
};
$this->_arObjParameterMethodConstraintHash = Array();
$this->_arObjConstraintFailure = Array();
$this->_blHasRunConstraintTests = false;
}

function IsRedirectFollowingConstraintFailure() {
    return($this->_blIsRedirectOnConstraintFailure);
}

function GetOriginalRequestObjectFollowingConstraintFailure() {
    if ($this->_blIsRedirectOnConstraintFailure) {
        return($this->_objOriginalRequestObject);
    };
}

function SetRedirectOnConstraintFailure($blTrueOrFalse) {
    $this->_blRedirectOnConstraintFailure = $blTrueOrFalse;
}

function SetConstraintFailureRedirectTargetURL($strURL) {
    $this->_strConstraintFailureRedirectTargetURL = $strURL;
}

function SetConstraintFailureDefaultRedirectTargetURL($strURL) {
    $this->_strConstraintFailureDefaultRedirectTargetURL = $strURL;
}

function GetParameterValue($strParameter) {
    return($this->_arRequestVars[$strParameter]);
}

function GetParameters() {
    return($this->_arRequestVars);
}

function GetCookies() {
    return($this->_arCookieVars);
}

function GetPostVariables() {
    return($this->_arPostVariables);
}

function GetGetVariables() {
    return($this->_arGetVariables);
}

function AddConstraint($strParameter, $intMethod, $objConstraint) {
    $newHash["PARAMETER"] = $strParameter;
    $newHash["METHOD"] = $intMethod;
    $newHash["CONSTRAINT"] = $objConstraint;
    $this->_arObjParameterMethodConstraintHash[] = $newHash;
}

```

```

function TestConstraints() {
    $this->_blHasRunConstraintTests = true;
    $anyFail = false;
    for ($i=0; $i<=sizeof($this->_arObjParameterMethodConstraintHash)-1; $i++) {
        $strThisParameter =
            $this->_arObjParameterMethodConstraintHash[$i] ["PARAMETER"];
        $intThisMethod =
            $this->_arObjParameterMethodConstraintHash[$i] ["METHOD"];
        $objThisConstraint =
            $this->_arObjParameterMethodConstraintHash[$i] ["CONSTRAINT"];
        $varActualValue = "";
        if ($intThisMethod == VERB_METHOD_COOKIE) {
            $varActualValue = $this->_arCookieVars[$strThisParameter];
        };
        if ($intThisMethod == VERB_METHOD_GET) {
            $varActualValue = $this->_arGetVars[$strThisParameter];
        };
        if ($intThisMethod == VERB_METHOD_POST) {
            $varActualValue = $this->_arPostVars[$strThisParameter];
        };
        $intConstraintType = $objThisConstraint->GetConstraintType();
        $strConstraintOperand = $objThisConstraint->GetConstraintOperand();

        $thisFail = false;
        $objFailureObject = new constraintfailure($strThisParameter,
                                                   $intThisMethod, $objThisConstraint);
        switch ($intConstraintType) {
            case CT_MINLENGTH:
                if(strlen((string)$varActualValue) < (integer)$strConstraintOperand)
                {
                    $thisFail = true;
                };
                break;
            case CT_MAXLENGTH:
                if(strlen((string)$varActualValue) > (integer)$strConstraintOperand)
                {
                    $thisFail = true;
                };
                break;
            case CT_PERMITTEDCHARACTERS:
                for ($j=0; $j<=strlen($varActualValue)-1; $j++) {
                    $thisChar = substr($varActualValue, $j, 1);
                    if (!strpos($strConstraintOperand, $thisChar) === false) {
                        $thisFail = true;
                    };
                };
                break;
            case CT_NONPERMITTEDCHARACTERS:
                for ($j=0; $j<=strlen($varActualValue)-1; $j++) {
                    $thisChar = substr($varActualValue, $j, 1);
                    if (!(!strpos($strConstraintOperand, $thisChar) === false)) {
                        $thisFail = true;
                    };
                };
                break;
            case CT_LESS THAN:
                if ($varActualValue >= $strConstraintOperand) {
                    $thisFail = true;
                };
                break;
            case CT_MORE THAN:
                if ($varActualValue <= $strConstraintOperand) {
                    $thisFail = true;
                };
                break;
        }
    }
}

```

```

    };
    break;
  case CT_EQUALTO:
    if ($varActualValue != $strConstraintOperand) {
      $thisFail = true;
    };
    break;
  case CT_NOTEQUALTO:
    if ($varActualValue == $strConstraintOperand) {
      $thisFail = true;
    };
    break;
  case CT_MUSTMATCHREGEXP:
    if (!preg_match($strConstraintOperand, $varActualValue)) {
      $thisFail = true;
    };
    break;
  case CT_MUSTNOTMATCHREGEXP:
    if (preg_match($strConstraintOperand, $varActualValue)) {
      $thisFail = true;
    };
    break;
  };
  if ($thisFail) {
    $anyFail = true;
    $this->_arObjConstraintFailure[] = $objFailureObject;
  };
}
if ($anyFail) {
  if ($this->blRedirectOnConstraintFailure) {
    $targetURL = $_ENV["HTTP_REFERER"];
    if (!$targetURL) {
      $targetURL = $this->_strConstraintFailureDefaultRedirectTargetURL;
    };
    if ($this->_strConstraintFailureRedirectTargetURL) {
      $targetURL = $this->_strConstraintFailureRedirectTargetURL;
    };
    if ($targetURL) {
      $objToSerialize = $this;
      $strSerialization = serialize($objToSerialize);
      $strResult = setcookie ("phprqcOriginalRequestObject",
$strSerialization, time() + 3600, "/");
      header("Location: $targetURL");
      exit(0);
    };
  };
  return(!($anyFail)); // Возвращает TRUE, если
                      // все тесты пройдены, иначе FALSE
}
}

function GetConstraintFailures() {
  if (!$this->blHasRunConstraintTests) {
    $this->TestConstraints();
  };
  return($this->_arObjConstraintFailure);
}
?>
```

Рассмотрим переменные-члены этого класса. Все они являются закрытыми, а доступ к ним обеспечивается через соответствующие методы. При этом мы продолжаем использовать венгерскую запись, т.е. в начале имени каждой переменной указываем ее тип.

Переменная-член	Роль
<code>\$_arGetVars</code>	Копия массива <code>\$_GET</code> , содержащая переменные запроса GET
<code>\$_arPostVars</code>	Копия массива <code>\$_POST</code> , содержащая переменные запроса POST
<code>\$_arCookieVars</code>	Копия массива <code>\$_COOKIE</code> , содержащая данные cookie, передаваемые в данном запросе
<code>\$_arRequestVars</code>	Копия массива <code>\$_REQUEST</code> , содержащая переменные GET и POST, а также cookie
<code>\$_objOriginalRequestObject</code>	Содержит копию исходного объекта <code>request</code> , используемую в случае перенаправления пользователя на исходную страницу при неправильном задании параметров
<code>\$_blIsRedirectFollowingConstraintFailure</code>	Логическая переменная, определяющая факт создания объекта <code>request</code> в результате перенаправления, выполняемого при нарушении ограничений
<code>\$_blRedirectOnConstraintFailure</code>	Логическая переменная, используемая при нарушении ограничений и определяющая необходимость автоматического перенаправления пользователя на исходную страницу
<code>\$_strConstraintFailureRedirectTargetURL</code>	Строковая переменная, используемая при нарушении ограничений. Если переменная <code>\$_blRedirectOnConstraintFailure</code> принимает значение <code>true</code> , а данная строка является пустой, то перенаправление осуществляется по прежнему адресу URL
<code>\$_strConstraintFailureDefaultRedirectTargetURL</code>	Строковая переменная, определяющая адрес URL, по которому перенаправляется пользователь, если переменная <code>\$_strConstraintFailureRedirectTargetURL</code> не задана, а флаг <code>\$_blRedirectOnConstraintFailure</code> содержит значение <code>true</code>
<code>\$_arObjParameterMethodConstraintHash</code>	Массив констант, индексы которых содержат три ключевых компонента в хеше. Первым является имя параметра, к которому применяется ограничение. Второй — это метод, которому передается данный параметр (как константа). Третий — объект ограничения
<code>\$_arObjConstraintFailure</code>	Массив объектов нарушения ограничений
<code>\$_blHasRunConstraintTests</code>	Логическая переменная, определяющая успешность проверки ограничений для данного объекта запроса

В следующей таблице описаны методы объекта `request`.

Метод	Роль
<code>__construct</code>	Создает объект класса <code>request</code> . Не содержит параметров. Проверяет значения <code>\$_REQUEST</code> , <code>\$_POST</code> , <code>\$_GET</code> , <code>\$_COOKIE</code> и заполняет переменные-члены. Проверяет существование данных cookie под именем <code>phprqcOriginalRequestObject</code> . При их существовании объект <code>request</code> передается обратно, поскольку ограничения нарушены, а данные cookie обнуляются, чтобы предотвратить бесконечный цикл. Затем содержимое преобразуется в новый объект запроса (с помощью функции <code>stripslashes()</code>). Доступ к этому объекту осуществляется с помощью метода <code>GetOriginalRequestObjectFollowingConstraintFailure</code> . Этот метод описывается ниже в данной таблице

Продолжение таблицы

Метод	Роль
IsRedirectFollowingConstraintFailure	Этот метод возвращает значение <code>true</code> , если страница автоматически отображается вследствие перенаправления 302 из-за нарушения ограничений
GetOriginalRequestObjectFollowingConstraintFailure	Если страница автоматически отображается вследствие перенаправления 302 из-за нарушения ограничений, этот метод возвращает объект <code>request</code> , созданный при вызове этой страницы
SetRedirectOnConstraintFailure	Этот метод позволяет сообщить объекту <code>request</code> , нужно ли выполнять перенаправление 302 при нарушении ограничений. Решение принимается на основе значения логического параметра метода. Целевая страница для данного перенаправления определяется с помощью методов <code>SetConstraintFailureRedirectTargetURL</code> и <code>SetConstraintFailureDefaultRedirectTargetURL</code>
SetConstraintFailureRedirectTargetURL	Задает целевую страницу для перенаправления 302 в случае нарушения ограничений. Если она не задана, используется либо исходная страница, либо используемый по умолчанию адрес URL, который задается с помощью метода <code>SetConstraintFailureDefaultRedirectTargetURL</code>
SetConstraintFailureDefaultRedirectTargetURL	Если с помощью предыдущего метода не задан адрес URL целевой страницы и не задана никакая другая страница, этот метод определяет адрес URL, используемый по умолчанию
GetParameterValue	Возвращает значение указанного параметра, получаемое из массива <code>\$_REQUEST</code>
GetParameters	Возвращает хеш-значение параметров, по существу копию массива <code>\$_REQUEST</code>
GetCookies	Возвращает хеш-значение данных cookie, по существу копию массива <code>\$_COOKIE</code>
GetPostVariables	Возвращает хеш-значение параметров POST, по существу копию массива <code>\$_POST</code>
GetGetVariables	Возвращает хеш-значение параметров GET, по существу копию массива <code>\$_GET</code>
AddConstraint	Добавляет ограничения к данному запросу. Это ограничение применяется к единственному параметру и задает необходимые условия. Методу передаются имя параметра, метод передачи (<code>VERB_METHOD_GET</code> , <code>VERB_METHOD_POST</code> или <code>VERB_METHOD_COOKIE</code> как константа) и объект ограничений. После добавления ограничения его нельзя удалить. Одному параметру может соответствовать несколько ограничений
TestConstraints	Проверяет каждое ограничение, уведомляет о результатах проверки путем заполнения переменной-члена <code>\$_arObjConstraintFailure</code> , доступ к которой осуществляется с помощью метода <code>GetConstraintFailures</code> . Если переменная-член <code>\$_blRedirectOnConstraintFailure</code> принимает значение <code>true</code> , выполняется перенаправление по соответствующему адресу URL. При этом создаются временные данные cookie, представляющие собой сериализованное представление текущего объекта запроса

Окончание таблицы

Метод	Роль
GetConstraintFailures	Возвращает массив объектов constraintfailure. Если в методе TestConstraints происходит перенаправление вследствие нарушения ограничений, данный метод применяется к исходному объекту запроса, доступ к которому осуществляется с помощью метода GetOriginalRequestObjectFollowingConstraintFailure (описанному выше)

Возможно, эта теория покажется вам непонятной. Однако ниже в этой главе будет приведен практический пример использования данного класса в реальном приложении. Теперь рассмотрим класс ограничений и близкий ему класс constraintfailure.

Класс ограничений

Класс constraint обеспечивает объектно-ориентированную инкапсуляцию ограничений, никак не связанную с проверкой их выполнения. Проверку выполнения ограничений осуществляет класс request.

Рассмотрим следующий код, содержащийся в файле constraint.phpm.

```
<?php
require_once("constants.phpm");
class constraint {
    private $_intConstraintType;
    private $_strConstraintOperand;

    function __construct($intConstraintType, $strConstraintOperand) {
        $this->_intConstraintType = $intConstraintType;
        $this->_strConstraintOperand = $strConstraintOperand;
    }

    function GetConstraintType() {
        return($this->_intConstraintType);
    }

    function GetConstraintOperand() {
        return($this->_strConstraintOperand);
    }
}
?>
```

Конструктор этого класса в качестве первого параметра получает тип ограничения (выбираемый из заданного списка), а в качестве второго — его операнд. Операнд зависит от типа ограничения. Интерпретация этого ограничения выполняется объектом класса request. Например, операнд ограничения CT_MAXLENGTH должен содержать максимальную длину, а операнд ограничения CT_PERMITTEDCHARS должен содержать строку, включающую все допустимые символы, используемые для данного параметра.

Для обеспечения гибкости операнд передается в виде строки, но в процессе соответствия ограничениям он преобразуется к естественному виду.

Заметим, что имя параметра не хранится в объекте ограничения. Ограничение — это условие, применяемое к любому параметру (или да же к нескольким параметрам) с помощью методов объекта запроса. Поэтому для инкапсуляции невыполнения ограничений необходимо хранить соответствующий параметр и метод проверки. Это обеспечивается с помощью класса constraintfailure.

Класс constraintfailure

Объект constraintfailure инстанцируется при нарушении некоторого ограничения. Конструктору этого объекта передается имя параметра, метод передачи (VERB_METHOD_GET, VERB_METHOD_POST или VERB_METHOD_COOKIE) и исходный объект ограничения.

Метод GetFailedConstraintObject обычно используется для определения типа ограничения.

Рассмотрим следующий код, содержащийся в файле constraintfailure.phpm.

```
<?php
require_once("constants.phpm");
require_once("constraint.phpm");
class constraintfailure {
    private $_strParameterName;
    private $_intVerbMethod;
    private $_objFailedConstraintObject;

    function __construct($strParameterName, $intVerbMethod, $objFailedConstraintObject) {
        $this->_strParameterName = $strParameterName;
        $this->_intVerbMethod = $intVerbMethod;
        $this->_objFailedConstraintObject = $objFailedConstraintObject;
    }

    function GetParameterName() {
        return($this->_strParameterName);
    }

    function GetVerbMethod() {
        return($this->_intVerbMethod);
    }

    function GetFailedConstraintObject() {
        return($this->_objFailedConstraintObject);
    }
}
```

Этот класс инкапсулирует нарушение ограничения при вводе пользователем данных, не соответствующих заданному критерию.

В данном классе хранится имя параметра, метод его передачи (GET, POST, COOKIE) и ограничение (в виде объекта constraint).

Экземпляры этого объекта извлекаются из класса request с помощью метода GetConstraintFailures.

Использование предложенного набора

Познакомившись с основными классами набора, можно рассмотреть их использование в реальном приложении.

Приведенный пример будет основываться на шаблоне MVC.

Предположим, вы разрабатываете систему поиска, состоящую из двух страниц: страницы, на которой вводятся критерии поиска, и страницы с результатами.

search.php

Исходная страница поиска выполняет следующие функции.

- Импортирует все необходимые компоненты.
- Определяет файл шаблона (search.phtml).

- Определяет ассоциативный массив (`$displayHash`), содержащий данные, обрабатываемые файлом шаблона.
- Проверяет выполнение ограничений.
- В случае нарушения ограничений это отражается в массиве `$displayHash`, а исходный объект запроса доставляется с помощью временных данных `cookie`.
- Из полученного объекта запроса извлекается массив объектов `constraintfailure`.
- На основе массива `constraintfailure` в цикле заполняется вновь созданный массив `$displayHash`.
- Для обработки этих данных подключается файл шаблона.

Приведем код этого файла.

```
<?
require_once("constants.phpm");
require_once("request.phpm");
require_once("constraint.phpm");
require_once("constraintfailure.phpm");

$strTemplateFile = "search.phtml";

$displayHash = Array();

$objRequest = new request();
$blHadProblems = ($objRequest->IsRedirectFollowingConstraintFailure());

$displayHash["HADPROBLEMS"] = $blHadProblems;

if ($blHadProblems) {
    $objFailingRequest = $objRequest-
>GetOriginalRequestObjectFollowingConstraintFailure();
    $arConstraintFailures = $objFailingRequest->GetConstraintFailures();
    $displayHash["PROBLEMS"] = Array();
    for ($i=0; $i<=sizeof($arConstraintFailures)-1; $i++) {
        $objThisConstraintFailure = &$arConstraintFailures[$i];
        $objThisFailingConstraintObject = $objThisConstraintFailure-
>GetFailedConstraintObject();
        $intTypeOfFailure = $objThisFailingConstraintObject->GetConstraintType();
        switch ($intTypeOfFailure) {
            case CT_MINLENGTH:
                $displayHash["PROBLEMS"] [] = "Строка поиска слишком коротка.";
                break;
            case CT_MAXLENGTH:
                $displayHash["PROBLEMS"] [] = "Строка поиска слишком длинна.";
                break;
            case CT_PERMITTEDCHARACTERS:
                $displayHash["PROBLEMS"] [] = "Строка поиска содержит неизвестные символы.";
                break;
        };
    };
}

require_once($strTemplateFile);
exit(0);
?>
```

Обратите внимание на использование оператора `exit` в конце приведенного фрагмента. Он является необязательным, но предотвращает случайную запись кода при передаче управления шаблону.

Теперь рассмотрим сам шаблон.

search.phtml

Этот шаблон отвечает исключительно за интеллектуальную обработку массива \$displayHash и его представление в виде кода HTML. Обратите внимание, что массив \$displayHash не нужно объявлять глобальным, поскольку оператор require подключает его в конце исходного сценария, и интерпретатор обрабатывает этот шаблон как единый сценарий.

В этом шаблоне необходимо создать форму поиска и проверить сообщения об ошибках, генерированных в результате поиска.

```
<html>
<head>
    <title>Страница поиска Эда</title>
</head>

<body>
<H1>Страница поиска Эда</H1>
<hr>
Здесь можно определить типы стейка.
<BR><BR>
<? if ($displayHash["HADPROBLEMS"]) { ?>
    <B>Извините, <?=(sizeof($displayHash["PROBLEMS"]) > 1) ?
        "возникли проблемы" : "возникла проблема")?> в процессе поиска!</B>
    <? for ($i=0; $i<=sizeof($displayHash["PROBLEMS"])-1; $i++) { ?>
        <?=$displayHash["PROBLEMS"][$i]?>
    <? } ; ?>
<? } ; ?>
<FORM METHOD="GET" ACTION="searchresults.php">
    <TABLE BORDER="0">
        <TR>
            <TD>Тип стейка</TD>
            <TD><INPUT TYPE="TEXT" NAME="typeOfSteak"></TD>
        </TR>
    </TABLE><BR>
    <INPUT TYPE="SUBMIT" VALUE = "Передать запрос">
</FORM>

</body>
</html>
```

В этом коде содержатся только базовые операторы PHP. Если окажется, что в файле шаблона вам придется использовать более сложный код, вероятно, его следует перенести на управляющую страницу (.php) или в один из классов.

В следующей части этой главы будет описан альтернативный метод использования реальных шаблонов, не предполагающий программирования на языке PHP.

При запуске страницы search.php в окне браузера появится стандартная форма поиска (рис. 13.1).

Теперь разработаем страницу результатов. Она тоже будет состоять из двух частей: управляющей части и шаблона.

searchresults.php

Страница результатов поиска будет выводиться в случае успешного ввода параметров поиска.

При неправильном задании параметров поиска пользователь будет перенаправлен к исходной странице, на которой будут отображены сообщения об ошибках.

В следующем примере используется только один параметр поиска — введенная пользователем строка. Она должна содержать от 3 до 12 символов, включающих только буквы.

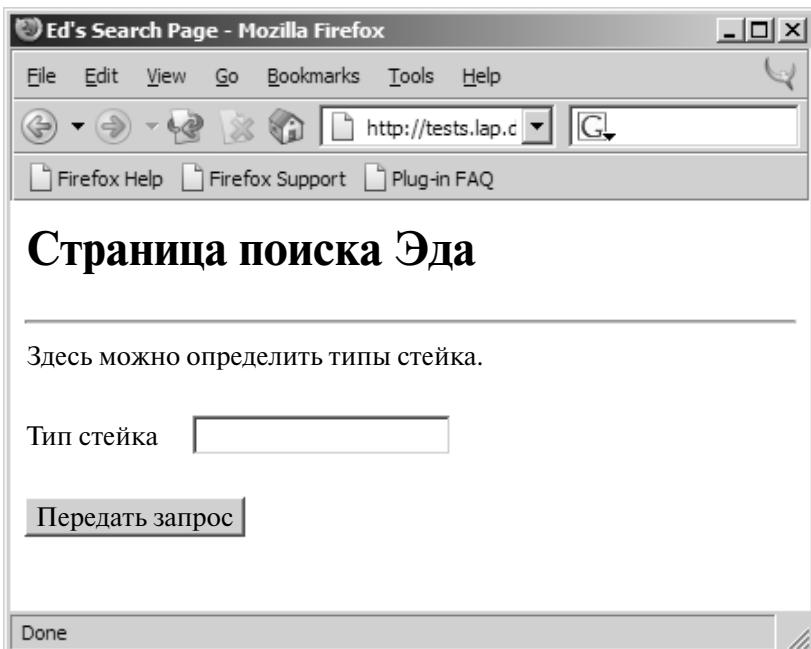


Рис. 13.1.

```

<?
require_once("constants.phpm");
require_once("request.phpm");
require_once("constraint.phpm");
require_once("constraintfailure.phpm");

$strTemplateFile = "searchresults.phtml";
$displayHash = Array();

$objRequest = new request();
$objRequest->SetRedirectOnConstraintFailure(true);

$objConstraint = new constraint(CT_MINLENGTH, "3");
$objRequest->AddConstraint("typeOfSteak", VERB_METHOD_GET, $objConstraint);
$objConstraint = new constraint(CT_MAXLENGTH, "12");
$objRequest->AddConstraint("typeOfSteak", VERB_METHOD_GET, $objConstraint);
$objConstraint = new constraint(CT_PERMITTEDCHARACTERS,
"ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz");
$objRequest->AddConstraint("typeOfSteak", VERB_METHOD_GET, $objConstraint);

$objRequest->SetConstraintFailureDefaultRedirectTargetURL("/search.php");
$objRequest->TestConstraints();

# Если мы дошли до этого места, все проверки выполнены
# и можно начинать поиск.
$displayHash["RESULTS"] = Array();
$arSteaks = array("ребрышки", "задняя часть", "филейная часть", "зажаренный");

for ($i=0; $i<=sizeof($arSteaks)-1; $i++) {

```

```

if (!strpos(trim(strtolower($arSteaks[$i])), strtolower(trim
    ($objRequest->GetParameterValue("typeOfSteak")))) === false) {
    array_push($displayHash["RESULTS"], $arSteaks[$i]);
}
};

require_once($strTemplateFile);
exit(0);
?>

```

Обратите внимание, что при повторном инстанцировании объекта запроса снова определяется переменная \$displayHash.

Каждое ограничение задается как новый экземпляр класса constraint. Он добавляется с помощью метода AddConstraint, которому передаются метод (в данном случае GET), имя параметра и дополнительный операнд. Затем задается используемый по умолчанию адрес URL (адрес исходной страницы поиска). Если вы попробуете запустить данный пример, вам потребуется соответствующим образом настроить пути.

Затем вызывается единственный метод TestConstraints(), выполняющий проверку всех ограничений. Если хотя бы одно из ограничений нарушено, соответствующая информация добавляется в массив нарушений, содержащихся в объекте request.

Если нарушено хотя бы одно ограничение, пользователь перенаправляется на исходную страницу с помощью метода SetRedirectOnConstraintFailure(). В этом методе выполняется сериализация объекта request и его передача браузеру в виде данных cookie. В результате анализа этих данных пользователю выдаются сообщения об ошибках.

В случае такого перенаправления шаблон вообще не отображается, и пользователь вообще не узнает о существовании второй страницы. Ему покажется, что он не видел исходную страницу поиска.

Если все ограничения соблюdenы, выполняется поиск. В данном случае его механизм чрезвычайно прост, а в реальных приложениях для реализации поиска вам придется воспользоваться дополнительными классами.

После заполнения переменных результатов поиска шаблон выводится на экран.

searchresults.phtml

Шаблон отображения результатов поиска чрезвычайно прост. В нем не предусмотрён вывод сообщения “Поиск не дал никаких результатов”. Однако вывести такое сообщение не составит труда.

```

<html>
<head>
    <title>Страница поиска Эда</title>
</head>

<body>
<H1>Результаты поиска</H1>
<hr>
Бот результаты поиска.
<UL>
<? for ($i=0; $i<=sizeof($displayHash["RESULTS"])-1; $i++) { ?>
    <LI><?=$displayHash["RESULTS"][$i]?></LI>
<? } ; ?>
</UL>
<A HREF="search.php">Начните поиск сначала!</A>
</body>
</html>

```

Проверим код на практике

Откройте страницу поиска и введите какие-либо данные.

Для начала воспользуйтесь следующими рекомендациями. В поле поиска введите фил, и вы получите результат, показанный на рис. 13.2.

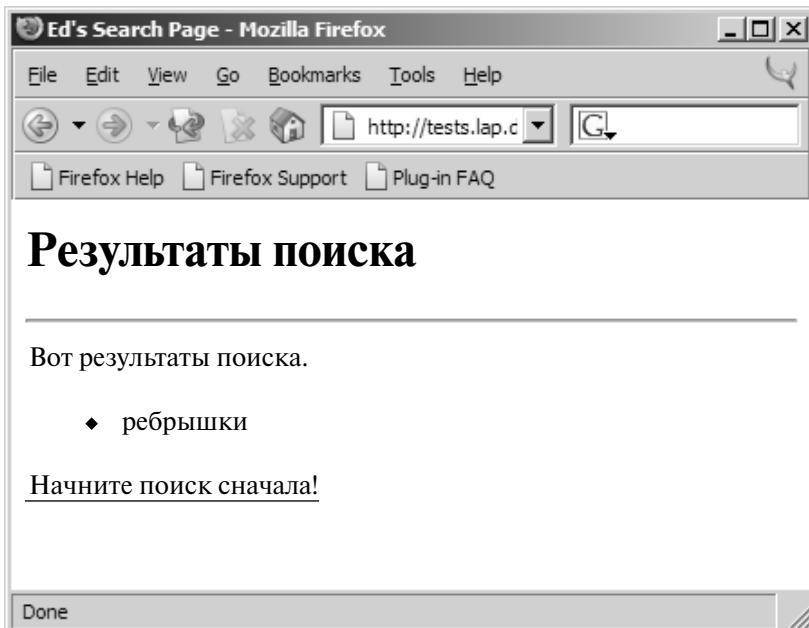


Рис. 13.2.

Все в порядке. Теперь введите фи. Поскольку длина слова не соответствует ограничениям, отобразится страница, показанная на рис. 13.3. В файле `searchresults.php` выполняется проверка ограничений. Поскольку одно из них нарушено, отображается исходная страница поиска с сообщениями об ошибках, полученными с помощью объекта `request`.

Попробуйте ввести слишком длинную строку или строку, содержащую цифры, и вы получите аналогичные сообщения об ошибках.

Применение разработанного набора

В начале этой главы был разработан набор классов, обеспечивающих обработку запросов и проверку ограничений на основе шаблона MVC.

Если вам понадобится использовать новые типы ограничений, возможно, требующие обращения к базе данных, класс `request` можно усложнить и расширить. Дальше речь пойдет о другой реализации шаблонов — применении пакета Smarty.

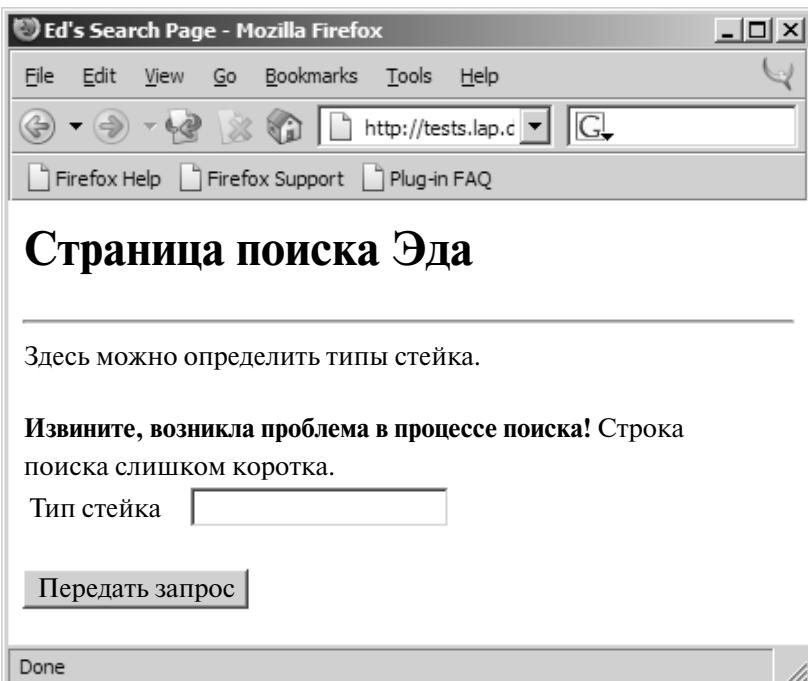


Рис. 13.3.

Реальные шаблоны

Чем же рассмотренные шаблоны PHP отличаются от реальных шаблонов?

Реализация шаблонов средствами PHP

Используемая до сих пор идея реализации шаблонов основывалась на заполнении единственного многомерного массива и формировании на его основе кода HTML с использованием операторов `if`, `for` и `while`. Однако данный подход имеет несколько недостатков.

Недостатки шаблонов на языке PHP

Первой и самой очевидной проблемой является использование очень мощного языка PHP для выполнения очень простой задачи анализа содержимого. Эта проблема не сводится к вопросам производительности. При использовании PHP для анализа содержимого могут возникнуть дополнительные ошибки.

Вспомним проблемы, с которыми приходится сталкиваться при написании больших классов: описки, ошибки в именах переменных, ошибочное использование переменных-счетчиков во вложенных циклах `for`. Эти же проблемы возникают при реализации шаблонов.

Возможны и более серьезные проблемы, возникающие при создании больших проектов. Одним из преимуществ применения шаблона MVC является возможность поручить реализацию контроллера и вида разработчикам, имеющим опыт проектирования

интерфейса и создания кода HTML. Тогда более квалифицированные специалисты смогут сосредоточиться на создании модели средствами PHP.

Это пример оптимального использования ресурсов. Такой подход не только позволяет сэкономить время и деньги, но и обеспечивает наилучшую реализацию как интерфейса, так и кода приложения.

Однако при реализации шаблонов на PHP у разработчиков не остается выбора. И дело не в том, что они должны хорошо знать язык PHP, а в том, что они должны себя ограничивать. При реализации таких шаблонов возможны бесконечные циклы, опасные системные вызовы и другие критические ошибки.

Этой проблемы можно избежать, если бы формирование выходной страницы выполнялось с использованием гибких дескрипторов на произвольном языке. Именно такое решение обеспечивает пакет Smarty.

Реализация шаблонов на основе пакета Smarty

Smarty – это пакет, позволяющий разработчикам готовить выходные данные для шаблона, предназначенного для их отображения.

Smarty допускает использование циклов, условных выражений и многомерных массивов. По существу, он реализует все традиционные методы PHP, используемые для реализации шаблонов на основе этого языка, без применения самого PHP.

Важным преимуществом Smarty является скорость его работы. Хотите верьте, хотите — нет, этот пакет на основе шаблонов создает код PHP, который затем кешируется и может использоваться повторно.

Установка пакета Smarty

Процесс установки Smarty довольно прост. Однако этот пакет не является частью пакета PEAR, поэтому его придется загружать и устанавливать вручную.

В настоящее время пакет Smarty можно найти по адресу <http://smarty.php.net>.

После загрузки архивного файла его нужно распаковать.

```
root@linuxvm:~# tar -xzvf Smarty-2.6.2.tar.gz
Smarty-2.6.2/
Smarty-2.6.2/COPYING.lib
Smarty-2.6.2/.cvsignore
Smarty-2.6.2/BUGS
Smarty-2.6.2/demo/
...
Smarty-2.6.2/misc/smarty_icon README
Smarty-2.6.2/misc/smarty_icon.gif
root@linuxvm:~#
```

Затем библиотеки Smarty необходимо перенести в каталоги, доступные интерпретатору PHP. Обычно для этого нужно просто скопировать все содержимое папки libs/ в папку /usr/local/lib/php с помощью следующей команды.

```
root@linuxvm:~/Smarty-2.6.2# cp -r libs/ /usr/local/lib/php
```

Затем необходимо подготовить свое приложение. В папках с PHP-файлами приложения, в которых будет использоваться Smarty, необходимо создать каталоги templates, templates_c, cache и configs.

```
root@linuxvm:~# cd public_html/tests/prophp5/mvc/
root@linuxvm:~/public_html/tests/prophp5/mvc/# mkdir templates_c
```

```
root@linuxvm:~/public_html/tests/prophp5/mvc/# mkdir configs
root@linuxvm:~/public_html/tests/prophp5/mvc/# mkdir templates
root@linuxvm:~/public_html/tests/prophp5/mvc/# mkdir cache
```

К счастью, пакет Smarty можно использовать и в тех проектах, в которых сценарии PHP содержатся в нескольких каталогах. При этом можно вручную указать, где будут располагаться эти папки. Об этом речь пойдет ниже в данной главе.

Затем для двух из этих папок (`templates_c` и `cache`) необходимо задать специальные разрешения. Владельцем этих папок должен быть тот же пользователь, который является владельцем Web-сервера. В большинстве случаев это пользователь `nobody`, но может быть выбран и любой другой. Если вы точно не знаете, кто является владельцем Web-сервера, проверьте это с помощью такой команды.

```
ps auxw | grep httpd | grep -v grep | awk '{print $1}' | tail -1
```

Главное, чтобы этот пользователь и группа, к которой он принадлежит, имел полные права на чтение и запись данных в эти папки. Настройте разрешения следующим образом.

```
chown nobody.users ./cache
chown nobody.users ./templates_c
chmod 770 ./cache
chmod 7780 ./templates_c
```

Режим 770 означает, что только владелец этих папок может читать, записывать и выполнять их содержимое. Если вы не уверены в возможности получения прав `root` на данной машине, то можете использовать режим 775, предполагающий доступ к папкам для чтения и записи и для других пользователей. Однако это влечет за собой определенный риск нарушения безопасности.

Использование пакета Smarty

Типичная реализация Smarty включает управляющий файл с расширением `.php` и файл шаблона с расширением `.tpl`.

`smartytest.php`

Рассмотрим простой пример.

```
<?
require('Smarty.class.php');

$objSmarty = new Smarty;
$strTemplate = "smartytest.tpl";

$objSmarty->assign("test_variable", "Это значение тестовой переменной");
$objSmarty->display($strTemplate);
?>
```

Для этого сценария необходимо обеспечить код шаблона, который будет располагаться в папке `templates`.

`smartytest.tpl`

Рассмотрим следующий код.

```
<html>
<head>
    <title>Без заголовка</title>
</head>

<body>
    Это очень простой шаблон!<BR>
    Значением переменной является: {$test_variable}

</body>
</html>
```

Несложно догадаться, что при выполнении приведенного выше сценария будет получен следующий результат.

Это очень простой шаблон!
Значением переменной является: Это значение тестовой переменной

Как видно из этого фрагмента, объекту шаблона Smarty присваивается переменная `test_variable`, которая выводится в самом шаблоне. Все довольно просто. Но как обеспечить вывод массивов, вывод данных при выполнении некоторых условий и все преимущества традиционной функциональности PHP?

Линейные массивы в Smarty

В Smarty довольно легко обрабатывать линейные массивы. Рассмотрим следующий фрагмент управляющего сценария на PHP.

```
$objSmarty->assign("FirstName", array("Джон", "Мэри", "Джеймс", "Генри"));
```

Как видно из этого фрагмента, переменные PHP передаются в шаблон Smarty как и обычная строка. Для обработки таких данных используется эквивалент цикла `for`.

```
{section name=x loop=$FirstName}
    {$FirstName[x]}<BR>
{/section}
```

Результат его выполнения имеет следующий вид.

Джон
Мэри
Джеймс
Генри

Ассоциативные массивы в Smarty

Аналогично, можно обрабатывать ассоциативные массивы (хеш-таблицы). Рассмотрим следующий управляющий код.

```
$arHash["Name"] = "Эд";
$arHash["Age"] = 22;
$arHash["Location"] = "Лондон";
$objSmarty->assign("Writer", $arHash);
```

Обработку данных в шаблоне можно реализовать следующим образом.

Автором этой главы является {\$Writer.Name}, в возрасте {\$Writer.Age} лет, который сейчас проживает в городе {\$Writer.Location}.

Обратите внимание на использование синтаксиса `{$X.Y}`, где `X` – имя переменной Smarty, а `Y` – ключ ассоциативного массива. В результате будет выведена следующая информация.

Автором этой главы является Эд, в возрасте 22 лет, который сейчас проживает в городе Лондон.

Условные выражения в Smarty

Вывод информации при выполнении некоторого условия очень полезен для отображения сообщений об ошибках.

При этом используются следующие переменные.

```
$objSmarty->assign("isError", 1);
$objSmarty->assign("ErrorText", "Вы не задали условие поиска.");
```

Для поддержки этих переменных используется следующая логика.

```
{if $isError ==1}
    Возникла следующая ошибка: {$ErrorText}
{/if}
```

При этом используется условное выражение, очень напоминающее соответствующий оператор PHP. Если значение переменной `isError` равно 1, будет выведено соответствующее сообщение.

Возникла следующая ошибка: Вы не задали условие поиска.

Для проверки работоспособности этого фрагмента измените значение переменной `isError` на 0, и никакое сообщение не будет выведено.

При реализации шаблонов на PHP можно просто проверить наличие сообщений об ошибках. Это же можно реализовать и в Smarty.

```
{if $ErrorText}
    Возникла следующая ошибка: {$ErrorText}
{/if}
```

При этом нет необходимости сообщать Smarty о возникновении ошибки. Шаблон определит это самостоятельно.

Преобразование сценария search.php на основе Smarty

Воспользуемся этим подходом в более сложной ситуации. С помощью Smarty можно переписать уже упоминавшиеся в этой главе файлы `search.php` и `search.phtml`.

Обратитесь к созданному ранее файлу `search.php` и подумайте, как его следует модифицировать.

Теперь при передаче в шаблон `search.phtml` переменная `$displayHash` может содержать следующие данные:

- логическую переменную `HADPROBLEMS`, представляющую в Smarty обычной переменной;
- массив строк `PROBLEMS`, каждая из которых описывает конкретную проблему и передается только в том случае, если переменная `HADPROBLEMS` принимает значение `true`.

Модифицируем файл search.php. Серым фоном выделены фрагменты кода, которые были изменены или добавлены в исходную версию.

```

require_once("constants.phpm");
require_once("request.phpm");
require_once("constraint.phpm");
require_once("constraintfailure.phpm");

$strTemplateFile = "s_search.tpl";

require('Smarty.class.php');
$objSmarty = new Smarty;

if ($blHadProblems) {
    $objSmarty->assign("HADPROBLEMS", "true");
}

if ($blHadProblems) {
    $objFailingRequest = $objRequest-
>GetOriginalRequestObjectFollowingConstraintFailure();
    $arConstraintFailures = $objFailingRequest->GetConstraintFailures();
    $problemArray = Array();
    for ($i=0; $i<=sizeof($arConstraintFailures)-1; $i++) {
        $objThisConstraintFailure = &$arConstraintFailures[$i];
        $objThisFailingConstraintObject = $objThisConstraintFailure-
>GetFailedConstraintObject();
        $intTypeOfFailure = $objThisFailingConstraintObject->GetConstraintType();
        switch ($intTypeOfFailure) {
            case CT_MINLENGTH:
                $displayHash["PROBLEMS"][] = "Строка поиска слишком коротка.";
                break;
            case CT_MAXLENGTH:
                $displayHash["PROBLEMS"][] = "Строка поиска слишком длинна.";
                break;
            case CT_PERMITTEDCHARACTERS:
                $displayHash["PROBLEMS"][] = "Строка поиска содержит неизвестные символы.";
                break;
        };
    };
}

if ($problemArray) {
    $objSmarty->assign("PROBLEMS", $problemArray);
}
$objSmarty->display($strTemplateFile);

```

Создайте соответствующий шаблон и сохраните его в папке templates под именем s_search.tpl.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
<title>Страница поиска Эда</title>

```

```

</head>

<body>
<H1>Страница поиска Эда</H1>
<hr>
Здесь вы можете найти типы стейков.
<BR><BR>
{if $ADPROBLEMS}
<B>Извините, в процессе поиска возникли проблемы!</B>
{section name=x loop=$PROBLEMS}
{$PROBLEMS[x]}
{/section}
{/if}
<FORM METHOD="GET" ACTION="searchresults.php">
<TABLE BORDER="0">
<TR>
<TD>Тип стейка</TD>
<TD><INPUT TYPE="TEXT" NAME="typeOfSteak"></TD>
</TR>
</TABLE><BR>
<INPUT TYPE="SUBMIT" VALUE = "Передать запрос">
</FORM>
</body>
</html>

```

Запустите обновленный сценарий `search.php` и постарайтесь сгенерировать ошибки. Реакция на них окажется почти прежней, значит, преобразование шаблона прошло успешно.

Есть лишь одна небольшая проблема. При возникновении одной ошибки вы по-прежнему получите сообщение о “проблемах”, возникших в процессе поиска.

В исходном шаблоне, сохраненном в файле `.phtml`, эта ситуация разрешалась с помощью условного оператора

```
<B>Извините <?=((sizeof(displayHash["PROBLEMS"])) > 1 ? "возникли проблемы" :
"возникла проблема")?> в процессе поиска!</B>
```

К счастью, это можно реализовать и при использовании Smarty, понадобится лишь использовать размер массива.

```
<B>Извините {if $PROBLEMS[1]}возникли проблемы{else}возникла проблема{/if} в
процессе поиска!</B>
```

Здесь для получения размера массива пришлось воспользоваться кодом PHP, проверяющим наличие второго элемента.

Теперь результат работы шаблона абсолютно одинаков.

Расширенные возможности пакета Smarty

Теперь познакомимся с некоторыми расширенными возможностями Smarty.

Параметры времени выполнения

Сначала разберемся с ситуацией, когда папки шаблона приходится создавать для проектов, включающих несколько каталогов с файлами `.php`. К счастью, пакет Smarty позволяет указать их местоположение в процессе выполнения.

```
$smarty->template_dir = '/web/www/mydomain.com/smarty/guestbook/templates/';
$smarty->compile_dir = '/web/www/mydomain.com/smarty/guestbook/templates_c/';
$smarty->config_dir = '/web/www/mydomain.com/smarty/guestbook/configs/';
$smarty->cache_dir = '/web/www/mydomain.com/smarty/guestbook/cache/';
```

Многомерные массивы

Многомерные массивы обрабатываются аналогично одномерным.

```
$multiArray = Array(Array("x", "o", "x"), Array("o", "x", "x"),
Array("o", "o", "x")); $objSmarty->assign("TicTacToBoard", $multiArray);
```

Выведем эти данные с помощью следующего шаблона.

```
<TABLE BORDER="1">
{section name=y loop=$TicTacToBoard}
<TR>
{section name=x loop=$TicTacToBoard[y]}
<TD>{$TicTacToBoard[y][x]}</TD>
{/section}
</TR>
{/section>
</TABLE>
```

Получим игру “крестики-нолики”, в которой победитель составил главную диагональ.

Модификаторы переменных

Шаблон Smarty позволяет использовать модификаторы переменных.

```
$smarty->assign('bookTitle', "PHP5 для начинающих");
```

В случае применения шаблона

```
{$bookTitle|replace: "начинающих": "профессионалов"}
```

результат будет очевидным.

Процедуры

В пакете Smarty определен ряд процедур. Они являются встроенными, т.е. не допускают модификации. Эти процедуры работают только с упомянутыми выше модификаторами типа `replace`, изменяющими входные данные перед отображением.

Полезным примером является процедура `strip`, обеспечивающая вывод файла HTML с корректной расстановкой символов табуляции и пробелов. Она удаляет все лишние пробелы перед отображением в браузере и облегчает поддержку шаблонов.

```
{strip}


|                           |
|---------------------------|
| Это некоторое содержимое. |
|---------------------------|


{/strip}
```

Такой текст легче воспринимается человеком. Однако при отправке в браузер исходный код совершенно не играет никакой роли, поэтому процедура `strip` удаляет все пробелы, которые могут повлиять на вид выходной информации.

```
<table border="0"><tr><td>Это некоторое содержимое.</td></tr></table>
```

Перехват вывода

Иногда результат шаблона не нужно выводить на экран, а требуется записать в некоторую переменную. Это необходимо при анализе XML-файлов с помощью шаблона Smarty и обработке их с использованием листов стилей XSL для отображения в Web-браузере. Полученный результат может вообще не выводиться на экран, а отправляться по электронной почте. Такое использование пакета Smarty будет описано в следующей главе.

```
//перехват вывода
$strOutput = $objSmarty->fetch("index.tpl");
```

Обработанный шаблоном результат присваивается переменной `$strOutput`, которую можно использовать по своему усмотрению.

Включение других шаблонов

Smarty позволяет включать и другие шаблоны, которые будут обрабатываться обычным образом. Эту возможность можно использовать для создания одинаковых верхних и нижних колонтитулов.

```
{include file="header.tpl"}
```

Дополнительные ресурсы

Функциональность пакета Smarty не исчерпывается материалом этой главы.

Более подробная информация об этом пакете содержится по адресу <http://smarty.php.net/manual/en/>. В этой документации достаточно полно описаны все модификаторы, процедуры и конфигурационные параметры. Документация делится на две части. Первая из них предназначена для программистов, а вторая — для разработчиков шаблонов HTML.

Когда лучше использовать пакет Smarty, а не традиционные шаблоны

Отдать предпочтение одному из подходов достаточно сложно. Хотя Smarty является мощным средством для создания шаблонов, этот пакет не позволяет решить все проблемы, возникающие в больших проектах.

Код Smarty можно расширить операторами PHP, выделяемыми парой дескрипторов `{php} {/php}`. Однако на Web-узле Smarty этого делать не рекомендуется.

Решающим фактором при выборе подхода должен стать опыт разработчиков. Если проект создается одним человеком, то лучше использовать обычные шаблоны. Однако если над проектом работает группа дизайнеров и специалистов по HTML, то шаблоны Smarty использовать надежнее, поскольку они позволяют формировать логику вывода средствами, близкими для этих специалистов и не требующими добавления кода PHP.

Резюме

В этой главе детально описан шаблон проектирования MVC и методология его применения в сложных PHP-проектах. Предложены два способа реализации этой методологии. Первый состоит в применении базового набора классов, реализующих иерархию объектов `request` и `constraint` для разделения компонентов модели, вида и контроллера.

Рассмотрено понятие шаблона и приведены примеры их реализации.

И наконец, детально описаны шаблоны Smarty. Вы узнали, как обычные шаблоны быстро преобразовать в шаблоны Smarty и воспользоваться преимуществами этого пакета в сложных проектах.

В следующей главе речь пойдет о взаимодействии с пользователями на основе объектной модели.

14

Общение с пользователями

Web — это простой способ взаимодействия и передачи сообщений. Конечный пользователь запрашивает страницу, нечто происходит, и Web-сервер передает назад определенный ответ. Это может продолжаться до бесконечности. Но если ограничиться только общением пользователей через их Web-браузер, теряются огромные возможности в использовании впечатляющего интерфейса пользователя.

В этой главе вы узнаете о строительных блоках системы общения с пользователями и их реализации в чрезвычайно естественной объектной иерархии PHP. Вы увидите, как расширять и развивать эту иерархию для поддержки практически любого вида электронного общения. Будут приведены практические примеры реализации некоторых функциональных возможностей, которые не предоставляет PHP и которые могут быть полезны в других проектах.

Но, главное, читателю станет ясно, зачем нужно общаться с пользователями.

Для чего общаться

На ранних этапах развития Web, задолго до появления PHP, опыт пользователя и побудительные мотивы его обращения к данному Web-узлу большого значения не имели. Основу составляло содержимое узла. На самом деле содержимое представляло единственный критерий посещаемости узла. Всемирная паутина представляла собой журнал с удобными ярлыками для простого перелистывания страниц.

Не более. Постепенно Web стали все чаще использовать как великолепный механизм для работы приложений на основе “тонкого клиента”, от простых приложений электронной почты для развлекательного портала до модернизации многопользовательских систем управления в сетях больших корпораций. Эти возможности являются первопричиной изучения PHP.

Однако и приложения на основе технологии “тонкого клиента” не являются последним словом техники. На сегодня эти технологии активно не развиваются, устаревают, и, по существу, являются средствами дня вчерашнего.

Одна из важнейших черт современного Web-приложения — возможность общаться с пользователем вне контекста пользовательского интерфейса. Рассмотрим абстрактный пример. Непосредственный интерфейс пользователя телефона состоит из трубки и номеронабирателя. Все телефоны также содержат звонок, сообщающий пользователю о входящих вызовах. Это пример уведомления пользователя о событии.

В качестве второго примера рассмотрим факс. Многие факсимильные аппараты печатают отчет об успешности выполненной операции. Этот отчет является средством общения пользователя с системой, дополнительным по отношению к клавиатуре и жидкокристаллическому дисплею, которые составляют пользовательский интерфейс. Отчет служит примером поддержки информирования пользователя.

И последний пример. Если вы когда-нибудь оформляли кредит по телефону, то получали разрешение на него в течение нескольких минут и без лишних проблем. Тем не менее, банк отсылает документы для подписи по зарегистрированному адресу. Без подписи на документах деньги никуда не перечисляются. Это пример использования связи с пользователем как формы обеспечения безопасности и проверки корректности данных.

Есть и другие примеры. В этой главе рассказывается о том, как реализовать описанные механизмы на основе принципов ООП.

Причины для общения с пользователями

Возьмем эти три примера и попробуем применить их для Web. Рассмотрим, в каких ситуациях приложение должно уведомлять пользователя, информировать его или проверять.

Уведомление

Существует множество ситуаций, когда приходится уведомлять пользователей без помощи Web-браузера.

Актуальность этой задачи становится особенно понятна, если вспомнить, что протокол HTTP не поддерживает состояния. Очень сложно заставить Web-браузер ожидать некоторого события, о котором необходимо проинформировать пользователя. Это можно сделать различными способами, но ни один из них нельзя считать достаточно хорошим.

Представьте себе, что приложение содержит сложную систему создания отчетов, которая позволяет пользователям запрашивать разнообразные сведения обо всех деталях какого-либо бизнес-процесса. Если в этих отчетах очень интенсивно используется язык SQL, целесообразно генерировать их в автономном режиме, т.е. независимо от использования Web-браузера. При этом пользователю не понадобится сталкиваться с долго обрабатывающимися запросами для получения расширенного отчета. Если отчет генерируется слишком долго (например, несколько минут), пользователь может устать от ожидания и остановить процесс на полпути. Обычно пользователи используют Web-браузер для оформления заказа на отчет и получают уведомление, когда отчет готов.

Рассмотрим другой пример. Представьте себе, что пользователь выполняет оплату счетов через Интернет, и Web-система настроена так, что кредитная карточка автоматически проверяется первого числа каждого месяца. Такой клиентский сервис не

очень хорош: если срок действия карточки истек в прошлом месяце, пользователь не сможет оплатить очередной счет через Web. Для решения этой проблемы используются *исключительные уведомления* — электронные сообщения пользователю о внеплановых и экстраординарных событиях (обычно неприятных).

Уведомления отсылаются, конечно, не только по электронной почте. Операторы информационных центров по всему миру влюбились в SMS (сервис коротких сообщений). Они используют этот сервис для уведомления пользователей сотовых телефонов (где бы они в это время ни находились) о проблемах с жизненно важными серверами или процессами.

Информирование

Информирование пользователей несколько отличается от уведомления. Оно выполняется не только в результате определенных событий. Наоборот, это вполне ожидаемое сообщение, присыпаемое с целью информирования пользователя о состоянии некоторого процесса. Информирование аналогично поступлению утренних газет, а уведомление — внезапному оповещению жителей о надвигающемся урагане через телевизионное шоу. Пользователь ежедневно ожидает поступления газет, но сообщение о надвигающейся буре в заливе Тампа будет для него неожиданностью.

Хорошим примером информирования пользователей является еженедельный информационный бюллетень для подписчиков Web-узла. Это может быть автоматически создаваемое сообщение, представленное в виде таблицы и содержащее выдержки из всех новостей за неделю, или материал, скомпонованный в редакционную статью. Любой из способов помогает держать пользователя в курсе событий.

Другим примером информирования являются интерактивные сообщения о состоянии заказа от Интернет-магазина. Если в течение семи дней вам не переслали заказанные книги, магазин может прислать сообщение с пояснением, когда ожидается поступление заказа. Это не уведомление. Ничего не изменилось. Это просто информационное сообщение.

Эти способы общения можно реализовать и для других методов передачи информации, таких как факс, текстовые сообщения, обычная почта и т.п. Необходимо понять, что различные средства общения, которые ранее казались вам совершенно не похожими друг на друга, на самом деле могут оказаться одинаковыми (как минимум с точки зрения архитектуры программного обеспечения).

Проверка пользователя

Проверка пользователя — это однократное общение с целью налаживания последующего взаимодействия. Значение некоторого дескриптора проверяется для того, чтобы удостовериться в его правильности и возможности его повторного использования. Чтобы удостовериться в получении пользователем нужной ему информации, ее необходимо выслать только на зарегистрированный пользователем адрес.

Приведенный выше пример с получением кредита иллюстрирует более серьезные стороны этой проблемы. Банк захочет подтвердить правильность зарегистрированного адреса, отправив по нему своего проверяющего. Многочисленные Web-узлы проверяют электронные адреса своих посетителей, чтобы гарантировать надежность последующих рыночных отношений с ними. Простым способом выполнения этой задачи является отправка активационных ссылок (обычно представляющих собой некоторый вид уникального ключа, хранящегося вместе с записью о данном пользователе в базе данных) на электронную почту пользователей. Пользователь должен щелкнуть на ссылке для подтверждения собственного членства.

Похожие методы могут использоваться для помощи пользователям, которые забыли свой пароль. Однако не всегда целесообразно сообщать забытый пароль посредством электронной почты. Пароли не стоит хранить в незашифрованном виде или передавать в виде обычного текста по электронной почте. Нерадивому пользователю можно предоставить ссылку (обычно с ограниченным временем действия) для получения временного доступа к Web-узлу и установки нового пароля.

В Европе, где общение с посетителями Web-узлов через их сотовые телефоны столь же популярно, как и общение по электронной почте, некоторые узлы присылают на сотовый телефон пользователя код активации — краткое текстовое сообщение, приглашающее пользователя зарегистрироваться. Потом пользователь должен ввести этот код активации на Web-узле, чтобы подтвердить правильность указанного им телефонного номера.

Подобный прием передачи кода посредством SMS-сообщений можно использовать как малозатратную форму аутентификации второго уровня. Аутентификация в системе обычно обеспечивается одним или несколькими из трех способов: по *личности* пользователя (по отпечаткам пальцев, снимку радужной оболочки глаза или даже точному IP-адресу), его *знаниям* (паролю) или *принадлежащим ему элементам* (ключу). Для обеспечения безопасности целесообразно использовать несколько (хотя бы два) из этих механизмов. Отправляя уникальный код на зарегистрированный мобильный телефон, вы обеспечиваете дополнительный уровень защиты помимо ввода имени пользователя и пароля.

За пределами Web-браузера

Использование альтернативных механизмов общения (особенно электронной почты) расширяет навыки пользователей и обеспечивает им дополнительный опыт работы с Web-узлом. Рассмотрим, как можно реализовать это на PHP (конечно, без использования функции `mail()`).

Вероятно, стоит отметить, что функция `mail()` в PHP имеет такое же отношение к отправке электронных сообщений из кода приложения, как телефон в игрушечном домике десятилетнего ребенка к цифровой телефонии. На серьезных Web-узлах ее практически никогда не используют.

Типы связи

Прежде чем приступить к написанию некоторого коммуникационного кода, посмотрим на эту проблему со стороны. Это даст возможность лучше разобраться в иерархии классов для обеспечения взаимодействия, которую придется реализовать самостоятельно, потому что PHP не обеспечивает стандартных средств для выполнения этих операций. Что общего в различных видах взаимодействия?

Общие свойства взаимодействия

Можно с уверенностью утверждать, что все типы взаимодействия обладают следующими общими свойствами.

- ❑ Получатели сообщения — это один или несколько человек, которые получают сообщения. Информация для обеспечения связи может существовать в форме

адресов электронной почты, факсимильных номеров, номеров мобильных телефонов, почтовых адресов, адресов почтовых ящиков, адресов приемных отделений и т.д.

- Сообщение — это реальный объект, который необходимо отправить. Сообщение может быть передано в виде обычного текста или HTML-кода (для электронной почты), растрового рисунка (для факса), 160 символов текста (для телефонного SMS-сообщения), потока аудиоинформации с частотой 11 Кгц (для автоматического сообщения по телефону) и т.д.

Свойства конкретных типов сообщений

Между разными типами связи существуют определенные различия. Перечислим свойства, характерные для определенных видов взаимодействия.

- Тема — свойство, присущее только электронным сообщениям.
- Варианты содержимого — программы электронной почты в наши дни часто наряду с HTML-кодом отправляют текстовую версию сообщения, помещая ее в один пакет MIME (подробнее об этом будет рассказано позже).
- Вложения — это дополнительные материалы, характерные для электронных и текстовых сообщений.
- Возможность создания копий и скрытых копий — при отправке факса, текстового сообщения или письма нескольким получателям каждый из них может не знать, кому еще отправлено такое же послание, если список получателей не включен в само письмо.

Информация о получателе

Информация о получателе сообщения электронной почты коренным образом отличается от сведений о получателе текстового SMS-сообщения. Данные, характеризующие получателей разных типов сообщений, приводятся в следующей таблице.

Тип взаимодействия	Информация о получателе	Формат
Электронная почта	Имя	Строка
	Адрес	Строка, соответствующая спецификации RFC822
Текстовое SMS-сообщение для сотовых телефонов	Номер	Строка, содержащая знак “плюс”, код страны и номер телефона
Факс	Номер	Строка, содержащая номер телефона
Письмо	Полное имя	Строка
	Номер дома	Целое число
	Улица	Строка
	Город	Строка
	Страна	Строка
	Индекс	Строка в соответствующем формате

Представление взаимодействия в виде иерархии классов

Иерархию классов можно представить по-разному. Хотя различные виды связи имеют много общего, существуют и фундаментальные различия между видами носителей. Примером таких различий является разное представление получателя.

Класс Recipient: быстрая проверка объектного мышления

Все сообщения имеют своих получателей (и зачастую нескольких). Поскольку данные о получателе сообщения включают не только строковую информацию, целесообразно создать класс, который содержал бы эту информацию и такие полезные методы, как `isValid()`, предназначенные для проверки корректности данных о получателе сообщения.

Сложность такой реализации очевидна. Как эффективно реализовать такой класс для большого разнообразия данных?

Можно создать единый класс `Recipient` (получатель), охватывающий все возможные сценарии. Однако такой способ не согласуется с объектно-ориентированным подходом, потому что каждому типу связи должно соответствовать множество специфических переменных и методов-членов, не относящихся к другим видам взаимодействия. Решение этой проблемы состоит в создании некоторого общего класса `Recipient`, частные случаи которого соответствуют различным видам коммуникаций. Но как связать между собой эти частные случаи с главным классом `Recipient`?

Прежде всего, откажитесь от использования традиционного наследования классов. Несмотря на то, что технически это выполнимо, — в данном случае это неверный ход мыслей. Суперклассы — это классы, расширяемые в подклассах. Форд — подкласс суперкласса Автомобиль. Класс Форд может содержать и дополнительные или перегруженные методы или поля, но он обязательно включает и некоторые характеристики автомобиля. Класс Автомобиль имеет свои собственные методы (такие, как `ехать()` и `переключать передачу()`) и поля (такие, как регистрационный номер и цвет), и они вряд ли будут перегружены в подклассах. Получатель конкретного типа сообщения (например, электронной почты) может иметь собственные поля и методы, точно так же как класс Форд, но он не наследует ни одного свойства от общего родительского суперкласса `Recipient`. По причинам, которые становятся очевидными при изучении таблицы из предыдущего раздела, универсальный получатель сообщения не может иметь никаких полезных полей или методов. Поэтому приходится отказаться от использования наследования классов.

Чтобы решить проблему, рассмотрим более общую картину. Как коммуникационный каркас будет связан с объектами получателей? Фактически можно ограничиться только двумя методами, которые будут доступны каркасу коммуникаций, — это `isValid()` и `getStringRepresentation()`. Они могут возвращать используемые строковые представления каркаса, в зависимости от контекста. Любые другие методы (например, `setHouseNumber()`) используются приложением исключительно для заполнения данных о получателе сообщения. Можно с уверенностью утверждать, что коммуникационный каркас их никогда не будет касаться.

Таким образом, задача сводится к следующему. Необходимо создать множество различных классов, представляющих получателей сообщений различных типов, и обеспечить к ним универсальный доступ со стороны внешнего каркаса других классов.

При этом необходимо обеспечить возможность позднее добавлять новые типы получателей без какой-либо модификации внешнего каркаса.

Решение состоит в создании *интерфейса* для родового класса получателя сообщения (на самом деле совсем не класса). Этот интерфейс должен содержать определение для двух ключевых методов (`isValid()` и `getStringRepresentation()`), которые должны быть реализованы в конкретных объектах получателей сообщений.

Рассмотрим следующий фрагмент кода, который обеспечивает такой родовой интерфейс. Этот код можно сохранить в файле `recipient.phpm`.

```
interface Recipient {
    public function isValid();
    public function getStringRepresentation();
}
```

Для достижения того же результата можно поддаться соблазну и реализовать абстрактный класс получателя сообщения, но это будет плохим проектным решением. Помните, что абстрактный получатель сообщения не имеет никаких полезных свойств и методов. Было бы глупо расширять его, когда необходимо лишь обеспечить общий метод доступа к коллекции не связанных друг с другом классов. Если у читателя все еще возникают сомнения в отсутствии общности различных классов получателей сообщений, обратитесь к приведенной выше таблице.

Класс `EmailRecipient`

Рассмотрим, как реализовать описанную идею. Поскольку нет необходимости воспроизводить каждый отдельный тип получателя сообщений, рассмотрим простой пример — класс `EmailRecipient`. Начнем с объявления класса, реализующего объявленный выше интерфейс `Recipient`.

```
class EmailRecipient implements Recipient {
```

Для хранения важных свойств получателя электронной почты добавим несколько переменных-членов. Как видно из таблицы, приведенной выше в этой главе, получатель электронной почты характеризуется именем и адресом.

```
private $recipient_name;
private $recipient_address;
```

Теперь нужно обеспечить реализацию каждого метода, объявленного в интерфейсе `Recipient`. Таких методов два. Первый проверяет корректность данных получателя.

```
public function isValid() {
    if (preg_match ("/[<>]\r\n{1,}/", $this->recipient_name)) {
        return(false);
    };
    if (preg_match ("^([A-Z0-9._%-]+) ([A-Z0-9._%-]+)(\.) ([A-Z0-9._%-]{2,4})$/i", $this->recipient_address)) {
        return(true);
    } else {
        return(false);
    }
}
```

Пока все ясно. В этой функции выполняется несколько проверок с использованием регулярных выражений. Сначала проверяется имя получателя сообщения, чтобы гарантировать, что оно состоит из одного или нескольких допустимых символов. Потом

адрес электронной почты сравнивается с регулярным выражением, чтобы гарантировать его соответствие формату RFC 822.

Затем необходимо обеспечить возможность получения строкового представление данных о получателе.

```
public function getStringRepresentation() {
    $strMyRepresentation = "";
    if ($this->recipient_name) {
        $strMyRepresentation .= $this->recipient_name . " ";
    };
    $strMyRepresentation .= "<" . $this->recipient_address . ">";
    return($strMyRepresentation);
}
```

Общепринятым является следующий формат адреса электронной почты: Имя Фамилия <пользователь@пример.ком>. Будем придерживаться этого формата.

В создаваемом классе осталось реализовать методы доступа — средства получения и установки различных свойств получателя электронной почты. Все эти значения будем устанавливать в конструкторе. Для других типов получателей, например адресатов обычной почты, это будет несколько громоздко.

```
public function __construct($strRecipientAddress, $strRecipientName = "") {
    $this->recipient_name = $strRecipientName;
    $this->recipient_address = $strRecipientAddress;
}

public function setRecipientName($strRecipientName) {
    $this->recipient_name = $strRecipientName;
}

public function setRecipientAddress($strRecipientAddress) {
    $this->recipient_address = $strRecipientAddress;
}

public function getRecipientName() {
    return($this->recipient_name);
}

public function getRecipientAddress() {
    return($this->recipient_address);
}
```

На этом реализация класса для описания получателя электронной почты завершена. Полученный код можно сохранить в файле `emailrecipient.php`.

Несмотря на то, что пока не создан коммуникационный каркас, полученный класс можно протестировать с помощью следующего фрагмента кода. При этом и для интерфейса `Recipient`, и для класса `EmailCommunication` необходимо использовать функцию `require()`.

```
$objEmailRecipient = new EmailRecipient("ed@example.com", "Эд");
if ($objEmailRecipient->isValid()) {
    print "Данные о получателе заданы корректно!";
    print "Текстовое представление данных о получателе имеет вид: " .
        htmlentities($objEmailRecipient->getStringRepresentation());
} else {
    print "Данные о получателе заданы некорректно!";
};
```

В результате будет выведена следующая информация.

Данные о получателе заданы корректно! Текстовое представление данных о получателе имеет вид: Эд <ed@example.com>

Если посмотреть на класс `EmailRecipient` более внимательно, то можно увидеть, что он очень прост. Для этого класса определены две закрытых переменных-члена, конструктор, а также методы доступа `setRecipientName()` и `setRecipientAddress()` (возможно, лишние) и соответствующие им методы `get`.

Поскольку этот класс реализует интерфейс `Recipient`, необходимо обеспечить два метода, описанные в этом интерфейсе. Если этого не сделать, PHP немедленно выдаст сообщение об ошибке.

В методе `getStringRepresentation()` фрагменты адреса просто склеиваются вместе, чтобы сформировать пригодное для использования RFC822-совместимое представление имени получателя и адреса электронной почты. Этую информацию удобно использовать в сеансе SMTP.

В методе `isValid()` имя получателя проверяется на предмет наличия любых несоответствующих символов, которые могут запутать сервер SMTP; а затем проверяется синтаксическая корректность адреса электронной почты с помощью регулярного выражения.

Реализация остальных классов

В предыдущем разделе был реализован класс `EmailRecipient`, инкапсулирующий понятие получателя электронной почты. Другим типам связи соответствуют другие типы получателей, которые тоже должны реализовать интерфейс `Recipient`.

Чтобы реализовать другие типы получателей, нужно выполнить похожую процедуру. Сначала необходимо описать все нужные переменные-члены, содержащие свойства данного типа получателя, а затем методы доступа `get` и `set` для чтения и записи значений этих переменных. И наконец, нужно реализовать обязательные методы, определяемые интерфейсом `Recipient`: метод проверки корректности данных и метод обеспечения строкового представления.

Класс `Communication`

Хотя получатели сообщений различны по своей природе, общий интерфейс доступа к любому объекту получателя позволяет существенно упростить коммуникационный класс `Communication`.

Объект класса `Communication` описывает свойства взаимодействия. При этом не важно, какой тип связи реализует данный объект и каков весь перечень возможных типов взаимодействия.

Класс `Communication` никогда не будет инстанцирован, но он будет расширяться с помощью классов, представляющих конкретные формы связи, например `EmailCommunication`, `SMSCommunication` и т.д.

Однако, в отличие от получателя общего вида, класс `Communication` будет иметь полезные собственные методы. Поэтому он действительно представляет собой реальный класс, а не только интерфейс. Тем не менее, один из его методов — метод `send()` — объявляется абстрактным и реализуется только в подклассах.

Подклассы класса `Communication` могут, конечно, иметь собственные методы и переменные, дополняющие свойства суперкласса. Типичными примерами таких методов являются функции управления свойствами сообщений для данного вида взаимодействия.

В частности, к таким функциям относится метод загрузки растрового рисунка (файла .BMP) в классе SMSCommunication, предназначенный для отправки логотипа оператора сотовой связи или другого изображения вместо стандартного текстового сообщения.

Рассмотрим следующий код. Обратите внимание на использование классов Collection и Collection Iterator, описанных в главах 5 и 6. Если вы хотите скомпилировать этот код, необходимо подключить соответствующие классы с помощью функции require().

```
abstract class Communication {

    protected $arRecipientCollection;
    private $strMessage;

    protected $strErrorMessage;
    protected $errorCode;

    abstract public function send();

    public function __construct() {
        $this->strMessage = "";
    }

    public function addRecipient($objRecipient, $strIdentifier = "") {
        $strRecipient = $objRecipient->getStringRepresentation();
        if (!$strIdentifier) {
            $strIdentifier = $strRecipient;
        };
        $this->arRecipientCollection->addItem($objRecipient, $strIdentifier);
    }

    public function removeRecipient($strIdentifier) {
        $this->arRecipientCollection->removeItem($strIdentifier);
    }

    protected function _setMessage($strMessage) {
        $this->strMessage = $strMessage;
    }

    protected function _getMessage() {
        return($this->strMessage);
    }

    public function getErrorMessage() {
        return($this->strErrorMessage);
    }

    public function getErrorCode() {
        return($this->errorCode);
    }
}
```

Как видно из приведенного фрагмента, коллекция получателей объявляется как защищенная переменная-член. Это означает, что доступ к элементам коллекции открыт только для подклассов класса Communication. Для внешней части приложения эти данные недоступны.

Вы, вероятно, заметили, что объекты добавляются в коллекцию получателей с помощью метода addRecipient(). При этом абсолютно не нужно знать, к какому типу относится объект и даже сама коллекция arRecipientCollection. На первый

взгляд, может показаться, что такое проектное решение станет источником ошибок во время выполнения программы. Но на самом деле здесь используется подкласс, который инициализирует коллекцию нужного типа. Если сущность, передаваемая в качестве объекта-получателя, реализует объявленный выше интерфейс и ее тип соответствует типу подкласса, никаких проблем не возникнет.

Строковое представление объекта используется в качестве ключа для коллекции. Свойство, характеризующее сообщение, остается закрытым, и доступ к нему можно получить с помощью методов `_setMessage()` и `_getMessage()`, доступных только из подкласса. Пока не стоит заботиться о сообщениях об ошибках и соответствующих переменных-членах. О них речь пойдет немного позже.

Итак, как использовать разработанный класс на практике? Очень просто. Нужно расширить этот класс, чтобы сформировать полезный подкласс. Например, можно создать класс `EmailCommunication`, предназначенный для отправки сообщений одному или нескольким объектам `EmailRecipient`.

Переписка с пользователями по электронной почте

Рассмотрим подробнее класс `EmailCommunication` (подкласс `Communication`). Сначала необходимо обеспечить, чтобы коллекция получателей, определенная для суперкласса, содержала только объекты `EmailRecipient`. Это можно сделать путем создания класса `EmailRecipientCollection`, несколько модифицирующего класс `Communication`. Благодаря этому коллекция получателей по определению будет состоять только из элементов `EmailRecipient`.

```
class EmailRecipientCollection extends Collection {
    public function addItem>EmailRecipient $obj, $key = null) {
        parent::addItem($obj, $key);
    }
}
```

Обратите внимание, что добавление этого класса не обеспечивает никаких новых функциональных возможностей. Это просто дает возможность задать тип (вид класса) добавляемых элементов (передаваемых в качестве параметра функции).

С помощью этого класса, который теперь поддерживается PHP, можно непосредственно реализовать класс `EmailCommunication`.

Создание тестовой версии

Сначала целесообразно определить реализацию класса с усеченной функциональностью для тестирования интерфейсов ООП. Если это сделать — остальную функциональность будет довольно просто реализовать. При наличии базовой функциональности можно легко проверить логику работы приложения.

Обеспечив выполнение принципов ООП и интерфейсов, можно сконцентрировать внимание на реализации; т.е. получении электронной почты.

Приступая к созданию класса, как обычно, объявим два его свойства: видимый адрес и коллекцию видимых адресов для рассылки копий документа. Об адресатах скрытых копий волноваться не следует. Так как они невидимы, их можно обрабатывать с помощью родительского класса `Communication`.

```
class EmailCommunication extends Communication {
    private $objApparentPrimaryRecipient; // Видимый адрес в поле Кому:
    private $arObjApparentSecondaryRecipients; // Видимый адрес в поле Копия:
```

Затем обеспечим возможность задания главного получателя электронной почты. Обратите внимание, что этот класс добавляет информацию к коллекции получателей родительского (унаследованного) класса, а также устанавливает главного видимого получателя (это делается только для класса EmailCommunication).

```
public function setPrimaryRecipient($objRecipient) {
    if (!($this->arRecipientCollection->exists($objRecipient-
        >getStringRepresentation())))
        parent::addRecipient($objRecipient);
    }
    $this->objApparentPrimaryRecipient = $objRecipient->__clone();
}
```

Методы для добавления и удаления получателей копий сообщения аналогичны. В них тоже пополняется коллекция родительского класса и локальный массив видимых получателей.

```
public function addCarbonRecipient($objRecipient) {
    if (!($this->arRecipientCollection->exists($objRecipient->
        >getStringRepresentation())))
        parent::addRecipient($objRecipient);
    if (!($this->arObjApparentSecondaryRecipients->exists($objRecipient->
        >getStringRepresentation())))
        $this->arObjApparentSecondaryRecipients->addItern($objRecipient,
        $objRecipient->getStringRepresentation());
    }
}

public function removeCarbonRecipient($objRecipient) {
    if ($this->arRecipientCollection->exists($objRecipient->
        >getStringRepresentation()))
        parent::removeRecipient($objRecipient);
    if ($this->arObjApparentSecondaryRecipients->exists($objRecipient->
        >getStringRepresentation()))
        $this->arObjApparentSecondaryRecipients->removeItern($objRecipient->
            >getStringRepresentation());
    }
}
```

Этот же принцип применяется для добавления и удаления получателей скрытых копий. Однако в этом случае список видимых получателей не пополняется.

```
public function addBlindRecipient($objRecipient) {
    if (!($this->arRecipientCollection->exists($objRecipient->
        >getStringRepresentation())))
        parent::addRecipient($objRecipient);
    }

public function removeBlindRecipient($objRecipient) {
    if (!($this->arRecipientCollection->exists($objRecipient->
        >getStringRepresentation())))
        parent::removeRecipient($objRecipient->getStringRepresentation());
    }
}
```

Теперь стоит обратить внимание на конструктор, который должен инициализировать коллекцию суперкласса элементами соответствующего типа, а также локальную коллекцию видимых получателей элементами того же типа (не обязательно с тем же самым содержимым). Заметим, что при завершении работы конструктора вызывается конструктор родительского класса.

```
public function __construct() {
    // Коллекция суперкласса
    $this->arRecipientCollection = new EmailRecipientCollection();
    // Локальная коллекция видимых получателей копий
    $this->arObjApparentSecondaryRecipients = new EmailRecipientCollection();
    parent::__construct();
}
```

Теперь создадим тестовый метод send, который будет использоваться для проверки корректности работы класса.

```
public function send() {
    print "РЕАЛЬНЫЕ ПОЛУЧАТЕЛИ<BR><BR>";
    foreach ($this->arRecipientCollection as $strRecipientIdentifier =>
        $objEmailRecipient) {
        print "ИМЯ: " . $objEmailRecipient->getRecipientName() . "<BR>";
        print "АДРЕС ЭЛЕКТРОННОЙ ПОЧТЫ: " . $objEmailRecipient->getRecipientAddress() . "<BR>";
    };

    print "<BR><BR>ПОЛУЧАТЕЛЬ<BR><BR>";
    print "ИМЯ: " . $this->objApparentPrimaryRecipient->getRecipientName() . "<BR>";
    print "АДРЕС ЭЛЕКТРОННОЙ ПОЧТЫ: " . $this->objApparentPrimaryRecipient->
        getRecipientAddress() . "<BR>";
    print "<BR><BR>ПОЛУЧАТЕЛИ КОПИЙ<BR><BR>";
    foreach ($this->arObjApparentSecondaryRecipients as
        $strRecipientIdentifier => $objEmailRecipient) {
        print "ИМЯ: " . $objEmailRecipient->getRecipientName() . "<BR>";
        print "АДРЕС ЭЛЕКТРОННОЙ ПОЧТЫ: " . $objEmailRecipient-
            >getRecipientAddress() . "<BR>";
    };
}
```

Рассмотрим приведенный выше код подробнее. Сначала обратите внимание на переменные-члены.

```
private $objApparentPrimaryRecipient; // Видимый адрес в поле Кому:
private $arObjApparentSecondaryRecipients; // Видимый адрес в поле Копия:
```

Эти две переменных-члена существуют в дополнение к списку главных получателей, унаследованному от суперкласса Communication. Они обеспечивают единственное представление объекта EmailRecipient, которому напрямую адресуется электронная почта (в строке Кому:), и коллекцию объектов EmailRecipients для представления списка получателей копий сообщения (задаваемых в поле Копия:). Строго говоря, эти две переменные не важны для определения получателей сообщения, поскольку эти данные также обрабатываются с помощью коллекции в суперклассе. Однако переменные-члены используются для форматирования сообщения электронной почты, чтобы “главные” получатели и получатели копий указывались в нужном месте заголовка сообщения.

В конструкторе формируются коллекции, ссылки на которые хранятся в этих двух закрытых переменных-членах, а также вызывается конструктор суперкласса.

Методы, используемые для добавления и удаления главного адресата, получателей явных и скрытых копий, работают по одинаковой схеме. Перед выполнением соответствующей операции они проверяют, существует ли в коллекции объект с заданным ключом (ключом является строковое представление объекта получателя). Это делается с помощью метода `exists()`, обеспечиваемого объектом коллекции.

Рассмотрим несколько тестовых адресов и проверим на них работоспособность созданных функций. Затем можно перейти к отправке реальных электронных сообщений. Чтобы выполнить следующий код, необходимо использовать функцию `require()` для всех классов, разработанных выше в этой главе.

```
$objEmail = new EmailCommunication;
$objEmailRecipient = new EmailRecipient("ed@example.com", "Эд");
$objEmailCCRecipient = new EmailRecipient("ted@example.com", "Тед");
$objEmailBCCRecipient = new EmailRecipient("zed@example.com", "Зед");
$objEmail->setPrimaryRecipient($objEmailRecipient);
$objEmail->addCarbonRecipient($objEmailCCRecipient);
$objEmail->addBlindRecipient($objEmailBCCRecipient);
$objEmail->send();
```

Запустите приведенный код, и вы получите результат, подобный показанному на рис. 14.1.

Все хорошо. Из фрагмента кода видно, что Эд — основной получатель этого электронного сообщения почты, Тед — получатель видимой копии, а Зед — получатель скрытой копии. При получении электронного сообщения Эд увидит свое имя в строке Кому: и имя Тэда в строке Копия:. Эд не будет знать, что Зед тоже получил это сообщение. Тед при получении почты будет видеть то же самое. Он не будет знать, что Зед является получателем скрытой копии данного сообщения. Зед, получая почту, не увидит своего имени в списке получателей, но сможет сделать вывод, что он, должно быть, является получателем скрытой копии сообщения.

Выполнение теста показывает, что код написан правильно. В списке фактических получателей отображаются все три получателя электронной почты.

Получение сообщения

Чтобы класс `EmailCommunication` функционировал должным образом, необходимо внести некоторые корректизы.

Сначала тело электронного сообщения необходимо связать с переменной-членом, отвечающей за текст сообщения. Обычный текст сообщения электронной почты состоит двух частей — заголовка и тела. Для удобства они разделяются пустой строкой. Заголовок содержит адрес отправителя, адрес получателя, тему, адреса получателей копии и некоторые другие данные.

Добавим две переменных-члена — для темы и отправителя. Отправитель может быть представлен объектом `EmailRecipient`, потому что он, как и получатель, тоже имеет имя и адрес.

```
private $objApparentSender; // Адрес отправителя
private $strSubjectLine; // Стока темы
```

Эти свойства можно задавать с помощью некоторых базовых методов класса.

Предположим, вам придется иметь дело исключительно с электронными сообщениями в текстовом формате. Добавим простой метод, позволяющий явно устанавливать свойства сообщения. Свойство сообщения аналогично телу электронного сообщения, потому что заголовки обрабатываются отдельно.

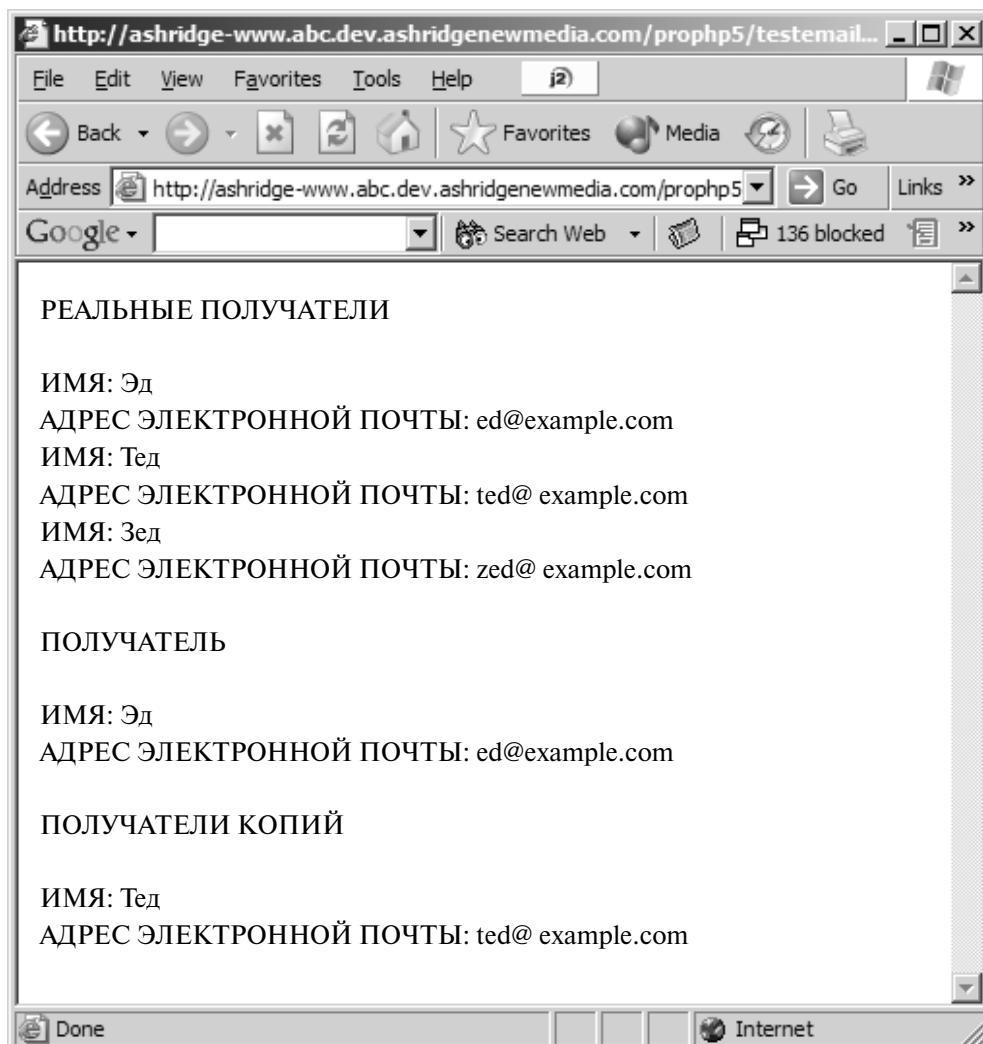


Рис. 14.1.

```
public function setSubject($strSubject) {
    $this->strSubjectLine = $strSubject;
}

public function setMessageBody($strMessageBody) {
    $this->_setMessage($strMessageBody);
}
```

```
public function setSender($objSender) {
    $this->objApparentSender = $objSender->__clone();
}
```

Теперь необходимо преобразовать метод `send()`, чтобы обеспечить возможность выбора заголовка и тела, их объединения и отправки в Интернет.

Как это сделать? Самый легкий путь — с помощью пакета Net_SMTPL PEAR, который еще раз демонстрирует преимущества модуля PEAR и позволяет избежать изобретения колеса при разработке приложений. Пакет Net_SMTPL может быть установлен при установке PHP. В противном случае установите его обычным способом.

Базовый синтаксис использования пакета Net_SMTPL довольно прост. Можно передать готовое сообщение без изменений. Для этого нужно лишь сообщить операнды, которые необходимы для общения с сервером электронной почты. Необходимый минимум данных для почтового сервера — это отправитель и список получателей. Сервер не смотрит на заголовки, поэтому можно использовать скрытые копии. Хотя сервер доставляет сообщение конкретному получателю, получатель не указывается в заголовке сообщения.

Покажем, шаг за шагом, как использовать Net_SMTPL. Будем считать, что переменная `$full_message_content` содержит готовое сообщение, а множество получателей представлено с помощью переменной `$rcpt`.

```
if (! ($smtp = new Net_SMTPL("mail"))) {
    die("Невозможно создать объект Net_SMTPL\n");
}
```

Сначала создается новый экземпляр класса Net_SMTPL с именем вашего SMTP-сервера. Если по каким-то причинам объект создать не удается, процесс завершается.

```
if (PEAR::isError($e = $smtp->connect())) {
    die($e->getMessage() . "\n");
}
```

Затем проверяется возможность подключения к SMTP-серверу и в случае неудачи процесс завершается.

```
if (PEAR::isError($smtp->mailFrom("sender@example.com"))) {
    die("Невозможно установить отправителя\n");
}
```

В данном случае отправитель определен как `sender@example.com`.

```
foreach($rcpt as $recipient) {
    if (PEAR::isError($res = $smtp->rcptTo($rcpt))) {
        die("Невозможно добавить получателя: " . $res->getMessage() . "\n");
    }
}
```

В цикле выполняется добавление каждого из получателей сообщения в экземпляр класса Net_SMTPL. Если для некоторого получателя операцию выполнить не удается — выводится сообщение об ошибке.

```
if (PEAR::isError($smtp->data($full_message_content))) {
    die("Невозможно отправить данные\n");
}
$smarty->disconnect();
```

Наконец сообщение отправлено, и соединение с SMTP-сервером разрывается.

Стоит отметить, что в этом примере в качестве узла удаленного SMTP-сервера используется mail. Если вы работаете с Linux или аналогичной операционной системой, то ссылку на это имя можно хранить в файле /etc/hosts. В противном случае необходимо указать ближайший доступный SMTP-сервер.

Теперь рассмотрим готовый класс EmailCommunication, использующий для отправки электронных сообщений класс Net_SMTPr.

```
class EmailCommunication extends Communication {

    private $objApparentSender; // Адрес отправителя
    private $strSubjectLine; // Стока темы
    private $objApparentPrimaryRecipient; // Видимый адрес в поле Кому:
    private $arObjApparentSecondaryRecipients; // Видимый адрес в поле Копия:

    public function construct() {
        // Коллекция суперкласса
        $this->arRecipientCollection = new EmailRecipientCollection();
        //Локальная коллекция видимых получателей копий
        $this->arObjApparentSecondaryRecipients = new EmailRecipientCollection();
        parent::__construct();
    }

    public function setPrimaryRecipient($objRecipient) {
        if (!($this->arRecipientCollection->exists(
            $objRecipient->getStringRepresentation())))) {
            parent::addRecipient($objRecipient);
        };
        $this->objApparentPrimaryRecipient = $objRecipient->clone();
    }

    public function addCarbonRecipient($objRecipient) {
        if (!($this->arRecipientCollection->exists(
            $objRecipient->getStringRepresentation())))) {
            parent::addRecipient($objRecipient);
        };
        if (!($this->arObjApparentSecondaryRecipients->exists(
            $objRecipient->getStringRepresentation())))) {
            $this->arObjApparentSecondaryRecipients->addItem(
                $objRecipient, $objRecipient->getStringRepresentation());
        };
    }

    public function removeCarbonRecipient($objRecipient) {
        if ($this->arRecipientCollection->exists(
            $objRecipient->getStringRepresentation())) {
            parent::removeRecipient($objRecipient);
        };
        if ($this->arObjApparentSecondaryRecipients->exists(
            $objRecipient->getStringRepresentation())) {
            $this->arObjApparentSecondaryRecipients->removeItem(
                $objRecipient->getStringRepresentation());
        };
    }

    public function addBlindRecipient($objRecipient) {
        if (!($this->arRecipientCollection->exists(
            $objRecipient->getStringRepresentation())))) {
            parent::addRecipient($objRecipient);
        };
    }
}
```

```

public function removeBlindRecipient($objRecipient) {
    if (!($this->arRecipientCollection->exists(
        $objRecipient->getStringRepresentation())))
        parent::removeRecipient($objRecipient->getStringRepresentation());
    };
}

public function setSubject($strSubject) {
    $this->strSubjectLine = $strSubject;
}

public function setMessageBody($strMessageBody) {
    $this->_setMessage($strMessageBody);
}

public function setSender($objSender) {
    $this->objApparentSender = $objSender->clone();
}

public function send() {
    // Создание заголовков
    $strHeaders .= "От: " . $this->objApparentSender->
        getStringRepresentation() . "\n";
    $strHeaders .= "Кому: " . $this->objApparentPrimaryRecipient->
        getStringRepresentation() . "\n";
    foreach ($this->arObjApparentSecondaryRecipients as
        $strRecipientIdentifier => $objEmailRecipient) {
        $strHeaders .= "Копия: " . $objEmailRecipient->getStringRepresentation() . "\n";
    };
    $strHeaders .= "Дата: " . date("D, M j H:i:s T Y O") . "\n";

    // Создание тела
    $strBody = $this->_getMessage();

    // Формирование электронного сообщения с заголовком
    $strFullEmail = $strHeaders . "\n" . $strBody;

    if (!($smtp = new Net_SMTPO("mail")))
        {
            $this->strErrorMessage = "Невозможно инстанцировать объект Net_SMTPO";
            $this->errorCode = 1;
            return(false);
        }

    if (PEAR::isError($e = $smtp->connect()))
        {
            $this->strErrorMessage = $e->getMessage();
            $this->errorCode = 2;
            $smtp->disconnect();
            return(false);
        }

    if (PEAR::isError($smtp->mailFrom(
        $this->objApparentSender->getStringRepresentation())))
        {
            $this->strErrorMessage = "Невозможно установить отправителя";
            $this->errorCode = 3;
            $smtp->disconnect();
            return(false);
        }

    // Рассылка каждому получателю
    foreach ($this->arRecipientCollection as
        $strRecipientIdentifier => $objEmailRecipient) {
        $strThisAddress = $objEmailRecipient->getRecipientAddress();

```

```

if (PEAR::isError($res = $smtp->rcptTo($strThisAddress))) {
    $this->strErrorMessage = "Невозможно добавить получателя" . $strThisAddress;
    $this->errorCode = 4;
    $smtp->disconnect();
    return(false);
};

if (PEAR::isError($smtp->data($strFullEmail))) {
    $this->strErrorMessage = "Невозможно отправить данные на сервер";
    $this->errorCode = 5;
    $smtp->disconnect();
    return(false);
}

$smtp->disconnect();
return(true);
}
};

```

Следует обратить внимание на несколько моментов. Во-первых, при добавлении получателя в список основных адресатов или получателей копии его можно добавить в коллекцию родительского суперкласса, а затем клонировать объект получателя при добавлении его к списку адресатов. Это означает необходимость реализовать собственный метод `_clone()` в классе `EmailRecipient` для модификации поведения объекта при создании этой копии.

Во-вторых, здесь используются переменные-члены суперкласса `Communication`, задающие код ошибки и сообщение об ошибке. Они доступны через открытые функции и могут использоваться для отладки приложения. Электронная почта порой ведет себя непредсказуемо, и сбой может возникнуть на любом этапе, начиная с отказа SMTP-сервера и заканчивая запретом на отправку почты для конкретного IP-адреса.

В-третьих, для описания даты отправки в заголовке электронного сообщения использован код `date("D, M j H:i:s T Y O")`. Согласно документации по PHP, такая запись определяет следующий формат даты и времени (временную метку), принятую в заголовках электронной почты `Tue, Jul 20 22:58:58 BST 2004 +0100`.

Чтобы протестировать полученный класс, нужно написать код, подобный следующему.

```

$objEmail = new EmailCommunication;
$objEmailRecipient = new EmailRecipient("ed@example.com",
"Эд");
$objEmailCCRecipient = new EmailRecipient("cc@example.com",
"Тед");
$objEmailSender = new EmailRecipient("info@example.com", "Тестовый отправитель");
$objEmail->setPrimaryRecipient($objEmailRecipient);
$objEmail->setSender($objEmailSender);
$objEmail->setMessageBody( "Привет, \n\nЭто короткое тестовое сообщение.\n\nПока !");
$objEmail->setSubject("Тестовая тема");
$objEmail->addCarbonRecipient($objEmailCCRecipient);
$objEmail->addBlindRecipient($objEmailBCCRecipient);

if ($objEmail->send()) {
    print "ГОТОВО! Все прошло хорошо! Письмо успешно отправлено.";
} else {
    print "Извините, письмо не отправлено.";
};

```

Применение шаблонов

В главе 13 речь шла о разделении уровней логики приложения и отображения с помощью применения шаблона Model–View–Controller (модель–представление–контроллер). Этот же подход следует использовать при разработке архитектуры собственного приложения.

В главе 13 также рассмотрен пакет Smarty, который помогает реализовать это разделение в традиционных Web-приложениях. Пакет Smarty имеет и другое применение, которое чаще всего не обсуждается. С его помощью можно реализовать управление электронной почтой на основе шаблона.

Обычно электронные сообщения довольно однотипны. Их можно описать с помощью нескольких дескрипторов. Например, после слова “Дорогой” следует имя получателя и т.п.

Слияние переменных в строку прекрасно подходит для небольших сообщений. Но когда электронные сообщения становятся большими и громоздкими, содержат целые массивы структур, задача усложняется. Чтобы решить эту проблему, следует использовать класс `TemplatedEmailCommunication`.

Класс `TemplatedEmailCommunication` полностью базируется на использовании пакета Smarty. Если вы не читали главу 13, сейчас самое время это сделать. Для успешного усвоения последующего материала необходимо знать, как работает Smarty.

Создаваемый класс фактически расширяет `EmailCommunication`, поэтому его необходимо подключить. Кроме того, для работы этого класса потребуется установить пакет Smarty.

```
class TemplatedEmailCommunication extends EmailCommunication {

    private $path_to_template_file;
    private $objSmarty;

    public function __construct($strPathToTemplateFile) {
        $this->objSmarty = new Smarty;
        $this->path_to_template_file = $strPathToTemplateFile;
        parent::__construct();
    }

    public function setParameter($strParameter, $strValue) {
        $this->objSmarty->assign($strParameter, $strValue);
    }

    public function parse() {
        $this->setMessageBody($this->objSmarty->fetch($this->
            path_to_template_file));
    }
}
```

Приведенный код прост, но эффективен. Теперь посмотрите на приведенный ниже код, который использует простой шаблон Smarty. Так что сначала создайте файл шаблона `test.tpl` следующим образом.

```
Name: {$name}
Favorite Food: {$favefood}
```

Теперь введите следующий код PHP. Удостоверьтесь, что он находится в той же папке, что и шаблон Smarty, либо измените первую строку, указав правильный путь. Этот фрагмент очень похож на код, который использовался для класса `EmailCommunication`. Отличия в коде выделены.

```
$objEmail = new TemplatedEmailCommunication("test.tpl");
$objEmail->setParameter("name", "Эд");
$objEmail->setParameter("favefood", "Стейк");
$objEmail->parse();

$objEmailRecipient = new EmailRecipient("ed@example.com", "Эд");
$objEmailSender = new EmailRecipient("info@example.com", "Тестовый отправитель");
$objEmail->setPrimaryRecipient($objEmailRecipient);
$objEmail->setSender($objEmailSender);
$objEmail->setSubject("Тестовая тема");

if ($objEmail->send()) {
    print "ГОТОВО! Все прошло хорошо! Письмо успешно отправлено.";
} else {
    print "Извините, письмо не отправлено.";
}
```

В результате в теле электронного сообщения появятся следующие строки.

```
Name: Эд
Favorite Food: Стейк
```

Как видите, метод `parse()` автоматически формирует тело электронного сообщения. Smarty проанализировал шаблон, задающий два параметра. Для преобразования информации в строку предпочтительнее использовать не традиционный метод `display()`, а метод `fetch()`. Затем тело сообщения связывается с этой строкой, и ее можно отправлять по электронной почте.

Обратите внимание, что расширение класса `EmailCommunication` позволяет не просто избежать изобретения велосипеда при реализации управления соединением SMTP и связью с получателем. Создание класса `TemplatedEmailCommunication` требует лишь нескольких строк кода.

Конечно, продемонстрированная здесь возможность — всего лишь малая, но достаточно убедительная часть потенциала Smarty.

Использование MIME

Несложно заметить, что в этой главе речь шла об электронных сообщениях в текстовом формате. С ними проще иметь дело, однако сегодня все чаще электронные сообщения создаются в формате HTML либо даже в виде шаблонов HTML. Возникает также вопрос относительно обработки вложений. Как их обрабатывать? Ответ на эти вопросы состоит в использовании MIME, но это — неисчерпаемая тема.

К счастью, многие проблемы использования MIME решены в пакете PEAR. Более подробную информацию можно найти по адресу http://pear.php.net/package/Mail_Mime. Предлагаемый подход заключается в расширении класса `EmailCommunication` и формировании класса `RichEmailCommunication`, содержащего дополнительные методы типа `setHTMLContent()`, `setPlainTextContent()`, `addAttachment()` и т.д.

Сейчас не время для более детального рассмотрения этих вопросов. Однако если у вас возник интерес к этой теме, вы можете интегрировать функциональность MIME в описанные выше классы. Загрузите и установите пакет PEAR с узла http://pear.php.net/package/Mail_Mime и внимательно ознакомьтесь с документацией. Создание нового важного класса `RichEmailCommunication` не составит никаких трудностей.

Детальные сведения о MIME содержатся в разделе часто задаваемых вопросов Usenet по адресу `comp.mail.mime`, в группах новостей и на различных Web-узлах, например <http://www.uni-giessen.de/faq/archiv/mail.mime-faq.part1-9/>.

Другие подклассы `Communication`

В этой главе обсуждается только электронная почта, как самая распространенная форма связи с пользователями в Web. Этим способом связи может воспользоваться каждый.

Но класс `Communication` можно расширить и на другие типы взаимодействия: факс, SMS-сообщения, голосовую связь и т.д. Как реализовать эти возможности? Вот несколько предложений.

Передача SMS-сообщений

В последние годы появилось множество провайдеров шлюзов, предлагающих за небольшую плату разослать SMS-сообщения от вашего имени. Вместо того чтобы использовать для отправки таких сообщений настоящие сотовые телефоны, используется непосредственная связь с сотовой сетью. Это экономит время и деньги.

Обычно интерфейс осуществляется по методу POST протокола HTTP. Если вас это удивляет, запустите в Web поиск по ключевой фразе "SMS Gateway". Более реальной альтернативой является подключение сотового телефона через последовательный кабель. В Интернет можно найти много детальной информации о программных интерфейсах приложений для взаимодействия с телефоном. Идея состоит в том, чтобы рассматривать эту связь как модемную. Более детальную информацию можно найти по адресу http://www.cellular.co.za/sms_at_commands.htm.

Факс

Вместо того чтобы физически создавать макет страницы факса и отсыпал его через modem, удобнее использовать один из многих уже существующих сегодня преобразователей электронных сообщений в формат факса. Большинство из них позволяет получить файл в формате TIFF, отправленный по некоторому адресу электронной почты в виде вложения, и отправить его в виде факса от вашего имени. С помощью функций PHP для работы с графикой можно сформировать TIFF-файл, отправить его по нужному адресу электронной почты и в результате получить динамически сформированное факсимильное сообщение.

Примером такой службы является eFax, подробная информация о которой содержится по адресу www.efax.com.

Резюме

В этой главе речь шла о взаимодействии с пользователями не только через Web-браузер, но и через наиболее популярный и современный тип связи — электронную почту.

Вначале мы отказались от PHP-функции `mail()` и показали, как связь с пользователями можно реализовать в виде изящной иерархии классов. Затем были реализованы элементы этой иерархии — классы `EmailRecipient` и `EmailCommunication`, — достаточно устойчивые и удобные для повседневного использования.

Затем была показана важная роль пакета Smarty в управлении электронной почтой на основе шаблонов, а также очерчены возможности создания более мощного коммуникационного каркаса.

В следующей главе будет рассказано о некоторых важнейших понятиях, необходимых для корпоративных приложений PHP, — сессиях и аутентификации.

15

Сеансы и аутентификация

Несмотря на значительный прогресс языка PHP за последние годы, в результате которого он превратился в полнофункциональную объектно-ориентированную платформу для разработки Web-приложений, он, как и все подобные языки, по-прежнему основывается на нескольких базовых принципах. К ним относятся протокол HTTP и функциональность интерфейса CGI (Common Gateway Interface – интерфейс общего шлюза) — базового строительного блока любых приложений для Web.

Протокол HTTP часто называют протоколом без поддержки состояния. При этом подразумевается, что от одного запроса пользователя другому не передается никакая информация. Запрос осуществляется с помощью метода GET или POST, в результате запроса возвращаются некоторые данные и запрос считается обработанным.

Это свойство нельзя назвать полезным при создании сложных Web-приложений, эмулирующих работу своих настольных аналогов. В таких приложениях необходимо отслеживать историю действий пользователя, а не только знать, что пользователь делает в данный момент. К счастью, язык PHP содержит средства для реализации такого подхода — поддержку сеансов. Этот прием позволяет передавать информацию о состоянии от одного запроса данного пользователя к другому, что приводит к очень полезному результату — обеспечению аутентификации и хранению состояния пользовательского запроса.

В первой части этой главы описываются встроенные средства PHP для поддержки сеансов. Читатель увидит, что несмотря на простоту существующих функций, их можно расширить и адаптировать, обеспечивая практически любую степень гибкости, включая интеграцию с базами данных от сторонних производителей. Вы также познакомитесь с базовыми принципами обеспечения безопасности сеансов, что позволит разрабатывать защищенные от хакеров архитектуры приложений.

Во второй части главы будет рассказано о том, как применить знания об управлении сеансов для разработки важного компонента — класса аутентификации на основе баз данных, который можно развернуть на любом Web-узле, требующем аутентификации пользователей, и многократно использовать.

Знакомство с сессиями

Прежде чем перейти к описанию принципов поддержки сессий в языке PHP и преимуществ такого подхода, познакомимся с понятием сеанса в целом. Постараемся ответить на следующие вопросы. Что такое сеанс? Как он работает и почему так важен?

Краткий экскурс в историю протокола HTTP

Рассмотрим, как в действительности обрабатывается запрос.

Когда Web-браузер пользователя формирует запрос, в нем помимо прочей информации содержатся следующие важные сведения для Web-сервера.

- ❑ Метод запроса — GET или POST — и номер версии протокола (1.0 или 1.1).
- ❑ Запрашиваемый документ (например, /index.php).
- ❑ Имя узла сервера, на котором содержится запрашиваемый документ. Это очень важная информация, поскольку на одном и том же сервере с одним IP-адресом зачастую хранится много Web-узлов.
- ❑ Параметры запроса (например, foo=bar, username=fred, password=letmein) в формате URL.
- ❑ Тип браузера, который зачастую называют пользовательским агентом (обычно указывается название, платформа и версия).
- ❑ Данные cookie с клиентской машины, ранее записанные сервером, к которому в настоящее время клиент делает запрос.

Если вы хотите проверить все это в действии, установите telnet-соединение с портом 80 любого Web-сервера и передайте на сервер следующую информацию.

```
GET /pub/WWW/TheProject.html HTTP/1.1
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
Host: www.w3.org
```

В ответ на запрос Web-сервер вернет требуемый документ.

Отсюда видно, что протокол HTTP действительно не поддерживает состояния: в момент запроса сервер абсолютно ничего не знает о предыдущих запросах.

Такой подход может вызывать проблемы, поскольку Web-сервер не поддерживает состояния запросов пользователя, но сам пользователь может об этом не знать. Он точно помнит, что выполнялось при загрузке последней страницы и ожидает (вполне резонно), что сервер об этом тоже должен “помнить”. Например, допустим при обращении к некоторому ресурсу требуется ввести имя пользователя и пароль. С точки зрения пользователя эту информацию логично вводить всего один раз. Приложение должно помнить ее и не запрашивать при каждом обращении пользователя к Web-узлу.

Именно так работали компьютерные настольные приложения в течение нескольких десятилетий. Для пользователя чрезвычайно важно, чтобы эта функциональность распространялась и на Web-приложения.

На заре развития Web эта функциональность обеспечивалась за счет проверки сервером удаленного IP-адреса подключаемого пользователя в базе данных. При первом обращении IP-адрес записывался в базу данных и при последующих запросах (в рамках некоторого промежутка времени) с того же IP-адреса предполагалось, что они генерируются тем же пользователем.

Этот подход был работоспособным в Интернет до появления брандмауэров, службы NAT (Network Address Translation), proxy-серверов и других подобных средств защиты от



Рис. 15.1.

вторжений. Однако современная сеть Интернет претерпела существенные изменения. Некоторые провайдеры услуг Интернет предоставляют совершенно разные внешние адреса proxy-сервера при каждом запросе пользователя к Web-узлу. В этом состоит побочный эффект применения технологий балансировки нагрузки proxy-сервера.

Еще одним стандартным решением данной проблемы, не зависящим от постоянства IP-адреса, является HTTP-аутентификация. Если вам приходилось посещать Web-узлы, предоставляющие диалоговые окна для ввода регистрационных данных (рис. 15.1), значит, вы сталкивались с HTTP-аутентификацией.

С момента своего появления на заре зарождения Web HTTP-аутентификация позволяет ограничивать доступ к некоторым файлам и каталогам Web-сервера для отдельных групп пользователей. Традиционно список пользователей, имеющих доступ к Web-ресурсу, хранился в простом текстовом файле. Однако современные средства сервера Apache позволяют записывать эту информацию напрямую на SQL-сервер.

Этот метод работает достаточно хорошо. Введя имя пользователя и пароль для доступа к данному каталогу Web-сервера, вы автоматически получаете доступ к этому каталогу или его подкаталогам при последующих запросах.

Однако такой метод редко используется при создании приложений PHP. Вряд ли имеет смысл защищать отдельные сценарии или даже каталоги. Гораздо важнее ограничить определенным образом действия пользователей. Причем эти ограничения должен распознавать и понимать интерпретатор PHP, а не обязательно сервер Apache. Подобную функциональность можно эмулировать путем репликации HTTP-заголовков и предоставления их интерпретатору PHP. Однако это не всегда желательно по следующим причинам.

- ❑ Вы практически не управляете появлением диалогового окна аутентификации, поэтому такой подход нельзя назвать дружественным для пользователя.
- ❑ Вы не можете запросить дополнительную информацию (например, задать стандартный вопрос: "Какова девичья фамилия вашей матери?").

- Нельзя сохранить никакую дополнительную информацию кроме имени пользователя и пароля.

Этот список можно продолжать и дальше. К счастью, существует третье более предпочтительное решение, которое сводится к использованию *сессий* (session).

Определение сеанса

Строго говоря, сеанс — это последовательность HTTP-запросов, сделанных в течение определенного интервала времени одним и тем же пользователем с одного и того же компьютера к одному и тому Web-приложению.

Методология поддержки сессий базируется на следующем. При первом запросе пользователя генерируется новый сеанс, а все последующие запросы рассматриваются как часть этого сеанса, если они сгенерированы в рамках заданного интервала времени (пока не истекло *время ожидания сеанса* — session timeout).

Сеанс обычно используется для определения пользователя, подключенного в данный момент к приложению. Если пользователь успешно зарегистрировался, в базу данных приложения добавляется запись с идентификатором сеанса для данного пользователя. Тогда все последующие запросы в рамках этого сеанса воспринимаются как запросы именно этого пользователя, а не какого-либо другого.

Основным элементом сеанса является его *идентификатор* (session identifier), который однозначно определяет сеанс. Каждый сеанс может существовать одновременно с сотнями сессий других пользователей. Сгенерированный и отправленный клиенту при первом запросе идентификатор сеанса (ID сеанса) должен быть уникальным и в то же время достаточно защищенным, чтобы злоумышленник не мог легко его подделать. Например, хотя последовательность целых чисел 1, 2, 3, 4 удовлетворяет требованию уникальности, она не гарантирует защищенности сеансов. Дело в том, что пользователь с сессионным номером 3 может легко изменить номер сеанса на 4 и получить доступ к сеансу другого пользователя.

Поэтому идентификаторы сессий зачастую представляют собой 32-символьные строки, состоящие из цифр и символов алфавита, а иногда формируются еще сложнее. Это зависит от встроенного в PHP средства поддержки сессий (которое будет обсуждаться далее в этой главе). При таком подходе подделка идентификатора сеанса существенно усложняется и требует специальных усилий. Ниже в этой главе будет рассказано о нескольких простых способах предотвращения таких попыток.

Сохранение сеанса

Сгенерировав идентификатор сеанса при первом запросе пользователя, необходимо обеспечить сохранность этого идентификатора для каждого последующего запроса.

Существует два способа решения этой проблемы: модификация URL и данные cookie. Рассмотрим сначала принципы реализации этих подходов, не вдаваясь в подробности работы средств поддержки сессий в PHP.

Модификация URL

Это простейший способ сохранения сеанса. При его использовании идентификатор сеанса включается в качестве параметра методов GET и POST при каждом обращении по ссылке, при передаче данных формы или при выполнении функций JavaScript.

Рассмотрим следующий пример. Предположим, PHP сгенерировал следующий идентификатор сеанса:

```
abcde1234567890abcde1234567890ab
```

который необходимо использовать при каждом последующем запросе Web-браузера. Тогда каждую ссылку в коде HTML необходимо преобразовать соответствующим образом. При этом ссылка

```
<A REF="mybasket.php">Переход</A>
```

примет вид:

```
<A REF="mybasket.php?session_id= abcde1234567890abcde1234567890ab">Переход</A>
```

Естественно, вы не станете вводить весь этот фрагмент вручную в код HTML, а захотите как-то автоматизировать процедуру добавления идентификатора сеанса “на лету”.

Придется также преобразовать параметры формы. В частности, фрагмент

```
<FORM METHOD="POST" ACTION="mybasket.php">
```

примет вид:

```
<FORM METHOD="POST" ACTION="mybasket.php">
<INPUT TYPE="HIDDEN" NAME="session_id" VALUE=
"abcde1234567890abcde1234567890ab">
```

Кроме того, понадобится модифицировать вызовы функций JavaScript. Функция `window.location.replace("index.php")`

изменится на:

```
window.location.replace("index.php?session_id=abcde1234567890abcde1234567890ab")
```

При таких модификациях некоторые запросы можно пропустить, и при их выполнении идентификатор сеанса будет потерян. Самое неприятное состоит в том, что при выполнении единственного немодифицированного запроса идентификатор будет потерян навсегда. Но это не единственный недостаток метода модификации URL.

Как быть в этом случае с закладками? При установке закладки на странице сохранится и идентификатор сеанса, который не будет действителен при следующем сеансе. Следовательно, придется вручную удалять идентификаторы при установке закладок.

Но самая большая проблема возникает при попытке скопировать ссылки и передать их друзьям или коллегам. Пользователю не придет в голову удалить ссылки на идентификатор сеанса при отправке страницы вашего Web-узла своему другу по электронной почте. В результате при попытке перехода по ссылке его друг столкнется с одной из следующих ситуаций (в зависимости от реализованного у вас уровня безопасности): либо получит полный доступ к данным исходного пользователя, либо система блокирует его действия, поскольку идентификатор сеанса не подтвержден другими данными.

Существует лучший способ сохранения сеанса — данные cookie. Но и он не лишен недостатков.

Данные cookie

Хотя метод модификации URL теоретически является самым простым способом сохранения сеанса, использование данных cookie еще проще, поскольку требует меньшего изменения кода.

Данные cookie — это небольшие фрагменты информации, отправляемые Web-браузеру Web-сервером вместе с результатами запроса. Web-браузер записывает эту информацию и предоставляет ее при последующих запросах к тому же Web-серверу.

Как и переменные, данные cookie имеют имя и значение. Некоторые из них включают также срок действия и область действия (для каких серверов они предназначены). При каждом запросе к Web-серверу браузер предоставляет имена и значения всех данных cookie, имеющих отношение к соответствующему Web-серверу. Просоченные данные автоматически удаляются Web-браузером, но и действующие данные могут при необходимости удаляться Web-сервером.

Реализация этого процесса очень проста. Как и в предыдущем примере предположим, что идентификатор сеанса имеет вид:

```
abcde1234567890abcde1234567890ab
```

При первом запросе браузера в данном сеансе этот идентификатор необходимо направить браузеру, чтобы он сообщал его при последующих запросах. Браузеру отправляются данные cookie с инструкцией сохранить значение abcde1234567890abcde1234567890ab в соответствующем именованном идентификаторе.

При jedem последующем запросе к данному Web-серверу интерпретатор PHP находит cookie с идентификатором сеанса и проверяет его срок действия с помощью некоторого внешнего набора правил или базы данных. Если действующий идентификатор сеанса отправлен как часть данных cookie от браузера, PHP считает его корректным и продолжает выполнение сценария. Если же идентификатор не предоставлен либо срок его действия истек, генерируется новый идентификатор сеанса и отправляется браузеру вместе с данными cookie.

Этот цикл продолжается от запроса к запросу в течение всего сеанса.

В качестве имени cookie необходимо выбирать осмысленное название, например `session_id`. Область действия следует ограничить своим Web-сервером (либо доменом), а срок действия установить равным максимальному промежутку времени, необходимому для выполнения всех возможных действий на вашем узле. Например, если вы считаете, что пользователь вряд ли проведет на вашем узле более получаса, срок действия идентификатора можно установить равным 30 минутам.

Данные cookie стали предметом жарких дебатов в прессе. Основные проблемы использования cookie возникали по причине слишком плохих проектных решений при создании Web-узлов и недобросовестности их операторов.

Однако несмотря на все доводы, абсолютно не имеет смысла бояться передать с помощью cookie простой идентификатор сеанса. При этом следует удостовериться, что данные cookie передаются только вашему серверу и не могут случайно быть отправлены на другие узлы для недобросовестного использования. Кроме того, на своем Web-узле следует установить четкую и последовательную политику безопасности и объяснить пользователям цель использования данных cookie и принятые меры защиты.

Однако при таком подходе возможны некоторые “подводные камни”, о которых речь пойдет в следующем разделе.

Безопасность сеанса

Действительно ли идентификатор сеанса является безопасным? Существует несколько рисков, связанных с использованием идентификатора сеанса для определения зарегистрировавшегося пользователя, однако несколько контрмер позволяют минимизировать эти риски.

Подбор идентификатора сеанса

Если недобросовестный посетитель узла каким-то образом получит корректный идентификатор сеанса, он сможет выполнять любые действия от имени владельца

сеанса. Если пользователь А использует идентификатор сеанса X, а недобросовестный пользователь Б выполняет запрос с таким же идентификатором сеанса, то Web-сервер будет считать, что пользователь Б является пользователем А и может иметь доступ к любой информации, доступной пользователю А, в том числе конфиденциальной.

Для реализации такого сценария потенциальному хакеру необходимо подобрать корректный идентификатор сеанса. Насколько это реально?

Рассмотрим 32-символьную шестнадцатеричную строку. Ее можно сгенерировать произвольным образом. Предположим, она сформирована совершенно случайно. Если каждый из 32 байт может содержать один из шестнадцати допустимых символов (от A до F и от 0 до 9), то существует 16^{32} комбинаций значений идентификатора сеанса. Потенциальному хакеру потребуется слишком много времени, чтобы перебрать все эти значения.

Если же идентификатор представлен в виде десятизначного числа в кодировке MD5 и хакер знает об этом, то ему придется просмотреть лишь 10^{10} комбинаций. Ему не составит труда написать сценарий (возможно, на языке PHP), который в цикле проверяет все идентификаторы и сообщает о проделанной работе.

Рассмотрим, что происходит при запросе пользователя на обновление персональных данных (файл `mydetails.php`). При обработке этого запроса возможны две ситуации в зависимости от корректности идентификатора сеанса.

- Если полученный идентификатор сеанса является корректным, пользователю передается информация о данных учетной записи, и он может ее изменить, в том числе пароль.
- Если идентификатор сеанса не корректен, выполняется перенаправление 302 на страницу регистрации.

Сценарий хакера должен перебрать 10^{10} комбинаций для HTTP-запроса к странице `mydetails.php`. При перенаправлении 302 он должен проигнорировать полученный результат и продолжить свою работу. При получении содержимого страницы с данными пользователя необходимо переустановить пароль на 12345 с помощью формы, зафиксировать имя пользователя и перейти к следующему идентификатору.

Если такой сценарий проработает в течение нескольких недель, то гарантированно сможет обнаружить несколько корректных идентификаторов сеанса. Этот сценарий может даже предупреждать хакера по электронной почте или с помощью SMS-сообщения о переустановке пароля для некоторого пользователя. При достаточно хорошем соединении и наличии возможности запустить несколько экземпляров сценария для нескольких серверов хакер каждую секунду будет обнаруживать несколько тысяч корректных идентификаторов.

Если хакер не знает, по какому принципу формируется идентификатор сеанса, его задача усложняется. Вряд ли стоит для генерации идентификатора использовать случайное число из заданного диапазона, даже если хакер не знает его границ. Злоумышленнику будет не сложно написать сценарий “бомбардировки” Web-узла простейшими запросами, каждый из которых генерирует новый идентификатор сеанса. По теории вероятности хакер вскоре получит полную базу данных всех возможных идентификаторов сеансов в кодировке MD5 (в предыдущем примере все 10 миллиардов). Хакер не сможет выполнить обратное преобразование этих кодов, но ему это и не понадобится. Он сможет просто воспользоваться предварительно сформированной хеш-таблицей.

Единственный способ предотвратить этот анализ — обеспечить единственность идентификатора сеанса, т.е. гарантировать, что один и тот же идентификатор никогда

не будет сгенерирован дважды по крайней мере в течение обозримого промежутка времени. Чтобы обеспечить это, идентификаторы сеансов необходимо комбинировать на основе комбинации случайного числа и текущей временной метки. Тогда сценарий получения полной базы данных идентификаторов сеанса будет бесполезен, поскольку такая база данных будет неограниченно растя.

Предотвращение подбора идентификаторов

Простой способ предотвращения подбора идентификаторов состоит в отправке второго фрагмента данных cookie, содержащего ключ к идентификатору сеанса. Этот ключ может случайным образом генерироваться в момент создания сеанса и храниться в базе данных.

При каждом запросе необходимо передавать данные cookie с идентификатором сеанса и ключом. Даже если хакеру удастся подобрать идентификатор сеанса, интерпретатор PHP проверит соответствие ключа, и в случае его несовпадения немедленно прекратит сеанс. При завершении сеанса любые попытки подбора ключа для данного идентификатора будут бесполезными. Единственным недостатком такого подхода является отключение легального пользователя при попытке подбора его идентификатора. Однако лучше обезопасить себя, чем дать возможность взломать сеанс.

При генерации случайных ключей следует избегать генераторов случайных чисел, работа которых основана на использовании системных часов. Лучше применять действительно случайные инициализаторы, такие как температура процессора, идентификаторы процессов PHP и статистические данные сетевых интерфейсов.

Перехват идентификатора

Перехват идентификатора сеанса — это менее распространенная угроза, но от нее сложнее защититься. По существу, эта угроза сводится к тому, что некоторый недоброжелатель получает доступ к данным cookie на машине легитимного пользователя и применяет их для получения доступа к сеансу.

К сожалению, в данном случае не помогает описанная в предыдущем разделе методология использования ключей. Если хакер получил доступ к данным cookie с идентификатором сеанса, то ему будут легко доступны и данные cookie с ключом.

Подобная угроза может реализоваться в следующих случаях.

- Данные cookie используются не по назначению на Web-узле и передаются другим узлам, операторы которых могут использовать эти данные для взлома сеанса на исходном узле. При этом пользователю даже не нужно посещать Web-узел злоумышленника. Миллионы электронных сообщений со спамом в формате HTML ежедневно рассылаются пользователям с единственной целью заставить получателя сформировать Web-запрос и передать данные cookie, предназначенные для другого узла.
- Доступ к данным cookie можно получить при физическом вторжении на машину, на которой они хранятся.
- Злоумышленник, получив доступ к машине, может модифицировать файл HOSTS и перенаправить трафик на другой Web-узел. При этом пользователь может ничего не подозревать, а идентификатор сеанса попадет в чужие руки.
- Если сеть плохо сконфигурирована, то недобросовестный системный администратор может перехватить идентификатор сеанса через HTTP-трафик.

- Недобросовестный сотрудник отдела информационных технологий, отвечающий за работу проху-серверов, может легко перехватить идентификаторы сеансов при регистрации пользователей на Web-узле.

Первому из этих сценариев легко противостоять. При использовании данных cookie на Web-узле необходимо обеспечить их дальнейшее “нераспространение”.

Физическому вторжению противодействовать практически невозможно. Минимизировать риск таких угроз позволяют рекомендации, приведенные в следующем разделе.

Еще одной существенной проблемой является модификация файла HOSTS, при которой подменяются имена серверов. На сегодняшний день еще не известны вирусы-черви, распространяемые по электронной почте и выполняющие эту нехитрую операцию. Но это лишь дело времени. Файл HOSTS существует почти во всех версиях системы Windows, и если администратор жестко не задаст права на использование системных файлов, пользователь сможет их модифицировать. Вирус-червь (распространяющийся в виде вложения электронного сообщения, как и большинство современных червей) может работать следующим образом.

1. Допустим, пользователь регулярно посещает узел www.myfictionalbank.com для проверки своего счета. Его IP-адрес — 10.123.123.123.
2. Файл вложения (с расширением .vbs, .scr, .rif или любым другим расширением, не заблокированным получателем электронной почты) создает запись в файле HOSTS, указывающую на IP-адрес злоумышленника. Допустим, это IP-адрес 192.168.123.123 Web-узла, расположенного где-то в Восточной Европе.
3. При следующей попытке пользователя связаться со своим банком он вводит адрес www.myfictionalbank.com и, как обычно, получает доступ к странице регистрации. На самом деле соединение осуществляется с Web-сервером хакера, который связывается с реальным сервером банка и транслирует ему все запросы пользователя. Пользователю передаются все ответы банка. Единственные данные, которые не отправляются пользователю, — это идентификатор сеанса, используемый злоумышленником для осуществления своих замыслов. Через несколько минут несколько тысяч долларов, честно заработанных пользователем, окажутся где-то в Восточной Европе.

Конечно, подобное безобразие долго продолжаться не будет. Банк обнаружит аферу и блокирует IP-адрес хакера (если вирус-червь не модифицирует себя и не обновит центральную базу IP-адресов, с которыми он связан). Однако это хорошая иллюстрация возможной угрозы.

И наконец, напомним, что многие провайдеры услуг Интернет используют прозрачные проху-серверы, т.е. весь HTTP-трафик передается через стандартный проху-сервер без явного уведомления пользователей (и соответствующей настройки их компьютеров). Речь идет о повышении производительности за счет кеширования часто посещаемых страниц у провайдера, так что пользователь чаще всего общается только с провайдером. Кроме того, провайдеров зачастую вынуждают вести очень детальные журналы действий пользователей Web. Это делается не с коммерческой целью, а в целях безопасности. Возможности недобросовестного использования этих данных совершенно очевидны. Если наряду с другой информацией провайдеры регистрируют и данные cookie, то системные администраторы могут воспользоваться ими в своих интересах.

Правильный подход к организации сеансов

Как вы уже поняли, физическому получению идентификатора сеанса (а также его ключа) очень сложно противостоять. Наиболее эффективными средствами противодействия таким атакам являются физические, политические или экономические меры. Они никак не связаны с языком PHP.

Однако существует несколько моментов, с помощью которых можно минимизировать риск практического применения вышеперечисленных теоретических возможностей.

Использование временных интервалов

В первую очередь следует использовать временные интервалы ожидания. Это не то же самое, что время истечения сеанса. При работе с данными cookie практически всегда устанавливается время истечения срока их действия. Однако нельзя гарантировать, что пользователь не уйдет пить кофе и не забудет завершить сеанс. Использование временных интервалов ожидания предполагает регистрацию временной метки каждого запроса и оценивание времени между последующими запросами в течение сеанса. При истечении времени ожидания (5 или 10 минут) строго рекомендуется прерывать сеанс и требовать повторной регистрации пользователя.

Это достаточно эффективная мера. Многие из описанных выше атак предполагают использование идентификатора сеанса сразу же после его завершения. Уменьшая интервал времени между запросами до нескольких минут, вы тем самым снижаете эффективность этих атак.

Установка интервалов ожидания никак не связана с данными cookie. Оценивание интервала между запросами выполняется в коде обработки сеанса. В следующем разделе этой главы будет описан класс `UserSession`, позволяющий реализовать эту идею.

Использование времени истечения срока

Снижение времени истечения срока для данных cookie тоже имеет смысл. Время истечения целесообразно задавать в пределах одного часа. При использовании этого подхода нужно обеспечить минимальное неудобство для пользователя при работе с вашим приложением. Например, при создании приложения электронной почты не следует прерывать сеанс пользователя в процессе отправки электронных сообщений. Постарайтесь спроектировать приложение с учетом этих особенностей. В следующем разделе данной главы будет рассказано о том, как эффективно реализовать краткое время истечения срока.

Проверка пользовательского агента

Можно использовать простую проверку, предполагающую создание агента пользователя для данного сеанса работы в Web. Это строка, определяющая производителя Web-браузера, его имя, версию и платформу. При этом нельзя гарантировать уникальность подобной строки для каждого компьютера. Однако в мире существует так много различных браузеров и их версий, что с высокой вероятностью различные компьютеры генерируют различные строки агента пользователя. Проверяя при последующих запросах соответствие данных агента пользователя, вы обеспечите дополнительную линию защиты против взлома сеансов. Интересно, что строка агента пользователя, генерируемая браузером Internet Explorer, может быть модифицирована в реестре Windows. Системные администраторы могут воспользоваться этим фактом и создать характерные для рабочих станций их сети строки агентов пользователей. Это несколько усилит данный механизм обеспечения безопасности.

Не бойтесь запросить повторную регистрацию

При выполнении операций электронной торговли или модификации персональных данных не стесняйтесь повторно запросить регистрационные данные: имя пользователя и пароль. Это может несколько снизить ущерб, наносимый хакером, получившим доступ к идентификатору сеанса. Многие большие коммерческие узлы используют эту практику, и их пользователи не в обиде.

Отслеживайте необычный трафик

Если вы себя чувствуете достаточно уверенно при разработке Web-приложений, то можно разработать некоторый алгоритм отслеживания необычного трафика в пределах конкретного сеанса с целью определения вмешательства в его ход. Как определить необычный трафик — открытый вопрос. Например, можно бить тревогу, если в течение одной-двух секунд поступает несколько запросов. Кроме того, можно отслеживать физический путь запроса пользователя и удостовериться в том, что для разных запросов он повторяется. Например, выбрав категорию товара на узле электронной коммерции, пользователь вряд ли сразу же выберет страницу товара, не попадающего в эту категорию. Для генерации такого запроса пользователь должен иметь соответствующую закладку или напрямую ввести адрес URL. Даже если ваш алгоритм в некоторых случаях объявит ложную тревогу, пользователь вряд ли будет в обиде.

Отслеживайте необычные вариации IP-адреса

Хотя корректность идентификатора сеанса нельзя с полной уверенностью определить на основе IP-адреса, его вариации иногда позволяют определить некоторые попытки взлома. Роху-серверы в процессе балансировки нагрузки несколько изменяют IP-адреса, но насколько радикально? Вряд ли изменятся данные за пределами последних двух октетов. Естественно, запрос останется в пределах того же сегмента сети (т.е. сетевой сегмент будет неизменным для всех запросов в рамках сеанса). Для проверки этого можно в реальном времени проконсультироваться в глобальных базах данных RIPE или ARIN.

При разработке приложения можно предусмотреть несложные средства проверки владельцев сетевого сегмента, основанные на технологии кэширования. Если запросы в рамках одного сеанса поступают от нескольких владельцев, можно смело бить в колокола и прерывать сеанс.

Избегайте хранения переменных сеанса на стороне клиента

Очевидным преимуществом использования идентификаторов сеанса является возможность их связывания с соответствующими переменными. В качестве этих переменных может выступать содержимое корзины или формы поиска.

Однако следует избегать хранения этой информации в клиентской части приложения, поскольку в этом случае их можно бесконтрольно модифицировать (злонамеренно или случайно).

Передавать эти данные в каждом HTTP-запросе весьма неэффективно. Напомним, что большинство соединений является асимметричными, т.е. загрузка данных выполняется быстрее, чем их отправка. То есть, если переменные сеанса занимают 16 Кбайт и хранятся на стороне клиента, то выполнение каждого запроса потребует на секунду больше времени. Может это и не так много, но пользователей это может раздражать.

В клиентской части приложения необходимо хранить только идентификатор сеанса (и при необходимости дополнительный ключ). Всю остальную информацию нужно хранить на сервере, вдали от хакеров, и находить соответствующие переменные по идентификатору сеанса.

Из этой части главы становится понятно, что 100%-ную защиту приложения обеспечить невозможно. Однако существует несколько шагов, позволяющих минимизировать риск вторжения.

Реализация сеансов в PHP

Язык PHP позволяет реализовать эффективную инфраструктуру управления сессиями, однако значительную часть работы придется выполнить самостоятельно.

Как станет ясно из следующего раздела, встроенные в PHP средства обработки сеансов не подходят для корпоративных приложений. К счастью, их можно расширить при создании своего собственного более гибкого решения.

Сеансы в PHP

Рассмотрим следующий код. Введите его и сохраните в файле `firstpage.php`.

```
<?php
    session_start();
    $_SESSION['favorite_artist'] = 'Слава Вакарчук';

    ?>Мой любимый певец - Слава Вакарчук. К слову, мой идентификатор
    для сеанса в данном браузере -
    <?=session_id()?>.
    <BR><BR>
    <A HREF="secondpage.php">Перейти на вторую страницу</A>
?>
```

Теперь введите еще один фрагмент кода и сохраните его в файле `secondpage.php`.

```
<?php
    session_start();

    ?>В результате проверки оказалось, что мой любимый певец -
    <?=$_SESSION['favorite_artist']?>.
    В данный момент мой идентификатор для данного сеанса
    <?=session_id()?>.
    <BR><BR>
    <A HREF="firstpage.php">Перейти на первую страницу</A>
?>
```

Запустите первую страницу и вы получите результат, представленный на рис. 15.2. Из приведенной страницы видно, что вашим любимым певцом является Слава Вакарчук, и PHP присвоил вам идентификатор сеанса.

Идентификатор сеанса состоит из 32 символов. Для его генерации используется весь латинский алфавит и цифры от 0 до 9. Следовательно, количество возможных идентификаторов сеансов составляет 36^{32} . Если щелкнуть на ссылке, то вы увидите страницу, показанную на рис. 15.3.

Следует отметить два важных момента. Идентификатор сеанса остался прежним, и создается впечатление, что сценарий запомнил ваши музыкальные предпочтения.

Этот результат получен довольно просто. Во-первых, перед началом кода HTML вызывается функция начала сеанса `session_start()`. Это очень важно, поскольку передаваемые Web-браузеру данные сеансов необходимо поместить в заголовок HTTP, а если в начале файла содержится код HTML (или даже пробел), то этого сделать нельзя.

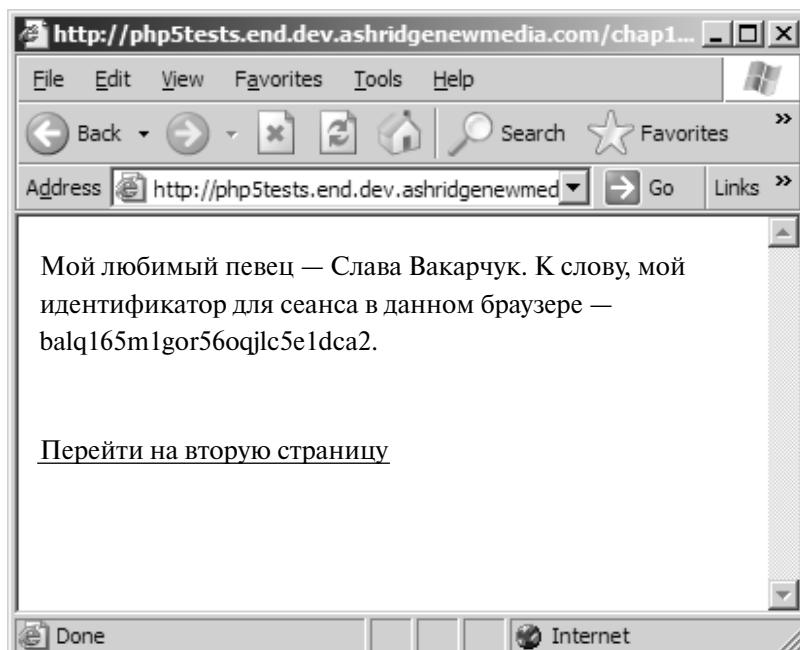


Рис. 15.2.

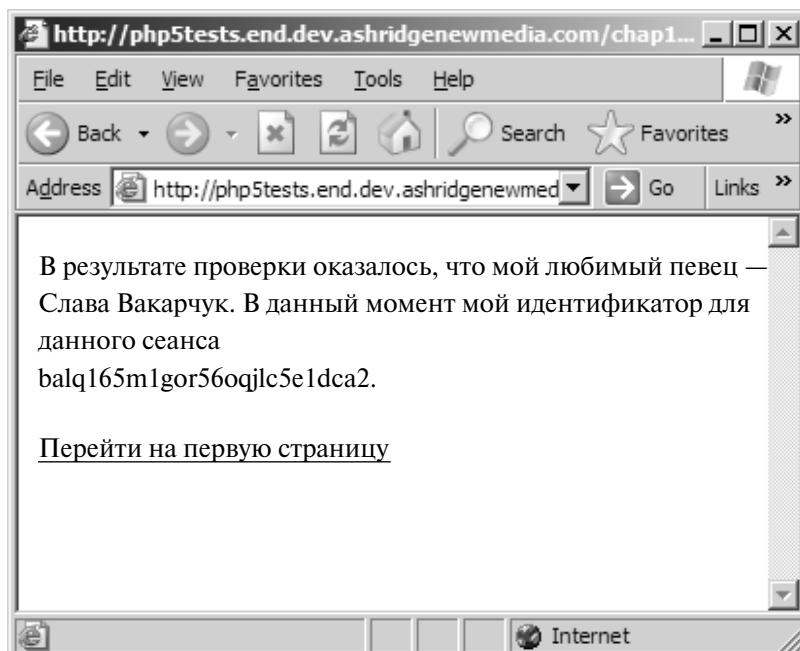


Рис. 15.3.

Во-вторых, переменная сеанса регистрируется путем добавления в глобальный ассоциативный массив `$_SESSION`. Этую операцию можно выполнить в любом месте сценария, поскольку она ничего не отправляет Web-браузеру, а лишь запоминает информацию в серверной части приложения.

На второй странице, как и на первой, вызывается функция `session_start()`, однако теперь из массива `$_SESSION` можно считать данные, записанные в него на первой странице. Таким образом, имя любимого певца перекочевало с первой страницы на вторую. Это значение было связано с идентификатором сеанса, поэтому сервер его и запомнил.

Но как PHP запомнил сам идентификатор?

Загляните в каталог `/tmp` на сервере. Просмотрите список файлов каталога с помощью команды `ls` и найдите все файлы, имя которых начинается с символов `sess_`. Обратите внимание на файл, в имени которого после символов `_sess` указан идентификатор, приведенный на первой странице. Откройте его с помощью любого редактора, и вы увидите картину, показанную на рис. 15.4.

Содержание этого файла вполне понятно. В нем в текстовом формате записана созданная переменная сеанса и ее значение.

```
favorite_artist|s:9:"Слава Вакарчук";
```

Этот формат выглядит знакомым. Он представляет собой строковое представление структуры данных PHP, возвращаемое функцией `serialize()`.

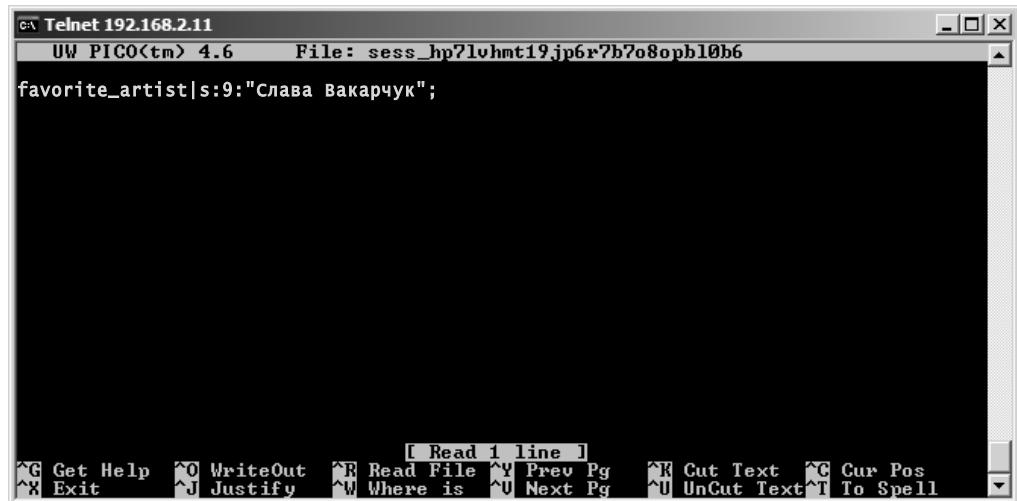


Рис. 15.4.

Независимо от сложности и величины структуры данных, сохраненной в переменной `$_SESSION`, ее содержимое будет помещено в этот файл по умолчанию. Местоположение подобных файлов, а также время их хранения задаются в файле `php.ini`.

Вот и все, что касается организации сеансов в PHP. Теперь рассмотрим некоторые их ограничения и определим, почему встроенные возможности обработки сеансов нельзя использовать в корпоративной среде.

Ограничения базовых сеансов PHP

При подобном подходе к реализации сеансов возникает несколько проблем, связанных со способом хранения данных сеанса на диске.

Предположим, запросы обрабатываются несколькими Web-серверами, для которых выполняется балансировка нагрузки. Тогда для хранения переменных сеансов придется организовать совместно используемый каталог `/tmp` на основе файловой системы NFS. Такое решение будет работать слишком медленно.

Кроме того, если вы используете Web-сервер хостинговой компании, содержимое каталога `/tmp` будет доступно и другим пользователям (даже если сами файлы будут для них закрыты). Поскольку идентификатор сеанса PHP указывается в имени файла, недобросовестный пользователь Web-сервера может легко получить полный список всех идентификаторов с указанием времени их создания. Это откроет ему двери к реализации описанных выше атак.

Далее, обработка сеансов в PHP выполняется не очень эффективно. Кроме того, в нем не реализована ни одна из описанных выше дополнительных мер обеспечения безопасности. Поэтому для коммерческих Web-узлов встроенные средства обработки сеансов PHP непригодны.

И наконец, если остальные данные приложения хранятся в базе данных, разработчику придется формировать запросы, которые должны обращаться как к таблице базы данных, так и к переменным сеанса. При этом придется извлекать значения переменных сеанса из файла с помощью оператора PHP, а затем встраивать их в операторы SQL. Очевидно, это ведет к снижению производительности приложения, и для корпоративных систем может стать серьезной проблемой. Ее решение состоит в интегрировании управления сеансами с базами данных.

Создание класса аутентификации

В заключительной части этой главы вы узнаете, как технологию обработки сеансов PHP интегрировать с базой данных PostgreSQL и обеспечить хранение данных сеансов в базе, а не просто на диске. При этом будет создан класс, обеспечивающий поддержку аутентификации пользователя, который можно повторно использовать в любом проекте.

Подключение механизма управления сеансами к базе данных

Использование базы данных для хранения переменных сеанса вместо обычного текстового файла на диске сервера обеспечить существенно проще, чем может показаться на первый взгляд. Следующие примеры рассчитаны на использование СУБД PostgreSQL, но предлагаемый метод легко адаптировать для работы с MySQL, XML или любой другой базой данных.

Для реализации подхода придется воспользоваться лишь одной функцией PHP `session_set_save_handler()`. Если обратиться к справочному руководству, то синтаксис этой функции покажется достаточно простым.

```
bool session_set_save_handler(string open, string close,
string read, string write, string destroy, string gc)
```

Идея использования этой функции довольно проста. Ее нужно вызвать *перед* функцией `session_start()` на каждой странице, связанной с обработкой сеансов. Эта функция указывает интерпретатору PHP, какие пользовательские функции необходимо

вызывать для обработки данного сеанса, т.е. для инициализации сеанса, его завершения, чтения, записи и прерывания. Параметры этих методов, а также возвращаемые ими значения должны удовлетворять определенным требованиям, которые детально описаны в справочном руководстве по функции `session_set_save_handler()`. Тем не менее в следующих разделах эти функции будут кратко рассмотрены.

Функция `open()`

Функция `open()` вызывается при вызове функции `session_start()`. Ей передаются два значения: путь, по которому должны храниться переменные сеанса при сохранении на диске (в рассматриваемом подходе этот параметр не понадобится), и имя файла cookie. В случае успешного создания сеанса функция возвращает значение `true`.

Функция `close()`

Эта функция (не путайте с `destroy()`) вызывается для эффективного завершения каждого сценария PHP, связанного с обработкой сеансов. Нам важно знать, что она возвращает значение `true`.

Функция `read()`

Эта функция используется при попытке извлечения переменной из массива `$_SESSION`. Ей в качестве единственного параметра передается идентификатор сеанса, а возвращается сериализованное представление массива `$_SESSION`. Эту функцию мы не будем использовать в разрабатываемом классе, поскольку реализуем собственную обработку переменных сеанса.

Функция `write()`

Эта функция вызывается при попытке модификации массива `$_SESSION`. Ей передается идентификатор сеанса, а также сериализованное представление массива `$_SESSION`. Функция возвращает значение `true`, если данные успешно обработаны. Эта функция вызывается даже в том случае, если переменные сеанса не зарегистрированы, и при первом вызове записывает генерированный идентификатор сеанса.

Функция `destroy()`

Эта функция вызывается при вызове в коде функции `session_destroy()`. При успешном выполнении она возвращает значение `true`.

Функция `gc()`

Эта функция сборки мусора в качестве параметра получает максимальное время жизни данных cookie для текущего сеанса и удаляет любые данные, срок жизни которых превышает заданный. При завершении работы функция возвращает значение `true`. Она должна вызываться перед функцией `open()` для удаления всех просроченных данных сеансов.

Знакомство с классом `UserSession`

Класс `UserSession` — удобный способ реализации объектно-ориентированного подхода для управления сеансами, а также обеспечения методов аутентификации в приложении. Мы реализуем собственные методы, заменяющие встроенные методы PHP, с использованием описанной выше функции `session_set_save_handler()`.

Это будет вполне самодостаточный класс, скрывающий от остальной части приложения все функции PHP `session_` для работы с сессиями.

Разрабатываемый класс также обеспечит обработку переменных сессии. В отличие от хранения всех переменных в виде единой сериализованной строки (как это реализовано в PHP), в создаваемом классе каждая переменная будет храниться в отдельной строке таблицы. Это существенно ускорит доступ к данным. Заметим, однако, что описанный выше метод обработки сессий не предназначен для работы с функциями-членами классов, поэтому нам придется потрудиться.

Схема базы данных

Работа класса будет связана с тремя таблицами. Приведем код SQL (в духе PostgreSQL) для создания этих таблиц. Первым делом необходимо создать новую базу данных с этими таблицами.

Таблицу `user` необходимо настроить в соответствии со своими требованиями. Возможно, в ней придется хранить не только имя и фамилию пользователя. В эту таблицу включен также столбец `last_impression`, предназначенный для хранения времени и даты последнего запроса к данной странице в течение сеанса. Данные этого столбца будут использованы для вычисления интервалов времени ожидания.

```
CREATE TABLE user_session (
    "id" SERIAL PRIMARY KEY NOT NULL,
    "ascii_session_id" character varying(32),
    "logged_in" bool,
    "user_id" int4,
    "last_impression" timestamp,
    "created" timestamp,
    "user_agent" character varying(256)
);

CREATE TABLE "user" (
    "id" SERIAL PRIMARY KEY NOT NULL,
    "username" character varying(32),
    "md5_pw" character varying(32),
    "first_name" character varying(64),
    "last_name" character varying(64)
);

CREATE TABLE "session_variable" (
    "id" SERIAL PRIMARY KEY NOT NULL,
    "session_id" int4,
    "variable_name" character varying(64),
    "variable_value" text
);
```

Из приведенного фрагмента видно, что данные сеанса индексированы по стандартному числовому идентификатору, а не по генерированному PHP идентификатору сеанса. Это существенно ускоряет поиск переменных сеанса (числа всегда лучше использовать в качестве индекса, чем строки).

Теперь целесообразно создать тестового пользователя (например, с именем `ed` и паролем `12345`). Для записи этого пароля в базу данных необходимо знать его представление в кодировке MD5. Конечно, в реальной жизни для выполнения такого преобразования придется использовать специальную процедуру, а пока воспользуемся следующим оператором SQL.

```
INSERT INTO "user" (username,md5_pw,first_name,last_name)
VALUES ('ed','827ccb0eea8a706c4c34a16891f84e7b','Эд','Леки-Томпсон');
```

В первую очередь рассмотрим полный код класса `usersession.phpm`. Сразу же после кода класса будут подробно рассмотрены все его составляющие, включая вызов функции `session_set_save_handler()`.

Код класса UserSession.phpm

Напомним, что расширение `.phpm` используется для явного указания того, что в файле хранится класс, а не шаблон или выполняемый сценарий.

```
<?php
class UserSession {
    private $php_session_id;
    private $native_session_id;
    private $dbhandle;
    private $logged_in;
    private $user_id;
    private $session_timeout = 600;    # 10-минутный тайм-аут
    private $session_lifespan = 3600; # длительность сеанса 1 час

    public function __construct() {
        # Соединение с базой данных
        $this->dbhandle = pg_connect("host=db dbname=prophp5 user=ed") or
            die ("Ошибка PostgreSQL: --> " . pg_last_error($this->dbhandle));
        # Установка обработчика
        session_set_save_handler(
            array(&$this, '_session_open_method'),
            array(&$this, '_session_close_method'),
            array(&$this, '_session_read_method'),
            array(&$this, '_session_write_method'),
            array(&$this, '_session_destroy_method'),
            array(&$this, '_session_gc_method')
        );
        # Проверка cookie: переданы ли эти данные и корректны ли они
        $strUserAgent = $GLOBALS["HTTP_USER_AGENT"];
        if ($_COOKIE["PHPSESSID"]) {
            # Проверка сроков давности для безопасности
            $this->php_session_id = $_COOKIE["PHPSESSID"];
            $stmt = "select id from \"user_session\" where ascii_session_id = '" .
                $this->php_session_id . "' AND ((now() - created) < '" . $this-
                >session_lifespan . " seconds') AND user_agent='". $strUserAgent . "' AND
                ((now() - last_impression) <= '".$this->session_timeout." seconds' OR
                last_impression IS NULL)";
            $result = pg_query($stmt);
            if (pg_num_rows($result)==0) {
                # Установка флага failed
                $failed = 1;
                # Удаление из базы данных - периодическая сборка мусора
                $result = pg_query("DELETE FROM \"user_session\" WHERE
                    (ascii_session_id = '". $this->php_session_id . "') OR (now() - created) >
                    $maxlifetime)");
                # Очистка переменных сеанса
                $result = pg_query("DELETE FROM \"session_variable\" WHERE session_id
                    NOT IN (SELECT id FROM \"user_session\")");
                # Удаление идентификатора... получение нового
                unset($_COOKIE["PHPSESSID"]);
            };
        };
        # Установка времени жизни данных cookie
        session_set_cookie_params($this->session_lifespan);
        # Вызов метода session_start
        session_start();
    }
}
```

```

public function Impress() {
    if ($this->native_session_id) {
        $result = pg_query("UPDATE \"user_session\" SET last_impression =
now() WHERE id = " . $this->native_session_id);
    }
}

public function IsLoggedIn() {
    return($this->logged_in);
}

public function GetUserID() {
    if ($this->logged_in) {
        return($this->user_id);
    } else {
        return(false);
    };
}

public function GetUserObject() {
    if ($this->logged_in) {
        if (class_exists("user")) {
            $objUser = new User($this->user_id);
            return($objUser);
        } else {
            return(false);
        };
    };
}

public function GetSessionIdentifier() {
    return($this->php_session_id);
}

public function Login($strUsername, $strPlainPassword) {
    $strMD5Password = md5($strPlainPassword);
    $stmt = "select id FROM \"user\" WHERE username = '$strUsername' AND
md5_pw = '$strMD5Password'";
    $result = pg_query($stmt);
    if (pg_num_rows($result)>0) {
        $row = pg_fetch_array($result);
        $this->user_id = $row["id"];
        $this->logged_in = true;
        $result = pg_query("UPDATE \"user_session\" SET logged_in = true,
user_id = " . $this->user_id . " WHERE id = " . $this->native_session_id);
        return(true);
    } else {
        return(false);
    };
}

public function LogOut() {
    if ($this->logged_in == true) {
        $result = pg_query("UPDATE \"user_session\" SET logged_in = false,
user_id = 0 WHERE id = " . $this->native_session_id);
        $this->logged_in = false;
        $this->user_id = 0;
        return(true);
    } else {
        return(false);
    };
}

```

```

public function __get($nm) {
    $result = pg_query("SELECT variable_value FROM session_variable WHERE
session_id = " . $this->native_session_id . " AND variable_name = '" . $nm . "'");
    if (pg_num_rows($result)>0) {
        $row = pg_fetch_array($result);
        return(unserialize($row["variable_value"]));
    } else {
        return(false);
    }
}

public function __set($nm, $val) {
    $strSer = serialize($val);
    $stmt = "INSERT INTO session_variable(session_id, variable_name,
variable_value) VALUES(" . $this->native_session_id . ", '$nm', '$strSer')";
    $result = pg_query($stmt);
}

private function _session_open_method($save_path, $session_name) {
    # Ничего не делается
    return(true);
}

private function _session_close_method() {
    pg_close($this->dbhandle);
    return(true);
}

private function _session_read_method($id) {
    # Используется для проверки существования сеанса
    $strUserAgent = $GLOBALS["HTTP_USER_AGENT"];
    $this->php_session_id = $id;
    # Установка флага failed
    $failed = 1;
    # проверка наличия в базе данных
    $result = pg_query("select id, logged_in, user_id from \"user_session\""
where ascii_session_id = '$id'");
    if (pg_num_rows($result)>0) {
        $row = pg_fetch_array($result);
        $this->native_session_id = $row["id"];
        if ($row["logged_in"]=="t") {
            $this->logged_in = true;
            $this->user_id = $row["user_id"];
        } else {
            $this->logged_in = false;
        };
    } else {
        $this->logged_in = false;
        # Необходимо создать запись в базе данных
        $result = pg_query("INSERT INTO user_session(ascii_session_id, logged_in,
user_id, created, user_agent) VALUES ('$id','f',0,now(),'$strUserAgent')");
        # Получение истинного идентификатора
        $result = pg_query("select id from \"user_session\" where
ascii_session_id = '$id'");
        $row = pg_fetch_array($result);
        $this->native_session_id = $row["id"];
    };
    # Возвращаем пустую строку
    return("");
}

private function _session_write_method($id, $sess_data) {
    return(true);
}

```

```

    }

private function _session_destroy_method($id) {
    $result = pg_query("DELETE FROM \"user_session\" WHERE ascii_session_id = '$id'");
    return($result);
}

private function _session_gc_method($maxlifetime) {
    return(true);
}

}

?>

```

Код модульного теста для класса UserSession

Прежде чем рассмотреть этот код, протестируем созданный класс. Для проверки его работы можно воспользоваться следующим простым сценарием.

```

<?php
require_once("usersession.phpm");

$objSession = new UserSession();
$objSession->Impress();

?>
Страница тестирования класса UserSession
<HR>
<B>Текущий идентификатор сеанса: </B> <?=$objSession->GetSessionIdentifier();?><BR>
<B>Пользователь зарегистрирован? </B> <?=(($objSession->IsLoggedIn() == true)
? "Да" : "Нет")?><BR>
<BR><BR>
Попытка регистрации ...
<?php $objSession->Login("ed", "12345"); ?>
<BR><BR>
<B>Пользователь зарегистрирован? </B> <?=(($objSession->IsLoggedIn() == true)
? "Да" : "Нет")?><BR>
<B>Идентификатор зарегистрированного пользователя: </B> <?=$objSession-
>GetUserID();?><BR>

<BR><BR>
Отключение...
<?php $objSession->Logout(); ?>

<BR><BR>
<B>Logged in? </B> <?=(($objSession->IsLoggedIn() == true) ? "Да" : "Нет")?><BR>
<BR><BR>

```

Запустите его, и вы получите результат, показанный на рис. 15.5.

Если щелкнуть на кнопке Refresh (Обновить) несколько раз, будет выведено одно и то же время и идентификатор сеанса.

Если вы хотите удостовериться, что состояние данного сеанса действительно сохраняется, закомментируйте строку отключения и щелкните на кнопке Обновить. Тогда в верхней части страницы будет указано, что пользователь в данный момент зарегистрирован.

Стоит также посмотреть на базу данных. Если пользователь зарегистрировался успешно, для данного сеанса в таблице базы данных устанавливается флагок, который остается там до тех пор, пока сеанс не удалится или не будет вызван метод LogOut().

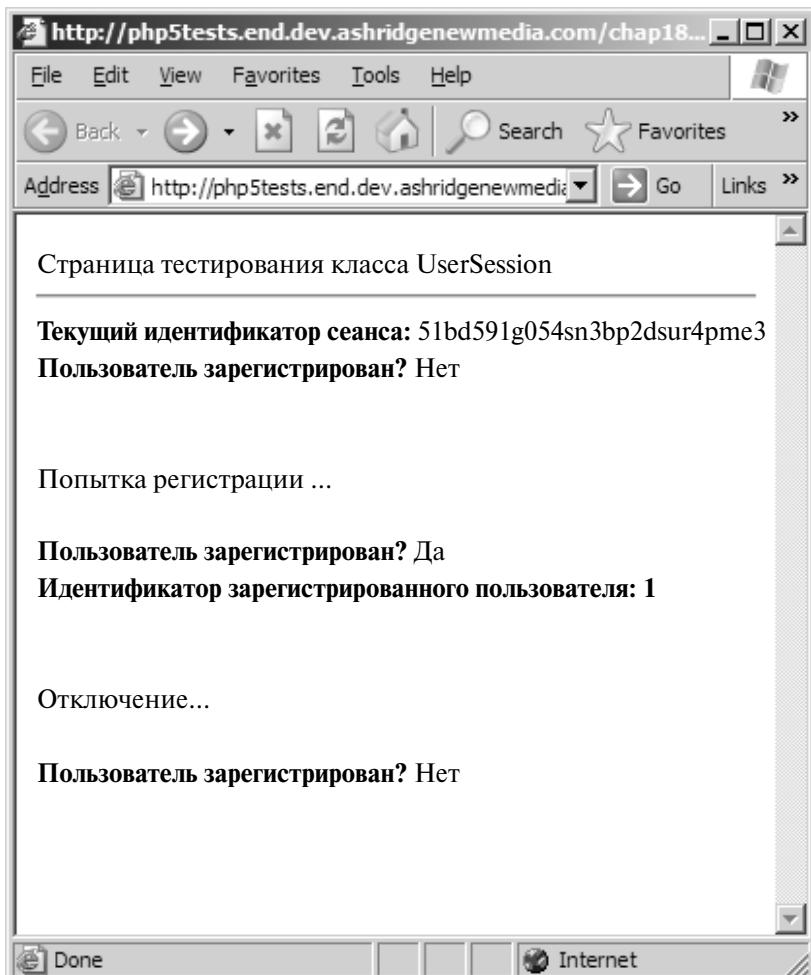


Рис. 15.5.

```
prophp5=# SELECT * FROM user_session ;
id      | ascii_session_id | logged_in | user_id| last_impression
         | created          |           |        | user_agent
-----+-----+-----+-----+
-----+-----+-----+
168 | 51bd591g054sn3bp2dsur4pme3 | f       |      0 | 2004-02-23 07:31:04.33
4694 | 2004-02-23 06:54:31.802746 | Mozilla/4.0 | (compatible; MSIE 6.0;
Windows NT 5.1; .NET CLR 1.1.4322)
(1 row)
```

Теперь проверим функциональность переменных сеанса.

```
<?php
require_once ("usersession.phpm");
$objSession = new UserSession();
```

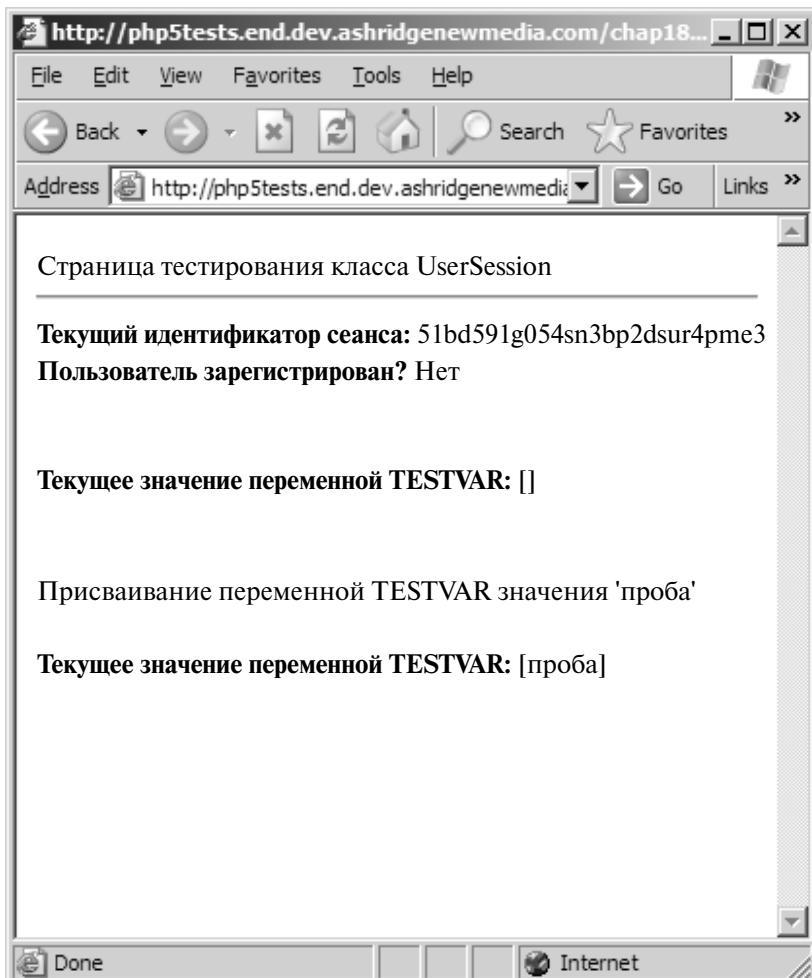


Рис. 15.6.

```
$objSession->Impress();

?>
Страница тестирования переменных сеанса класса UserSession
<HR>
<B>Текущий идентификатор сеанса: </B> <?=$objSession-
>GetSessionIdentifier();?><BR>
<B>Пользователь зарегистрирован? </B> <?=($objSession->IsLoggedIn() == true)
? "Да" : "Нет"?><BR>
<B>Текущее значение переменной TESTVAR:</B> [<?=$objSession->TESTVAR?>] <BR>
<BR><BR>
Присваивание переменной TESTVAR значения 'проба'
<BR><BR>
<?php
```

```
$objSession->TESTVAR = 'проба';
?>
<B>Текущее значение переменной TESTVAR:</B> [<?=$objSession->TESTVAR?>] <BR>
<BR><BR>
```

Запустите этот сценарий, и вы получите результат, показанный на рис. 15.6.

Щелкните на кнопке **Обновить**, и вы увидите, что значение переменной TESTVAR осталось неизменным. В таблице базы данных `session_variable` можно увидеть, какие данные сохраняются на самом деле.

```
prophtp5=# SELECT * FROM session_variable ;
+-----+-----+-----+-----+
| id | session_id | variable_name | variable_value |
+-----+-----+-----+-----+
| 6 | 168 | TESTVAR | s:3:"проба"; |
+-----+-----+-----+-----+
(1 row)
```

Из этого фрагмента видно, что данные хранятся в сериализованном виде. Однако в отличие от встроенного механизма обработки переменных сеансов PHP, для каждой переменной отводится новая строка. Это позволяет сопоставить каждому сеансу несколько отдельных переменных. При встроенной обработке переменных сеанса все они хранятся в едином ассоциативном массиве, который сериализуется для хранения на диске. Это существенно снижает производительность приложения: операции сериализации/десериализации являются достаточно ресурсоемкими, а PHP нельзя назвать самым быстрым языком в мире в смысле обработки строк (скорее, это “звание” принадлежит языку Perl). Поэтому использование базы данных существенно повышает производительность приложения.

Класс `UserSession` чрезвычайно прост в использовании. Рассмотрим его более детально, чтобы понять логику его проектирования и уверенно применять его в различных приложениях.

Как работает класс `UserSession`

Сначала рассмотрим закрытые переменные-члены этого класса.

Переменная	Назначение
<code>php_session_id</code>	Сгенерированный PHP идентификатор сеанса, состоящий из 32 символов
<code>native_session_id</code>	Идентификатор сеанса, используемый в базе данных. Он не передается Web-браузеру, а применяется только для связи сущностей в базе данных
<code>Dbhandle</code>	Дескриптор соединения с базой данных. В серьезных приложениях он объявляется как глобальный ресурс, а класс <code>UserSession</code> является его пользователем, как и другие классы
<code>Logged_in</code>	Логическая переменная, определяющая активность сеанса. Если она принимает значение <code>true</code> , идентификатор пользователя доступен
<code>user_id</code>	Идентификатор (полученный из базы данных) текущего зарегистрированного пользователя
<code>session_timeout</code>	Время ожидания. Если период между запросами превышает эту величину, сеанс прерывается
<code>session_lifespan</code>	Максимальный срок действия сеанса. Он используется в PHP для настройки режима cookie и сборки мусора для очистки базы данных от данных неактивных сеансов

Конструктор

Разрабатываемый класс должен легко подключаться, т.е. программист может однажды установить его и забыть о его существовании.

Перечислим функции конструктора.

- Установка соединения с базой данных. Эта функция в корпоративных приложениях обычно выполняется другим классом.
- Управление обработкой сеансовых событий в пользовательских классах (будет обсуждаться ниже).
- Проверка существования идентификатора сеанса во избежание открытия нового сеанса в течение действующего. Выполнение разнообразных проверок длительности сеанса, его активности и согласованности с данными пользовательского агента. В случае нарушения согласованности (или нахождения мусора) PHP открывает новый сеанс.
- Установка параметра длительности сеанса, который будет использоваться при обработке данных cookie.
- Нормальное начало сеанса.

Обратите внимание на интересный синтаксис, используемый для функции `session_set_save_handler()`.

```
session_set_save_handler(
    array(&$this, '_session_open_method'),
    array(&$this, '_session_close_method'),
    array(&$this, '_session_read_method'),
    array(&$this, '_session_write_method'),
    array(&$this, '_session_destroy_method'),
    array(&$this, '_session_gc_method')
);
```

Параметры этого метода не обязательно являются строковым представлением имен функций. В качестве аргументов можно передать массив из двух компонентов. Первым является экземпляр класса (в данном случае `&$this`), а вторым — имя метода этого класса.

Обработчик `_session_read_method()`

Поскольку этот метод вызывается первым для управления сеансом после установки корректного идентификатора сеанса, необходимо гарантировать актуальность базы данных сеанса. За это отвечает данный метод.

Если генерируемый PHP 32-символьный идентификатор сеанса существует в базе данных, то переменные-члены класса (`logged_in` и `user_id`) обновляются в соответствии с записью базы данных. Если идентификатор сеанса в базе данных не существует, то он добавляется по умолчанию.

Функция `Impress()`

Этот метод позволяет определить факт нового обращения к странице. Он должен вызываться на любой странице, в которой используется класс сеанса, сразу же после его инстанцирования.

В базе данных существует столбец с информацией о последнем посещении, данные которого используются для определения интервалов ожидания в течение сеанса. Поэтому очень важно вызывать этот метод, если вы хотите отслеживать активность пользователя.

Функция IsLoggedIn()

Этот метод просто сообщает с помощью закрытой переменной-члена об успешной регистрации пользователя в рамках сеанса.

Функции GetUserID() и GetUserObject()

Если пользователь зарегистрировался, эти две функции возвращают идентификатор зарегистрированного пользователя (из таблицы в базе данных) и по возможности инстанцированный объект класса пользователя, соответственно. Этот класс не описан в данной главе, поскольку он не относится к теме управления сеансами. Однако в серьезных корпоративных приложениях его придется создать для чтения и записи свойств зарегистрированного в данный момент пользователя.

Такой класс инстанцируется с использованием идентификатора, передаваемого в качестве единственного параметра конструктора, и работает в соответствии с принципами класса GenericObject, описанного в главе 7.

Функция GetSessionIdentifier()

Этот метод возвращает 32-символьный идентификатор сеанса PHP, а не внутренний идентификатор базы данных. Идентификатор PHP чаще используется в приложениях, поскольку внутренний идентификатор обычно применяется только в базе данных и никогда не передается приложению или Web-браузеру.

Использование класса UserSession

Созданный в этой главе класс UserSession является модульным компонентом, удобным для повторного использования. Его легко можно включить практически в любой проект, связанный с поддержкой сеансов или аутентификацией зарегистрированных пользователей.

При разработке класса обеспечен объектно-ориентированный интерфейс к встроенному механизму обработки сеансов PHP, и функциональность переменных сеанса заменена более гибким вариантом, в котором каждая переменная хранится в отдельной строке таблицы. Это ускоряет обработку переменных сеанса и делает код более понятным.

Интеграция этого класса с приложением выполняется очень легко. Возможно, вам придется расширить функциональность этого класса, но его самодостаточность существенно упрощает этот процесс по сравнению с расширением встроенной функциональности управления сеансами в PHP.

Отметим, что в разработанном классе реализован единственный дополнительный механизм безопасности, описанный в начале главы, — проверка данных пользовательского агента. При необходимости вы можете добавить дополнительную функциональность, в частности анализ вариаций IP-адреса.

Резюме

Данная глава посвящена сеансам — их функциональному назначению, способам реализации в приложениях и устранению некоторых проблем с безопасностью, связанных с их использованием.

Здесь рассказано о традиционной реализации сеансов в языке PHP и неадекватности этого подхода по отношению к сложным профессиональным программным продуктам. Вы узнали, как легко расширить эти функции и обеспечить их масштабируемость.

И наконец, эти знания были применены для построения единого модульного самодостаточного класса `UserSession`, который можно легко интегрировать в любое приложение.

16

Каркас для модульного тестирования

Одним из основных преимуществ структурного или модульного программирования на любом языке является самодостаточность практически всех разрабатываемых компонентов. Их просто можно изъять из исходного проекта и использовать по своему усмотрению. Это касается и тестирования.

Обеспечение необходимой функциональности компонентов — важнейшая задача в разработке приложения. Как уже упоминалось в этой книге, прежде чем встраивать компоненты в большое приложение, их обязательно необходимо протестировать.

Для сложных классов (или даже иерархии из нескольких классов) разработка тестовой стратегии требует квалификации и опыта. Возникает непреодолимое желание воспользоваться функцией `var_dump()` и записать огромные объемы данных в журнал регистрации ошибок. Однако, чтобы сэкономить значительное количество времени, лучше использовать специальный программный каркас для модульного тестирования. На это есть еще одна причина: профессиональный подход к тестированию предполагает демонстрацию начальству и заказчикам конкретных результатов, а не сотен записей `error_log()` в журнале ошибок.

Для PHP существует специализированный каркас, полуофициальная версия которого называется PHPUnit. В этой главе вы не только научитесь устанавливать и использовать PHPUnit, но и узнаете о преимуществах его повседневного применения. В завершение этой главы мы создадим класс, содержащий сознательные логические ошибки, которые трудно обнаружить, и применим PHPUnit для их исправления.

Методология и терминология

Перед изучением каркаса PHPUnit рассмотрим более детально идею модульного тестирования. Сначала вспомним традиционную последовательность действий, выполняемую во время тестирования компонента. Знаком ли вам следующий подход?

1. Написать класс.
2. Создать небольшой тестовый сценарий, который запрашивает класс с помощью функции `require()`, создает экземпляр класса, вызывает его некоторые методы и выдает какие-то результаты.
3. Запустить сценарий.
4. Сверить результат выполнения сценария с ожидаемым.
5. Если результат вас устраивает, можно двигаться дальше, если нет, повторить пп. 3 и 4, чтобы получить требуемый результат.
6. Встроить компонент в приложение.

Возможно, вы удивитесь, но на самом деле такой подход распространен куда более широко, чем можно себе представить.

Главная проблема такого подхода к тестированию компонентов состоит в том, что он не полон и не систематичен. Другими словами, вы можете протестировать только часть компонента, затратив на это больше времени, чем необходимо. Более того, даже если вы найдете ошибку, ваш короткий сценарий не позволит ее исправить. Кроме того, при каждом запуске теста вам придется принимать решение о том, насколько удачно он завершился.

К счастью, существует другой подход, который требует лишь некоторого изменения порядка тестирования компонента.

1. Разработать интерфейс (но не реализацию) класса.
2. Создать тестовый пакет для пустого класса и проверить, что все в порядке.
3. Написать реализацию класса.
4. Запустить пакет снова.
5. Исправить ошибки, которые приводят к неожиданным результатам, и затем вернуться к п. 4.
6. Встроить компонент в приложение.

Давайте более подробно рассмотрим каждый пункт и точнее определим, что имеется в виду под тестовым пакетом. Понятие тестового класса будет раскрыто ниже в этой главе.

Разработка интерфейса класса

Давайте вспомним некоторые принципы объектно-ориентированного подхода (ООП). При описании объектно-ориентированного подхода к проектированию часто подчеркивается, что наиболее важной частью является интерфейс и менее важной — реализация.

Это может показаться странным. Но давайте проследим за ходом мысли разработчика, проектирующего класс пользователей компьютерных систем.

“Мне нужны следующие свойства: имя, фамилия пользователя, зашифрованный пароль. Также нужны методы, которые будут сообщать приложению, к какой группе пользователей относится данный человек, метод, который позволяет узнать, как и когда он последний раз регистрировался в системе, и ...”

Вы поступаете разумно, даже если не обращаете на это внимания. Вы перечисляете поля и методы объекта. При этом вы не пытаетесь определить, как будете определять время с момента последнего входа в систему. На самом деле есть несколько вариантов решения. Это не имеет значения. Как бы вы ни реализовали метод, это не повлияет на интерфейс класса.

Интерфейс создается на основе знаний о предполагаемой работе приложения и закладывает первый камень в фундамент реализации требований. Поэтому вполне можно сказать, что реализация — это всего лишь деталь. Как специалист по PHP, вы, возможно, отдаите реализацию этих деталей на откуп подчиненному (если такие у вас имеются).

Если подойти с этих позиций к модульному тестированию, задача окажется довольно простой. Прежде всего, оставьте все методы класса пустыми и ничего не пишите в фигурных скобках. Мы вернемся к ним несколько позже.

Создание пакета тестирования для класса

Создав “скелет” класса, неплохо сразу же создать и тестовый пакет для него. Пока рассмотрим теорию. О конкретной реализации пакета речь пойдет в следующем разделе.

Задача тестового пакета — обеспечить простое решение для тестирования класса по принципу “черного ящика”. При запуске этого пакета он оперирует важными функциональными возможностями класса, производит серию тестов и сообщает, когда они были удачными, а когда нет.

Важно понять природу этих тестов. Тестовый пакет реализуется с помощью расширения и называется тестовым классом. Он реализуется как часть каркаса для модульного тестирования, который мы собираемся использовать (в данном случае каркаса PHPUnit, о котором речь пойдет в следующем разделе). Необходимо позаботиться о реализации некоторых “административных” деталей в расширенном классе (в основном связанных с необходимостью создания и удаления экземпляра тестируемого класса), после чего можно произвольным образом реализовать любой метод тестирования для любого реального метода тестируемого класса.

Тестовые методы должны быть названы в соответствии с требованиями каркаса для модульного тестирования, чтобы их можно было выполнять как часть тестового пакета. Они ответственны за тестирование базовой функциональности основного класса.

Каждый тестовый метод работает по стандартному шаблону.

1. Задаются некоторые входные параметры для метода тестируемого класса.
2. Определяются ожидаемые результаты, которые должен возвращать метод тестируемого класса в ответ на эти входные параметры.
3. Вызывается соответствующий метод с данными входными параметрами и фиксируется результат.
4. Выполняется проверка соответствия полученных результатов ожидаемым.

Больше ни о чем беспокоиться не стоит. Функциональность тестового пакета обеспечивает каркас модульного тестирования. Именно он отвечает за нужную реакцию программы. Это касается и каркаса PHPUnit, с которым вы ознакомитесь в самом ближайшем времени.

При создании тестового класса необходимо обеспечить внутреннюю работу тестового пакета.

На данном этапе уже можно запустить черный ящик и пронаблюдать за его неудачной работой. Это закономерно, так как ваш класс пока не содержит реализации. Необходимо получить первое представление о том, как будут выглядеть результаты после реализации класса.

Реализация класса

Не изменяя интерфейс класса, займитесь его реализацией. Пока нет необходимости что-либо тестировать, это задача пакета тестирования. Однако можно выполнить синтаксическую проверку и удостовериться в отсутствии опечаток. Тестовый пакет на самом деле больше подходит для отслеживания логических, а не компиляционных ошибок.

Если вам не приходилось исправлять подобные ошибки, можно провести синтаксическую проверку, запустив проверяемый файл из командной строки. Для этого нужно задать команду `rph` с параметром `-l`, после которой указать имя файла или класса, который подлежит проверке.

Если класс взаимодействует с базой данных, перед тем как заняться реализацией, целесообразно проверить SQL-запросы. Так как PHP не связан с конкретным сервером базы данных, разобраться с синтаксисом SQL лучше сейчас, а не после того, как обнаружится, что тестовый пакет не может выполнить один или несколько методов. Кроме того, данные в базе могут изменяться независимо от кода приложения, поэтому будет довольно сложно предугадать ожидаемый результат выполнения метода, который делает запрос к базе данных.

Вот пример простого запроса для типичного класса пользователя.

```
$sql = "SELECT group_id FROM user_group WHERE user_id=$user_id";
```

Замените `$user_id` осмысленной переменной и проверьте, выполняется ли запрос с использованием консоли PostgreSQL (или какой-либо другой). Если да, то все в порядке.

Предположим, проверка синтаксиса кода PHP и SQL-запросов прошла успешно. Теперь можно реализовать все методы класса и приступить к тестированию.

Повторный запуск

Запустите тестовый пакет еще раз. Сейчас, когда реализованы все методы класса, вам должно повезти больше, чем в первый раз.

Если класс прошел тесты не на всех методах, самое время проверить свои действия. Здесь пригодятся традиционные методы отладки, которые подробно описаны в главе 23 “Обеспечение качества”.

После того как ваш класс со 100%-ным успехом пройдет все тесты, можно считать, что он готов для работы в конечном продукте, и его можно встраивать в материнское приложение.

Знакомство с PHPUnit

Разработанный Себастьяном Бергманом (Sebastian Bergmann) каркас PHPUnit является одним из многих доступных средств тестирования модулей PHP в процессе разработки. Причинами, по которым можно выбрать PHPUnit, являются его бесплатное распространение, простота и, что наиболее важно, доступность из PEAR.

На самом деле в настоящее время существует два пакета, которые называются PHPUnit. Эти пакеты предназначены для работы с PHP и выполняют примерно одни и те же задачи; они даже содержат одинаковые части реализации. Реализация, которая рассматривается в этой главе, написана Себастьяном Бергманом и больше всего подходит для работы с PHP 5.

Пакет PHPUnit представляет собой набор классов, а не отдельный класс. Эта глава посвящена функциональности тестового класса и пакета тестирования. Если вас

интересуют другие аспекты PHPUnit, после изучения материала этой главы можно обратиться по адресу <http://www.sebastian-bergmann.de/PHPUnit/>, где этот пакет описан более подробно.

Те из вас, которые имеют опыт работы с Java, найдут много общего между PHPUnit и JUnit (<http://www.junit.org>). Это не удивительно, так как PHPUnit основан на JUnit. Тогда большая часть материала этой главы будет понятна читателям. Однако вряд ли вам известно, как этот мощный инструментарий можно применить в PHP 5.

Установка PHPUnit

Сначала нужно установить PHP и каркас PHPUnit. Он не является частью базовых классов PEAR, поэтому его придется установить самостоятельно. Даже если он был установлен ранее, стоит проверить, работает ли вы с последней версией программы.

Для инсталляции PHPUnit используйте стандартный синтаксис PEAR. Если вы работаете с последней версией, PEAR сообщит об этом. Если нет или он не установлен, на экране появится примерно следующее.

```
root@genesis: ~#pear install PHPUnit
downloading PHPUnit-0.6.2
...done: 11,551 bytes
install ok: PHPUnit 0.6.2
root@genesis:~#
```

Если вы хотите произвести установку вручную, например, в среде, где вам не предоставлен доступ к серверу с правами root, просто посетите страницу пакета PEAR по адресу <http://pear.php.net/package/PHPUnit>, загрузите и распакуйте архив там, где это необходимо. После успешной установки необходимого программного обеспечения можно продолжить работу.

Использование PHPUnit

Этот раздел посвящен реализации концепции тестовых пакетов и тестовых классов в PHPUnit, т.е. воплощению теории в практику.

Написав код тестового класса, вы сможете легко разработать тестовый пакет для обеспечения механизма тестирования своего класса по методу “черного ящика” даже в случае внесения значительных изменений в код исходного класса.

Тестовые классы

Посмотрим, как провести тестирование, используя PHPUnit.

Допустим, тестовый класс уже создан и сохранен в отдельном файле под именем `TestClass.phpm`. Поскольку пока речь идет о теории использования PHPUnit, на этом этапе не так важно, что содержится в классе. В соответствии со сказанным ограничимся следующим кодом класса.

```
<?php
class TestClass {
    private $testVar;

    function myMethod($strParam) {
        $this->testVar = $strParam;
        return('ожидаемый результат');
    }
}
```

Создайте тестовый пакет в отдельном PHP-файле и сохраните его под именем `testcase.phpm`.

Не забудьте включить в тестовый пакет директивы использования необходимого файла тестового класса и нужного класса PHPUnit.

```
require_once("testclass.phpm")
require_once("PHPUnit.php")
```

Если для установки PHPUnit вы использовали PEAR, то найти необходимый файл `PHPUnit.php` не составит труда. Если же установка выполнялась вручную, придется явно указать расположение файла, например, следующим образом.

```
require_once("/home/ed/myphplib/PHPUnit.php");
```

В этом примере PHPUnit установлен в каталоге `/home/ed/myphplib`.

Теперь разберемся, как расширить класс `PHPUnit_TestCase` и сформировать собственный класс с некоторым псевдокодом.

```
class MyTestCase extends PHPUnit_TestCase
{
    var $objMyTestClass;

    function __construct($name) {
        $this->PHPUnit_TestCase($name);
    }

    function setUp() {
        $this->objMyTestClass = new TestClass();
    }

    function tearDown() {
        unset($this->objMyTestClass);
    }

    function testMyMethod() {
        $actualResult = $this->objMyTestClass->myMethod('параметр');
        $expectedResult = 'ожидаемый результат';
        $this->assertTrue($actualResult == $expectedResult);
    }
}
?>
```

Разберем этот фрагмент шаг за шагом.

```
class MyTestCase extends PHPUnit_TestCase
```

Классу нужно присвоить некоторое имя. Помните, что он должен расширять класс `PHPUnit_TestCase`.

```
var $objMyTestClass;
```

Единственная переменная-член, которая понадобится в тестирующем пакете — это экземпляр тестового класса. В более сложных ситуациях, возможно, придется использовать несколько копий одного или даже нескольких классов. Конечно, количество классов, используемых в teste, необходимо минимизировать. При этом никогда не следует объединять несвязанные друг с другом классы в рамках одного пакета тестирования.

```
function __construct($name) {
    $this->PHPUnit_TestCase($name);
}
```

Это конструктор расширенного тестового пакета. Единственная его задача — вызвать конструктор родительского класса. Обратите внимание также на обязательный параметр `$name`, который используется классом тестового пакета.

```
function setUp() {
    $this->objMyTestClass = new TestClass();
}

function tearDown() {
    unset($this->objMyTestClass);
}
```

Эти два метода, по существу, представляют собой виртуальный конструктор и деструктор. Соответственно, они вызываются до и после того, как выполняются различные тестовые функции. Их основное назначение — инстанцировать рабочий экземпляр тестового класса в определенную ранее переменную-член.

```
function testMyMethod() {
    $actualResult = $this->objMyTestClass->myMethod('параметр');
    $expectedResult = 'ожидаемый результат';
    $this->assertTrue($actualResult == $expectedResult);
}
```

Здесь определяется основная задача тестового пакета. Для каждого тестируемого метода объявляется специальный метод. Имя этого метода должно начинаться со слова `test`, тогда он будет автоматически выполняться тестовым пакетом. За словом `test` должно следовать название реального метода тестового класса. Это существенно проясняет работу тестового пакета.

Реальная функциональность данного метода в значительной мере зависит от разработчика. В конечном счете вам придется выполнить следующую последовательность операций. Для начала придется объявить (т.е. явно указать) или определить тестовый параметр, который будет передаваться методу тестового класса. Затем придется объявить или определить ожидаемый результат для данного параметра, после чего вызывается метод с этим параметром. И наконец, полученный результат сравнивается с ожидаемым.

Сравнение должно производиться с использованием специальных проверочных методов, предоставляемых пакетом PHPUnit. Эти методы позволяют составить отчет по результатам тестирования, выполняемого тестовым пакетом.

Существует множество методов проверки — `assertEquals` (проверка на равенство), `assertTrue` (проверка истинности), `assertFalse` (проверка ложности), `assertNotNull` (проверка неравенства нулю), `assertSame` (проверка соответствия), `assertNotSame` (проверка несоответствия), `assertType` (проверка типа) и `assertRegExp` (проверка регулярного выражения). Их функциональность в основном определяется названиями. Для получения дополнительной информации посетите узел `/usr/local/lib/php/PHPUnit/Assert.php` или найдите файл `Assert.php` в каталоге PHPUnit.

Тестовый пакет

Использование тестового класса в тестовом пакете — относительно простая процедура. Следующий код необходимо сохранить в файле `testsuite.php`.

```
<?php
require_once 'mytestCase.phpm';
require_once 'PHPUnit.php';
```

```
$objSuite = new PHPUnit_TestSuite("MyTestCase");
$strResult = PHPUnit::run($objSuite);

print $strResult->toString();
?>
```

Замените `MyTestCase` реальным именем класса тестового пакета. Если запустить этот код из командной строки (а не в Web-браузере), то в случае успешного выполнения методов будет получен примерно следующий результат.

```
TestCase objMyTestClass->myMethod() passed
```

Если какой-то метод выполнить не удастся, результат будет иметь примерно следующий вид.

```
TestCase objMyTestClass->myMethod() failed: expected true, actual false
```

При запуске теста в Web-браузере (а не из командной строки) вместо метода `toString` необходимо использовать метод `toHTML`, чтобы конвертировать символ возврата каретки в дескриптор `
` и т.д.

Как видите, создание элементарного тестового класса — довольно простое упражнение. Но как применить эту методологию к тестированию настоящего класса? Следующий раздел даст ответ на этот вопрос.

Зачем беспокоиться?

На первый взгляд, довольно сложно разглядеть то огромное преимущество, которое предоставляет этот метод тестирования, по сравнению с обычным методом тестирования, описанным ранее и состоящим в ведении журнала ошибок.

Можно назвать четыре основных преимущества. Во-первых, тот черный ящик, в который превратится тестовый класс, позволит впоследствии эффективно тестировать этот класс, даже после внесения изменений в проект. Во-вторых, тестовый пакет станет своеобразной палочкой-выручалочкой, которую можно использовать в различных частях проекта, внося лишь небольшие изменения. В-третьих, этот метод тестирования обеспечит некоторые гарантии, которые никогда не сможет обеспечить подход, основанный на использовании обычных методов отладки. И наконец, при формальном тестировании с использованием описанного каркаса вы сможете значительно уменьшить бремя проверки функциональности своих классов.

В следующих разделах эти преимущества будут описаны более подробно.

Возвратное тестирование

Большая часть кода приложения на определенном этапе подвергается некоторой доработке и эволюции. Не важно, вы или не вы займитесь его доработкой, но можно смело делать ставки на то, что в скором времени код подвергнется изменению.

Возможно, вам придется переработать существующий метод, чтобы повысить его эффективность, сохранив все прежние функции. Единственный способ добиться этого — полностью переписать метод, оставляя неизменным внешний интерфейс, но улучшая внутренний алгоритм. В этом случае тестовый пакет будет очень полезен. Просто запустите его со старым и новым алгоритмом и убедитесь, что получили одни и те же результаты.

Более типичный вариант — дополнение класса новым методом для поддержки новых бизнес-требований. Даже в этом случае тестовый пакет окажет некоторую помощь — он позволит проверить, что новый метод никоим образом не изменяет функциональность существующих. После такой проверки останется только снабдить тестовый пакет новым методом.

Таким образом, использование каркаса для модульного тестирования облегчает решение задачи возвратного (или регрессионного) тестирования.

Удобство использования каркаса

Пакет тестовых классов имеет куда более широкое применение, чем кажется на первый взгляд. С его помощью можно быстро и надежно протестировать и другие классы с похожим или аналогичным интерфейсом.

В частности, в приложении могут использоваться производные классы, перекрывающие некоторые функции родительских классов. Одним из преимуществ новой объектной модели PHP 5 является поддержка абстрактных интерфейсов классов. Хорошим примером реализации интерфейса являются объекты, представляющие сущности из базы данных и имеющие методы `getProperty`, `setProperty` и подобные им. Тестовый пакет можно сразу или с небольшой адаптацией использовать для обеспечения быстрой и надежной проверки функциональности таких классов.

Гарантия качества

Разработку профессионального программного обеспечения обычно выполняет многочисленная команда, включающая менеджера проекта, главного архитектора, рядовых разработчиков и дизайнеров и т.д.

При создании таких систем каждый день пишутся тысячи строк кода. Главный архитектор физически не сможет каждый день проверять все компоненты программного обеспечения, создаваемые каждым из разработчиков.

При использовании методологии структурированного тестирования модное слово *взаимодоверие* становится вполне достижимой реальностью. Можно просто доверять коду разработчиков, потому что такая методология обеспечивает тщательность проверки, которую никогда не смогут обеспечить другие, бессистемные, методы тестирования. Главный разработчик может быть уверен в том, что и во время разработки, и в конечном итоге результат будет именно таким, на который он рассчитывает.

Формальная природа этого процесса позволяет нетехническим специалистам, например менеджерам и бухгалтерам, следить за развитием проекта.

Упрощение функционального тестирования

Часто говорят, что наименее интересная часть любого проекта — это его тестирование. Верно это утверждение или нет, неизвестно, но на эту часть приходится выделять и время, и деньги.

Если каждый разработчик будет сам выполнять функциональное тестирование своих компонентов, сложность тестирования всего приложения в целом значительно уменьшится. Конечно, все трудности полностью не исчезнут, но, как вы знаете из главы 23, модульное тестирование является жизненно необходимым.

Пример из жизни

Следующий пример поможет на практике реализовать знания о PHPUnit и каркасах для модульного тестирования в целом.

Рассматриваемый компонент называется `xDir`. Это один из вариантов встроенно-го PHP класса `Dir` (см. <http://www.php.net/manual/en/class.dir.php>). Единственное отличие состоит в том, что `xDir` поддерживает рекурсию подкаталогов, что бывает очень полезно.

Синтаксис `xDir` фактически идентичен классу `Dir`. Кроме того, в нем добавлен специальный параметр, указывающий на то, нужна ли рекурсия.

Читатели, хорошо знакомые со структурным программированием, знают, что такое рекурсия. Это просто вызов функцией самой себя для выполнения некоторых операций рекурсивно по некоторой иерархии. Если вам когда-либо приходилось генерировать фракталы, для вас рекурсия не вызовет никаких трудностей.

Рассмотрим следующий код. Он содержит ошибку, которую довольно сложно обнаружить.

```
<?php

class xdir {

    public $path;
    public $entries = array();
    public $counter = 0;
    public $isRecursive;

    public function __construct($path, $recursive = false) {
        if ((substr($path, strlen($path)-1, 1) == "/") && (strlen($path) != 1)) {
            $path = substr($path, 0, strlen($path)-1);
        };
        $this->path = $path;
        $this->isRecursive = $recursive;
        if ($this->path) {
            $this->_getList($this->path);
        };
    }

    public function read() {
        if ($this->counter <= (sizeof($this->entries)-1)) {
            $s = ($this->entries[$this->counter]);
            return($s);
            $this->counter++;
        } else {
            return(false);
        };
    }

    public function isRecursive() {
        return($this->isRecursive);
    }

    public function rewind() {
        $this->counter = 0;
        return(true);
    }

    public function close() {
        return(true);
    }
}
```

```

public function _getDirList ($dirName) {
    $objDir = dir($dirName);
    if ($objDir) {
        while($strEntry = $objDir->read()) {
            if ($strEntry != "." && $strEntry != "..") {
                if (!is_dir($dirName."/". $strEntry)) {
                    array_push($this->entries, $dirName."/". $strEntry);
                } else {
                    if ($this->isRecursive) {
                        $this->_getDirList($dirName."/". $strEntry, true);
                    };
                };
            };
        };
        $objDir->close();
    };
}
?>

```

Обычная тестовая процедура выглядит примерно так.

```

<? require_once ("xdir.phpm"); ?>
<html>
<head><title>Простое тестовое приложение для xDir</title>
</head>
<body>
<?php
    $objxDir = new xDir("home/ed/public_html/pacha",true);
    while (false !==($entry = $objxDir->read())){
        echo $entry."<br>\n";
    }
    $objxDir->close();
?>
</body>
</html>

```

Заметим, что этот фрагмент взят из примера php.net. Мы рекурсивно запрашиваем у класса содержимое каталога /home/ed/public_html/pacha, получаемое как результат выполнения сценария. Это тестовое приложение будет работать как надо, если класс реализован правильно.

Печально то, что класс не работает, поэтому мы получим одну строчку результата, а затем бесконечный цикл. Журнал ошибок не отобразит ничего, так как интерпретатор PHP будет пребывать в уверенности, что сценарий все делает правильно.

Что же делать? Для начала создадим тестовый пакет. Заметим, что все переменные-члены и методы остаются открытыми. Это не проявление “плохой” техники программирования с нашей стороны. Такой прием позволяет легче разобраться с состоянием класса. После того как мы убедимся, что класс работает корректно, перенесем необходимые данные (а именно три переменные и метод _getDirList()) в область private.

```

class MyTestCase extends PHPUnit_TestCase
{
    var $objXDirClass;

    function __construct($name) {
        $this->PHPUnit_TestCase($name);
    }
}

```

```

function setUp() {
    $this->objXDirClass = new XDir("", "true");
}

function tearDown() {
    unset($this->objXDirClass);
}

function testRead() {
    $this->objXDirClass->counter = 1;
    $intCounterBefore = $objXDirClass->counter;
    $this->objXDirClass->entries = array("/home/ed/test1",
"/home/ed/test2", "/home/ed/test3", "/home/ed/test4");
    $strActualResult = $this->objXDirClass->read();
    $intActualCounterAfter = $this->objXDirClass->counter;
    $strExpectedResult = "/home/ed/test2";
    $intExpectedCounterAfter = 2;
    $this->assertTrue(($strActualResult == $strExpectedResult) &&
($intActualCounterAfter == $intExpectedCounterAfter));
}
}
?>
```

Заметим, что здесь мы тестируем только метод `read()`. В реальной ситуации вам бы пришлось разрабатывать тестовые методы для каждого метода класса, включая и `_getDirList()`, но поскольку `_getDirList()` обращается напрямую к файловой системе, фактически невозможно указать ожидаемый результат работы этого метода. Вначале протестируем первый метод. Может, именно в нем была допущена ошибка. Затем можно вернуться к тестированию метода `_getDirList()`.

Обсудим логику этого фрагмента. Сначала счетчик массива устанавливается в единицу. Эта переменная указывает текущую позицию в списке файлов (он начинается с 0, поэтому 1 — вторая позиция). Затем определяем искусственный список соответствующих файлов — `test1-test4` (для удобства).

Метод должен возвратить следующий файл в списке, т.е. `test2`. При этом нужно, чтобы счетчик увеличился на единицу и принял значение 2. Применяем метод проверки истинности `assertTrue()`. Если проверка дает отрицательный результат, значит, один из двух тестов “провалился”.

Создайте тестовый пакет. Предположим, тестовый класс содержится в файле `testcase.phpm`. Создайте следующий сценарий и назовите его `testsuite.php`.

```

<?php
require_once 'xdir.phpm';
require_once 'testcase.php';
require_once 'PHPUnit.php';

$objSuite = new PHPUnit_TestSuite("MyTestCase");
$strResult = PHPUnit::run($objSuite);

print $strResult->toString();
?>
```

Запустите его в командной строке, и вы получите примерно следующий результат.

```
TestCase my testcase->read() failed: expected true, actual false
```

Это одновременно и хорошо, и плохо. Теперь понятно, что именно метод `read()` приводит к бесконечному циклу, но пока не ясно почему (можно только гадать). Измените последнюю строчку метода `testRead()`, чтобы она выглядела так.

```
$this->assertTrue($strActualResult == $strExpectedResult);
```

Запустите сценарий еще раз, и вы получите:

```
TestCase my testcase->testRead() passed
```

Отсюда можно заключить, что тест не срабатывает при проверке соответствия ожидаемого значения переменной счетчика ее реальному значению после вызова метода `read()`. Измените последнюю строку еще раз, чтобы она выглядела следующим образом.

```
$this->assertEquals($intActualCounterAfter,$intExpectedCounterAfter);
```

Запустите сценарий снова, и вы получите:

```
TestCase my testcase->testRead() failed: expected 2, actual 1
```

Отсюда видно, что переменная счетчика не увеличивается, а должна бы. Ее значение должно быть равно 2, а на самом деле — 1. Что не так? Теперь хоть понятно, где искать ошибку. Вернемся к методу `read()` исходного кода метода `xDir`.

```
public function read() {
    if ($this->counter <= (sizeof($this->entries)-1)) {
        $s = ($this->entries[$this->counter]);
        return($s);
        $this->counter++;
    } else {
        return(false);
    }
}
```

Видите в чем проблема?

Функция `return()` вызывается до того, как увеличивается значение счетчика. А PHP не обрабатывает код после вызова `return()`. И, как результат, увеличение счетчика вообще никогда не происходит.

Поменяйте строки местами.

```
public function read() {
    if ($this->counter <= (sizeof($this->entries)-1)) {
        $this->counter++;
        $s = ($this->entries[$this->counter]);
        return($s);
    } else {
        return(false);
    }
}
```

Снова запустите тест.

```
TestCase my testcase->testRead() passed
```

Вернитесь к начальному тесту, в котором два теста объединялись при помощи метода `assertTrue()`. Проверьте и его. Сейчас он работает правильно и выдает результат:

```
TestCase my testcase->testRead() passed
```

Все отлично.

Теоретически этого более чем достаточно, чтобы гарантировать работоспособность метода. Но для внутреннего спокойствия вы можете запустить исходный тестовый сценарий, который сейчас работает без проблем. Запустите его в Web-браузере и получите что-то вроде этого.

```
/home/ed/public_html/pacha/perl/cleanbuf.sh  
/home/ed/public_html/pacha/perl/cleandb.pl  
/home/ed/public_html/pacha/perl/deploylocaldata.sh  
/home/ed/public_html/pacha/perl/mailfwd.pl  
/home/ed/public_html/pacha/perl/oldphotouploads.pl  
/home/ed/public_html/pacha/perl/pafserver.pl
```

Все работает нормально. Как вы видите, каркас PHPUnit позволил выявить довольно трудно обнаружимую ошибку. Представьте себе, как долго пришлось бы ее искать при помощи `var_dump` и `error_log`.

Резюме

В этой главе вы познакомились с методологией модульного тестирования и ее преимуществами по сравнению с менее формальными методами тестирования компонентов. Были описаны принципы построения типичного каркаса для модульного тестирования, а также рассмотрен конкретный пакет PHPUnit. Вы научились его устанавливать и получили базовые представления о его работе. Затем полученные знания были использованы для устранения ошибки в реальном классе.

В следующей главе вы познакомитесь с полезной концепцией, возникшей на заре развития компьютерной техники — конечным автоматом, и его реализацией в PHP 5. Это поможет в решении довольно хитрых алгоритмических проблем, которые могут встретиться во время разработки приложений.

17

Конечные автоматы и файлы конфигурации

Большинство специалистов считают, что PHP — это современный язык, предназначенный для разработки современных приложений. Как и большинство языков разработки Web-приложений, он является высокоуровневым и предназначен, скорее, для быстрой разработки приложений, а не для оптимизации производительности при взаимодействии с операционной системой.

Действительно, высокоуровневые языки, к которым относится и PHP, расширяют уже сформированную пропасть между программированием в традиционном смысле слова (как его понимают в учебниках для студентов) и разработкой приложений (при которой язык рассматривается просто как средство решения проблемы).

Например, еще не так давно разработчики компьютерных игр участвовали в непрерывном состязании на написание самой быстрой ассемблерной функции `putpixel()`, отображающей на экране одну точку. А сегодня разработчикам игр практически не приходится иметь дело с такими низкоуровневыми функциями. Они пользуются библиотекой DirectX от компании Microsoft, которая позволяет обойти все эти проблемы. Аналогично, PHP избавляет разработчика от написания низкоуровневых высокоточных оптимизированных процедур и позволяет сосредоточиться на полезной функциональности. PHP работает по принципу: “Скажи мне, что ты хочешь сделать, но не говори как”.

Однако в PHP используются некоторые более традиционные приемы программирования. Конечно, речь идет не о написании на PHP инструкций по управлению регистрами процессора, а о реализации концепции, присущей традиционным подходам к программированию — конечных автоматах.

В этой главе речь пойдет о конечных автоматах. Будет рассказано, что это такое, как они работают, как их можно использовать для получения эффективных решений многих сложных задач, возникающих при реализации проектов. После изучения концепции конечных автоматов читатель познакомится с классом пакета PEAR, обеспечивающим быструю и простую реализацию модели конечных автоматов на PHP. Как подтверждение правильности подхода, этот класс будет использован для решения

одной из сложных задач. По ходу обсуждения будут выделены области применения модели конечных автоматов, чтобы читатель смог легко идентифицировать их в дальнейшем. Также будет рассказано о конфигурационных файлах, представляющих прекрасный пример для реализации в виде конечных автоматов. Будет описана их роль в приложениях и способ реализации на PHP.

Знакомство с конечными автоматами

Конечный автомат (Finite State Machine) — это пример абстрактной машины. Это не машина в традиционном, физическом, смысле, а логическая модель, реализованная в программном обеспечении. В некотором смысле это мини-компьютер. Он содержит устройство ввода-вывода, а также обрабатывающий элемент, позволяющий получить некоторый результат на основе введенных данных.

Реальным тестом для любой абстрактной машины является возможность ее реализации вне программной среды. Для читателей, знакомых с электроникой, простейшим примером является логическая схема И, имеющая два входа и выход, определяемый входными данными. Ее можно физически реализовать с помощью двух транзисторов, а в программном обеспечении — с помощью оператора “логическое И” (в PHP оператор `&&`).

Конечный автомат представляет собой несколько более интеллектуальную абстрактную машину.

В традиционных учебниках пишут, что конечный автомат содержит набор произвольных состояний. К ним относится начальное состояние, применяемое при первой инициализации, набор входных событий, множество выходных событий (термин “событие” здесь несколько неуместен) и функции перехода.

Другими словами, конечный автомат — это устройство для итеративной обработки набора входов и получения некоторого результата. В отличие от простого оператора `foreach` или реализации итератора для класса коллекции, это не просто выполнение одного и того же действия над множеством кандидатов. Операция, выполняемая конечным автоматом над очередным кандидатом, зависит от предыдущих действий.

Конечный автомат нельзя реализовать с помощью цикла `for`, поскольку в этом цикле над набором объектов выполняется одно и то же действие. Хотя операции цикла могут подчиняться определенной логике, выполняемые в данный момент времени действия не зависят от предыстории. Еще одним примером таких действий является инстанцирование объектов на основе обработки запросов к базе данных. Результат текущего запроса никак не зависит от информации, полученной на предыдущей итерации.

Простой конечный автомат: калькулятор для обратной польской записи

Простейшим примером конечного автомата, часто рассматриваемым в учебниках по теории этого вопроса, является калькулятор для обратной польской записи. Пользователи UNIX должны быть знакомы с программами командной строки типа `bc`. Как видно из предыдущего примера, утилиты GNU, подобные `bc`, помогают упростить синтаксический анализ арифметических выражений.

```
ed@genesis:~$ bc
bc 1.05
Copyright 1991, 1992, 1993, 1994, 1997, 1998 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
```

```
For details type 'warranty'.
7 * (9 + 1)*29
2030
```

В данном примере утилита `bc` используется для вычисления простого арифметического выражения $[7 \times (9+1) \times 29]$.

Эту задачу легко решить с помощью современных интеллектуальных средств, таких как `bc` или Excel. Однако в прежние времена приоритет операций (операции в скобках, обращение по индексу, умножение и деление, сложение и вычитание) было совсем непросто реализовать, особенно при наличии нескольких пар вложенных скобок. Приходилось избавляться от скобок, и в этом помогла обратная польская запись.

Ее концепция очень проста. Операторы, расположенные справа от своих операндов, меняются с ними местами. В предыдущем примере выражение

$$7 * (9 + 1) * 29$$

в обратной польской записи примет вид:

$$1\ 9\ +\ 29\ 7\ *\ *$$

Для ее обработки нужно реализовать стек чисел. Если в процессе перемещения по записи слева направо встречается число — оно помещается в стек. Если же встречается символ операции, два первых числа извлекаются из стека и над ними отдельно выполняется данная операция. Результат добавляется в стек как обычное число.

В предыдущем примере в стек сначала добавляются числа 1 и 9. Затем встречается знак `+`, означающий, что данные числа нужно извлечь из стека и сложить. 1 и 9 извлекаются из стека (он остается пустым), и вычисляется их сумма, составляющая 10. Число 10 помещается в пустой стек и продолжается анализ выражения. Следующее число — 29, которое добавляется в стек, за ним в стек помещается число 7. Таким образом, стек содержит значения 10, 29, 7. Затем в выражении встречается символ умножения. Два верхних числа, 7 и 29, извлекаются из стека и перемножаются. В результате получаем число 203, которое помещается в стек к числу 10. И наконец, в выражении встречается еще один символ умножения. Из стека извлекаются числа 203 и 10. Они перемножаются, давая сумму 2030. Больше в выражении нет никаких данных, поэтому 2030 является ответом.

Это очень хороший пример конечного автомата. В нем существует два состояния: помещение данных в стек или выполнение операции. Для каждого элемента данных, будь-то число или символ операции, конечный автомат определяет последнее состояние (добавление в стек или выполнение операции) и тип текущих данных (оператор или число).

Затем он определяет, что делать дальше. Перечень возможных переходов описывается следующей таблицей.

Текущее состояние	Текущий символ	Действие	Новое состояние
Добавление	Число	Добавить число в стек	Добавление
Добавление	Оператор	Извлечь два числа из стека, выполнить над ними данную операцию и результат поместить в стек	Выполнение операции
Выполнение операции	Число	Добавить число в стек	Добавление
Выполнение операции	Оператор	Извлечь два числа из стека, выполнить над ними данную операцию и результат поместить в стек	Выполнение операции

Данный конечный автомат не должен на каждом шаге определять свое следующее состояние, поскольку оно не зависит от входных данных, а задается только текущим состоянием.

Такой автомат называется устройством PDA (Push Down Automation), поскольку реализуется память в форме стека. Все устройства PDA являются конечными автоматами, но не все конечные автоматы являются устройствами PDA.

Теоретические реализации конечных автоматов

В простейшем случае конечный автомат описывается таблицами переходов.

Эти таблицы содержат отображение входных символов и текущих состояний в набор действий и следующих состояний. Для каждой комбинации входного символа и состояния должно существовать отдельное правило перехода. Например, для двух входов и двух состояний можно составить следующую таблицу переходов.

```
{вход1, текущие1} --> {действие1, следующее1}
{вход2, текущие1} --> {действие2, следующее1}
{вход1, текущие2} --> {действие1, следующее2}
{вход2, текущие2} --> {действие2, следующее2}
```

С помощью этой таблицы для каждого входного символа конечный автомат определяет, какое из действий следует выполнить и в какое состояние перейти. В общем случае определяется также правило перехода по умолчанию, дополняющее конкретные правила переходов, описанные в таблице.

Существует два типа конечных автоматов: детерминированные и недетерминированные. В детерминированных конечных автоматах следующее состояние машины определяется исключительно входным символом. Следующее состояние недетерминированного конечного автомата зависит от текущего входного символа и одного или нескольких последующих символов. Калькулятор для обратнойпольской записи (его реализация рассматривается в данной главе ниже) является недетерминированным конечным автоматом, поскольку его следующее состояние зависит от типа следующего символа и не может определяться только текущим символом.

Реализация конечных автоматов на PHP

Конечный автомат реализовать достаточно легко. Вероятно, вам приходилось писать код, который можно считать конечным автоматом, поскольку он удовлетворяет некоторым алгоритмическим требованиям. Однако в PHP существует реализация конечного автомата в виде класса пакета PEAR, получившего название FSM.

Инсталляция класса FSM

Класс FSM устанавливается как и любой другой класс пакета PEAR: с помощью дистрессетчера пакета из командной строки.

```
root@genesis:~# pear install FSM
downloading FSM-1.2.1.tgz ...
...done: 3, 151 bytes
install ok: FSM 1.2.1
```

Протестировать этот класс можно с помощью простого кода, добавляемого непосредственно в командную строку PHP.

```
root@genesis:~/public_html/prophp5# php
<?
require_once ("FSM.php");
?>
```

Нажмите комбинацию клавиш <Ctrl+D> в UNIX (или <Ctrl+Z> в Windows) для обозначения конца файла и вы не увидите никакого результата. Если интерпретатор PHP генерирует сообщение об ошибке и не может найти указанный класс, проверьте, корректно ли установлен пакет PEAR по адресу /usr/local/lib/php (или по соответствующему адресу в вашем окружении).

Синтаксис класса FSM

Класс FSM очень прост. В отличие от таких программных каркасов как PHPUnit, его не приходится расширять, а лишь реализовывать как любой другой написанный вами класс. Полный синтаксис класса FSM можно найти на Web-узле автора Джона Париса (Jon Parise) по адресу <http://www.indelible.org/pear/FSM/api/>.

Эта конкретная реализация модели конечного автомата ограничивается копированием одной строки в качестве входных данных. Соответственно, если вы хотите работать с коллекцией объектов или другими более сложными структурами данных, вам придется их сериализовать. Кроме того, символычитываются строго по одному, поэтому вам придется реализовать переходы, связанные с выделением отдельных сущностей из строки символов. Например, строка '53 15 19 * * =' будет считываться как последовательность символов '5', '3', ' ', '1', '9', ' ', '=' и т.д. Поэтому в своем коде вам придется преобразовать пару символов '5' и '3' в строку '53'.

Инстанцирование класса

Класс FSM инстанцируется следующим образом.

```
$stack = array();
$fsm = new FSM('INIT', $stack);
```

Первым параметром конструктора является начальное состояние машины. Состояние в классе FSM именуется с помощью простых строк, и их имена не имеют никакого реального значения. Единственным требованием к именованию является осмысленность идентификаторов. (Их можно рассматривать как эквивалент имен переменных.) В данном случае начальное состояние названо INIT, что выглядит достаточно логичным.

Обратите внимание, что в качестве второго параметра конструктору передается переменная стека (типа обычного массива). Конечный автомат будет использовать ее в качестве памяти. Эта переменная передается по ссылке.

Установка перехода по умолчанию

В первую очередь конечному автомату необходимо сообщить имя метода, вызываемого по умолчанию в случае отсутствия конкретного перехода для данного входа и текущего состояния. Это может быть либо функция обработки ошибки (для обработки некорректного входа), либо метод перехвата исключений.

```
$fsm->setDefaultTransition('INIT', 'Error');
```

Первый параметр означает состояние, в которое переходит конечный автомат. Второй параметр — имя метода, вызываемого перед переходом в это состояние. Вызов этого метода является необязательным, но его настоятельно рекомендуется осуществлять в собственной реализации конечного автомата.

Установка конкретного перехода

Конкретный переход для данного входного символа и данного текущего состояния задается с помощью метода `addTransition()`.

```
$fsm->addTransition('=', 'INIT', 'INIT', 'DoEqual');
```

Первым параметром является символ; вторым — исходное состояние; третьим — состояние, устанавливаемое после перехода; четвертым параметром является имя метода, вызываемого для данного перехода. Этот последний параметр действия может быть равен нулю.

Задание перехода для массива входов

С целью упрощения кода можно определить переход для одного исходного состояния и целого массива входных символов. Это очень удобно для определения единого перехода, применяемого ко всем цифрам от 0 до 9.

```
$fsm->addTransitions(range(0, 9), 'BUILDING_NUMBER', 'BUILDING_NUMBER',
'BuildNumber');
```

Первым параметром является массив символов; вторым — исходное состояние; третьим — состояние, устанавливаемое после перехода; четвертым параметром является имя метода, вызываемого для данного перехода. Опять же, последний параметр действия может быть равен нулю.

Запуск конечного автомата

После определения переходов машину можно запустить с помощью вызова метода `processList()`.

```
$fsm->processList($symbols);
```

Этому методу передается единственный параметр — строка обрабатываемых символов. Если она напрямую зависит от ввода пользователя, сначала можно выполнить некоторую проверку корректности входных данных.

Пример калькулятора для обратной польской записи

Пример реализации калькулятора для обратной польской записи можно найти в классе API пакета PEAR по адресу <http://www.indelible.org/pear/FSM/api/>. Хотя по указанному адресу приводится полный исходный код работающего класса, в нем не содержится никаких объяснений его работы. Поэтому рассмотрим простой пример его функционирования.

Полный исходный код класса можно получить в Интернет, поэтому скопируйте его и вставьте в новый файл с именем `rpn.php`. Запустите этот сценарий из командной строки, а не из Web-браузера, как показано в следующем примере.

```
root@genesis:~/public_html/prophp5# php ./rpn.php
Expression:
1 9 + 29 7 * *
2030
root@genesis:~/public_html/prophp5#
```

Реализация калькулятора для обратной польской записи — это приложение командной строки, а не традиционный PHP-сценарий для Web. Однако, если вам понадобится, вы сможете легко его преобразовать в традиционный компонент.

Как видно из предыдущего фрагмента кода, с помощью конечного автомата вычислено значение выражения $29 \times (1+9) \times 7$ и получен корректный ответ. Но как работает этот конечный автомат?

Рассмотрим исходный код. Начнем с фрагмента, следующего за определением функций.

```
$stack = array();
$ fsm = new FSM('INIT', $stack);

$fsm->setDefaultTransition('INIT', 'Error');

$fsm->addTransitionAny('INIT', 'INIT');
$fsm->addTransition('=', 'INIT', 'INIT', 'DoEqual');
$fsm->addTransitions(range(0,9), 'INIT', 'BUILDING_NUMBER',
    'BeginBuildNumber');
$fsm->addTransitions(range(0,9), 'BUILDING_NUMBER', 'BUILDING_NUMBER',
    'BuildNumber');
$fsm->addTransition(' ', 'BUILDING_NUMBER', 'INIT',
    'EndBuildNumber');
$fsm->addTransitions(array('+', '-', '*', '/'), 'INIT', 'INIT',
    'DoOperator');
```

В первых строках этого фрагмента создается экземпляр класса `FSM` и задаются все возможные переходы.

Для конечного автомата определены только два состояния. Исходное состояние называется `INIT`. Оно активизируется в момент создания класса, после выполнения операции и после добавления числа в стек. Второе состояние называется `BUILDING_NUMBER`. Оно активизируется только для склеивания цифр числа.

Обратите внимание, что для задания диапазона цифр от 0 до 9 используется метод `range()`, а перечень арифметических операций задается с помощью обычного массива. Для описания переходов определены пять пользовательских методов: `BeginBuildNumber()`, `BuildNumber()`, `EndBuildNumber()`, `DoOperator()` и `DoEqual()`. Рассмотрим их более подробно.

Метод `BeginBuildNumber()`

Этот метод вызывается в том случае, если при обработке строки встречается цифра, а машина еще не сформировала число. Например, предыдущим символом был пробел, а текущим является цифра 5. Значит, цифра 5 должна стать началом числа. Для этого вызывается данный метод.

```
function BeginBuildNumber($symbol, $payload)
{
    array_push($payload, $symbol);
}
```

Этот метод вызывается как часть перехода и принимает два обязательных параметра: ссылку на стек конечного автомата и передаваемый символ.

При вызове этого метода передаваемый символ просто добавляется в стек. Состояние автомата не изменяется.

Метод `BuildNumber()`

Этот метод вызывается в том случае, когда при обработке строки встречается цифра, а автомат ранее начал формирование числа. Допустим, последним символом была цифра 5, а теперь встретился символ 1. Следовательно, новую цифру нужно добавить к числу.

```
function BuildNumber($symbol, $payload)
{
    $n = array_pop($payload);
    $n = $n . $symbol;
    array_push($payload, $n);
}
```

Это может быть число 51 или 51310. Поскольку вид числа заранее неизвестен, из стека извлекается существующее число, к нему добавляется новая цифра, и полученное число снова помещается в стек. Например, в стеке находилось число 5. Оно извлекается из стека, к нему добавляется 1, и полученное число 51 снова помещается в стек.

Метод EndBuildNumber()

Этот метод вызывается, когда при обработке строки встречается пробел, а автомат ранее “занимался” генерированием числа. Например, метод вызывается, если после цифры 1 встречается пробел.

```
function EndBuildNumber($symbol, $payload)
{
    $n = array_pop($payload);
    array_push($payload, (int)$n);
}
```

Сформированное число извлекается из стека. Оно имеет форму строки, поскольку символы добавлялись к числу с помощью операции конкатенации строк. Прежде чем поместить число в стек, оно преобразуется в целое, над которым можно выполнять арифметические операции.

Метод DoOperator()

Этот метод вызывается в том случае, если при обработке строки встречается символ операции (+, -, * или /), а текущим состоянием является INIT, т.е. процесс формирования числа завершен (см. выше описание метода BuildNumber()).

```
function DoOperator($symbol, $payload)
{
    $ar = array_pop($payload);
    $al = array_pop($payload);

    if ($symbol == '+') {
        array_push($payload, $al + $ar);
    } elseif ($symbol == '-') {
        array_push($payload, $al - $ar);
    } elseif ($symbol == '*') {
        array_push($payload, $al * $ar);
    } elseif ($symbol == '/') {
        array_push($payload, $al / $ar);
    }
}
```

Из стека извлекаются два последних числа. Поскольку они были обработаны методом EndBuildNumber(), они являются целыми, а не строками. С помощью оператора if определяется тип выполняемой операции. Результат выполнения операции помещается в стек.

Метод DoEqual()

Этот метод вызывается в том случае, если при обработке строки встречается символ равенства, а текущим состоянием является INIT. При этом конечный автомат завершает свои операции и выводит результат, содержащийся в стеке.

```
function DoEqual($symbol, $payload)
{
    echo array_pop($payload) . "\n";
}
```

Обратите внимание, что в этом фрагменте не выполняется проверка стека на наличие других чисел. Для любого корректного выражения в обратной польской записи после выполнения всех операций стек должен содержать лишь одно число, которое и является результатом выражения.

Строго говоря, к данному конечному автомату следовало бы добавить еще одно состояние, позволяющее определить готовность стека к завершению работы. В случае наличия в стеке нескольких чисел это состояние должно принимать значение FALSE, и метод `DoEqual()` не должен вызываться.

Реальные примеры конечных автоматов

Калькулятор для обратной польской записи является хорошим учебным примером, однако вряд ли вам придется реализовывать его в своих приложениях. В частности, в PHP для вычисления значений выражений можно использовать функцию `eval()`. При этом вам не потребуется преобразовывать выражение к виду обратной польской записи.

Однако существует несколько реальных примеров, для реализации которых очень удобно пользоваться конечными автоматами.

Многие конечные автоматы предназначены для анализа строк, который можно выполнить либо с помощью чрезвычайно сложных регулярных выражений (этот метод очень популярен среди программистов на языке Perl), либо с помощью вариаций оператора `while` с набором флагов. Но ни один из этих методов нельзя назвать “элегантным”. Модель конечных автоматов позволяет создавать более простой, понятный и управляемый код.

Файлы конфигурации

Зачастую конечные автоматы используются для обработки конфигурационных файлов. С одной стороны, эти файлы должны быть понятны для человека, поскольку именно человек выполняет их модификацию, а с другой стороны, они должны быть удобны для машинной обработки. При всех преимуществах языка XML этот формат рассчитан на специалистов, поэтому редко используется для конфигурационных файлов. (Гораздо чаще в конфигурационных файлах используется формат, характерный для файла конфигурации PHP `php.ini`.) В следующих разделах этой главы речь пойдет о синтаксическом анализе конфигурационного файла подобного формата с использованием встроенных методов PHP.

Лексический анализ в задачах искусственного интеллекта

Возможно, вам приходилось играть с чат-ботами (*chat-bot*). Первые примеры таких “говорящих” роботов как, например, Элиза, обучались на чрезвычайно сложных моделях и успешно справлялись с тестом Тьюринга. Тест Тьюринга предназначен для выявления в изучаемых объектах искусственного интеллекта. Согласно этому тесту человек-интервьюер задает вопросы собеседнику, расположенному за пределами его комнаты. Если он не может отличить машину от другого человека, то данное интеллектуальное устройство можно считать реализацией искусственного интеллекта. Чат-боты — это игрушечная реализация лексического анализа. Однако и она находит реальное применение, например, в справочных системах, позволяющих пользователю формулировать свои вопросы на естественном языке.

Конечный автомат обеспечивает замечательный механизм для анализа языковых конструкций. Он может распознавать символы, слова, предложения, а на более высоком уровне — глаголы, существительные, члены предложения и т.д. Поскольку конечный автомат — это машина с поддержкой состояний, его подход к распознаванию членов предложения базируется на текущем “состоянии” предложения.

Реализация такого конечного автомата не является задачей данной книги и, возможно, слишком сложна для класса FSM, рассмотренного выше. Однако в Интернет есть множество ресурсов по этому вопросу, в том числе статья, которую можно найти по адресу <http://www.coli.uni-sb.de/~kay/motivation.pdf>.

Пользовательские конфигурационные файлы

В многих приложениях, включая Web-приложения на языке PHP, используются установочные переменные, значения которых должен задать системный администратор. Эти переменные обычно включают или отключают некоторые дополнительные средства, задают пределы и диапазоны, имена узлов и адреса, а также определяют пути к файлам.

Подобные переменные хранятся в одном или нескольких конфигурационных файлах, которые системный администратор должен модифицировать в момент установки приложения для обеспечения его корректной работы. Эти переменные можно хранить в различных форматах, включая форматы XML и INI.

Использование PHP

В некоторых случаях целесообразно использовать файл constants.php с объявлением переменных, как в следующем примере.

```
<?
# Имя узла или IP-адрес сервера SQL Server для Windows
define("DATABASE_HOST", "winsqlDb");

# Имя пользователя и пароль для базы данных
define("DATABASE_USER", "xyz");
define("DATABASE_PASS", "abc");

# Имя базы данных
define("DATABASE_NAME", def);

?>
```

К описанным в нем константам можно обратиться по имени прямо из кода приложения. Приведем пример такого обращения, в котором соединение с базой данных устанавливается с помощью этих констант.

```
if (!array_key_exists("db", $GLOBALS)) {
    $GLOBALS["db"] = mssql_connect(DATABASE_HOST, DATABASE_USER, DATABASE_PASS);
    mssql_select_db(DATABASE_NAME);
};
```

Этот подход достаточно прост. Однако он связан с проблемой формулировки требований по модификации данного конфигурационного файла. Многие системные администраторы способны модифицировать этот конфигурационный файл, но могут легко допустить ошибку, например забыть точку с запятой (;). В этой связи использовать синтаксис PHP в конфигурационных файлах нежелательно.

Использование XML

Поскольку PHP 5 обеспечивает поддержку формата XML, возникает соблазн использовать этот формат в конфигурационных файлах, как показано в следующем примере. Этот код можно сохранить в файле config.xml.

```
<database>
  <hostname>winsqlDb</hostname>
  <hostusername>xyz</hostusername>
  <hostpassword>abc</hostpassword>
  <hostdbname>def</hostdbname>
</database>
```

Синтаксический анализ этого фрагмента можно выполнить с помощью следующего оператора.

```
$confFileObject = simplexml_load_file("config.xml");
$strDatabaseHostname = $confFileObject->hostname;
```

Этот подход предпочтительнее использования обычного языка PHP, поскольку защищен от случайных ошибок, связанных с модификацией файла. Однако для его использования администратор должен владеть языком XML, особенно для работы со сложными структурами данных. Например, даже опытным системным администраторам не всегда очевидно, какие символы необходимо выделять как служебные при формировании корректного документа XML.

Еще одним недостатком формата XML является неудобство чтения такого файла с экрана, особенно если конфигурационный файл имеет большой размер. Кроме того, в файлы XML нельзя включать детальные комментарии для потенциальных редакторов.

Использование INI-файлов

Последним и наиболее предпочтительным подходом является использование файлов .INI. Это довольно произвольный формат, применяемый в течение многих лет и признанный в качестве неформального стандарта, в частности в системе Windows. В ранних версиях Windows файлы .INI использовались вместо современного системного реестра.

INI-файлы чрезвычайно просты для чтения и редактирования, а также допускают использование комментариев, помогающих в процессе редактирования. Благодаря комментариям можно легко отказаться от ненужных параметров.

Универсальный формат для INI-файлов

Для INI-файлов существует общепринятый формат, имеющий следующий вид. Код этого фрагмента можно поместить в файл config.ini.

```
; Это пример конфигурационного файла в формате INI

[database]
; Имя узла или IP-адрес сервера SQL Server для Windows
DATABASE_HOST = winsqlDb

; Имя пользователя и пароль для базы данных
DATABASE_USER = xyz
DATABASE_PASS = abc

; Имя базы данных
```

```

DATABASE_NAME = def

[server_paths]
installation_path = /home/ed/public_html/application
public_url = "http://www.example.com/application"

```

Значения всех параметров задаются в формате

```
имя = значение
```

Имя параметра не должно содержать пробелов. Пробел ставится после имени параметра, за ним добавляется знак равенства, за которым следует пробел и значение. Значение записывается в одной строке.

ВINI-файле можно выделять различные разделы, облегчающие ориентацию пользователя в файле. Имена разделов заключаются в квадратные скобки. Строки комментариев вINI-файле начинаются с символа ; .

АнализINI-файлов с использованием встроенных методов PHP

ПосколькуINI-файл PHP записан именно в этом формате и интерпретатор PHP должен его корректно обрабатывать, разработчики PHP решили предоставить соответствующую процедуру обработкиINI-файлов. На самом деле эта процедура реализована на основе методологии конечных автоматов, о которой речь шла в начале этой главы.

Описываемая функция называется `parse_ini_file()`. Ей передается два параметра: путь кINI-файлу (который должен быть расположен на локальной машине) и параметр, определяющий необходимость включения имен разделов в выходной массив. Если второй параметр имеет значение TRUE, функция генерирует следующий многомерный массив.

```

Array
{
    [database] => Array
    {
        [DATABASE_HOST] => winsqldb
        [DATABASE_USER] => xyz
        [DATABASE_PASS] => abc
        [DATABASE_NAME] => def
    }
    [server_paths] => Array
    {
        [installation_path] => /home/ed/public_html/application
        [public_url] => http://www.example.com/application
    }
}

```

Если этот параметр принимает значение FALSE, в результате выполнения функции создается одномерный ассоциативный массив без учета заголовков разделов. Он имеет следующий вид.

```

Array
{
    [DATABASE_HOST] => winsqldb
    [DATABASE_USER] => xyz
    [DATABASE_PASS] => abc
    [DATABASE_NAME] => def
    [installation_path] => /home/ed/public_html/application
    [public_url] => http://www.example.com/application
}

```

Если вы отказываетесь от отображения заголовков разделов, необходимо обеспечить уникальность имен параметров в разных разделах.

На практике эта функция используется следующим образом.

```
$confFileArray = parse_ini_file("config.ini");
$strDatabaseHostname = $confFileArray[["database"] ["DATABASE_HOST"]];
```

Синтаксический анализ конфигурационного файла можно реализовать в отдельном классе. Тогда константы PHP будут доступны в различных частях вашего приложения.

Класс Config пакета PEAR

Хотя встроенная функция PHP позволяет очень легко выполнять синтаксический анализ конфигурационного файла, класс `Config` пакета PEAR обеспечивает более гибкий и интеллектуальный метод анализа различных типов файлов. Пакет устанавливается обычным образом, но в данном случае существует зависимость от класса `XML_Util`, который должен быть установлен первым.

```
root@genesis:~/scripts# pear install XML_Util
downloading XML_Util-0.5.2.tgz ...
Starting to download XML_Util-0.5.2.tgz (6,540 bytes)
.....done: 6,540 bytes
install ok: XML_Util 0.5.2
root@genesis:~/scripts# pear install Config
downloading Config-1.10.tgz ...
Starting to download Config-1.10.tgz (17,577 bytes)
.....done: 17,577 bytes
install ok: Config 1.10
root@genesis:~/scripts#
```

Полная документация по этому пакету содержится по адресу <http://pear.php.net/manual/en/package.configuration.php>, однако в следующих разделах будут кратко рассмотрены некоторые наиболее важные свойства.

Анализ INI-файлов с использованием класса Config

Анализ традиционного INI-файла с использованием класса `Config` не намного сложнее (но гораздо полезнее) применения встроенной функции PHP.

```
$c = new Config();
$c->parseConfig("config.ini", "IniFile");
$strDatabaseHostname = $c["DATABASE_HOST"];
```

Заметим, что метод `parseConfig()` зависит от двух параметров. Первым является путь к файлу конфигурации, а вторым — строковая константа, определяющая метод анализа конфигурационного файла.

Анализ XML-файлов с использованием класса Config

В отличие от встроенной функции PHP, класс `Config` позволяет анализировать конфигурационные файлы в формате XML.

```
$c = new Config();
$c->parseConfig("config.xml", "XML");
$strDatabaseHostname = $c["DATABASE_HOST"];
```

Запись конфигурационных файлов с использованием класса `Config`

Класс `Config` позволяет также записывать содержимое ассоциативного массива в конфигурационный файл выбранного формата.

```
$myConfig = array("foo" => "bar");
$c = new Config();
$root = &$c->parseConfig($myConfig, "PHPArray");
$c->writeConfig("my.conf", "IniFile");
```

Это чрезвычайно мощная возможность, которой можно случайно воспользоваться не по назначению (обсуждается в следующем разделе).

Рекомендации для работы с конфигурационными файлами

Конфигурационные файлы обеспечивают простой способ установки приложений без модификации кода PHP. Однако, как и другие преимущества, такой подход имеет и свои недостатки.

Не записывайте информацию в конфигурационные файлы, а только считывайте ее

Приложение не должно записывать данные в конфигурационный файл. Оно считывает информацию из него. Если в процессе работы приложения вам приходится записывать данные в конфигурационный файл, этот фрагмент лучше поместить в реляционную базу данных, например PostgreSQL.

Единственным исключением из этого правила является сценарий автоконфигурации, предназначенный для использования в отделе информационных технологий. Такой сценарий может быть написан на языке PHP, хотя для его создания прекрасно подойдет и язык оболочки.

По возможности используйте кеширование

Чтение конфигурационного файла, написанного не на языке PHP, связано с существенными затратами ресурсов. Поскольку протокол HTTP не поддерживает соединений, то анализ конфигурационного файла может потребоваться в каждом запросе. Для интенсивно используемого приложения это может привести к серьезным потерям производительности.

Поэтому по возможности необходимо реализовать некоторый механизм кеширования. Поскольку конфигурационный файл меняется не слишком часто, файл в формате INI или XML можно преобразовать в файл PHP с использованием класса `Config` пакета PEAR, а затем просто включить код PHP в свое приложение. Для определения возможных изменений в исходном конфигурационном файле необходимо предусмотреть проверку времени его последней модификации и при необходимости повторить преобразование.

Стремитесь к простоте

Страйтесь свести размер конфигурационного файла к минимуму. Если значение параметра может быть вычислено, например, с помощью объекта `$_SERVER`, его не зачем хранить в конфигурационном файле. Не стоит полагаться на системных администраторов. Задайте себе вопрос, могут ли измениться значения констант, диапазонов или пределов. Если эти значения не меняются, не включайте их в конфигурационный файл, а задайте в коде приложения.

Резюме

В этой главе вы познакомились с теорией конечных автоматов, узнали, как они определяются, чем отличаются от других абстрактных машин и где используются. В главе был рассмотрен пример конечного автомата, реализующий калькулятор для обратной польской записи. Мы подробно рассмотрели код примеров, уделив особое внимание вопросам использования класса `FSM` пакета `PEAR`.

Также речь шла о конфигурационных файлах. Вы узнали, зачем они используются, в каких форматах представляются и как анализируются.

На этом завершается часть III данной книги. Теперь читатель вооружен полным набором приемов и средств для реализации своего приложения. В части IV речь пойдет об их использовании в реальном приложении и методах управления проектами.

Часть IV

Учебный пример: автоматизация работы торгового предприятия

В ЭТОЙ ЧАСТИ...

Глава 18. Знакомство с проектом

Глава 19. Методологии управления проектами

Глава 20. Проектирование системы

Глава 21. Архитектура системы

Глава 22. Разработка средства автоматизации торговли

Глава 23. Обеспечение качества

Глава 24. Развёртывание

Глава 25. Разработка надёжной системы генерации отчетов

Глава 26. Что дальше

18

Знакомство с проектом

Разрабатывать программное обеспечение тяжело. Если бы это было легко, то читатель не утруждал бы себя изучением этой книги и нас бы уже давно поработили машины.

Но почему разработка программного обеспечения — трудный процесс? Для нее не нужно покупать ни красок, ни холста, и она требует от разработчика минимальных физических усилий. Придется начинать с чистого листа, и главная проблема — не загнать себя в процессе разработки в глухой угол. По сути, главный враг — ваш собственный ум.

Другое ограничение — это время. Несмотря на то, что технически можно реализовать почти все, человек смертен и физически не может за две недели создать мощную и гибкую программу, например операционную систему GNU/Linux.

А для кого пишется программное обеспечение? Для себя? Для друзей? Обычно оно разрабатывается для клиента. В этой главе в качестве примера клиента используется воображаемая фирма “Widget World”. В разрабатываемом программном обеспечении требуется воплотить довольно типичный сценарий, который может оказаться полезным и для других коммерческих или государственных компаний.

В этой главе предполагается, что вы являетесь обычным разработчиком, работаете над типичным проектом в обычной компании с традиционными клиентами. Однако для простоты в рассматриваемом примере проекта приняты некоторые предположения.

- Вы разрабатываете новый продукт. (Более общим сценарием была бы модификация существующей системы.)
- Вы работаете самостоятельно, поскольку самостоятельно читаете этот текст. (В действительности, вы, скорее всего, будете тесно сотрудничать с другими разработчиками.)

Компания Widget World

Фирма Widget World разрабатывает и продает стандартные элементы графического интерфейса. Сотрудники компании прочесывают весь земной шар, участвуют в конференциях по интерфейсам пользователя, ведут переговоры с бизнес-партнерами и проводят презентации своего товара.

Компания Widget World — жертва своего успеха. Она не планировала столь быстрого роста, поэтому ей не удается информировать сотрудников о возможностях сбыта

и новейших технологиях в сфере графических интерфейсных элементов, наладить обратную связь с персоналом, чтобы извлекать максимальную выгоду, основываясь на тенденциях развития стандартных графических интерфейсных элементов.

Компания Widget World не может нанять достаточное количество молодых сотрудников для работы на телефоне и поддержки производственного процесса. Директор фирмы понимает, что Widget World должна работать эффективнее. Поэтому, рассмотрев коммерческие предложения, он решил заказать программное обеспечение под название Widget World Sales Force Automation Tool (средство автоматизации работы Widget World), или сокращенно WW-SFAT.

Хотя фирма работает не достаточно эффективно, в ней есть отдел маркетинга, отвечающий за прогнозирование последних тенденций и составление “дорожной карты” развития фирмы.

Специалисты по маркетингу способствуют деятельности руководителей региональных отделений, снабжая менеджеров филиалов новейшей информацией. Маркетологи сообщают им информацию о ценах на стандартные графические элементы интерфейсов в компаниях-конкурентах.

Менеджеры региональных отделений направляют свой персонал, снабжая его свое временной информацией. Они также устанавливают нормы выработки, выписывают премии и всячески добиваются максимального “усердия” продавцов в том, чтобы поместить стандартные графические интерфейсные элементы Widget на каждый рабочий стол.

Менеджер по разработке средства автоматизации для компании Widget World — это специалист, получающий информацию от маркетологов и региональных менеджеров и возглавляющий процесс создания и поддержки программной системы WW-SFAT, призванной повысить эффективность работы фирмы. Менеджер WW-SFAT обладает полной информацией о бизнес-процессах на предприятии, что является решающим фактором для успеха проекта, потому что ограничивает число требуемых встреч и документов. Менеджер WW-SFAT также должен тесно сотрудничать с разработчиком, который не владеет информацией об особенностях бизнес-процессов данного предприятия. На рис. 18.1 показана используемая в настоящее время форма мониторинга.

Вот несколько вопросов, которые возникают у разработчика, занимающегося автоматизацией ввода информации.

1. “В каком формате выводить время: в 12- или 24-часовом?”
2. “Нужно ли учитывать часовые пояса?”
3. “Когда сотрудник X должен представлять свой отчет?”
4. “Что происходит, если отчет от X не поступает до указанного срока?”
5. “Как покрываются затраты?”
6. “Как часто меняются различные тарифы?”
7. “Нужно ли хранить информацию о тарифах? Приходится ли со временем обращаться к прежним тарифам?”

Компания Widget World изнутри

В следующем разделе подробно описываются различные уровни работы компании Widget World. Это описание включает в себя технический, финансовый и политический уровни, а также роль разрабатываемого программного обеспечения в автоматизации бизнес-процессов.

Еженедельный отчет о контактах компании **Widget World**

Имя сотрудника: _____ Отдел: _____

Номер сотрудника: _____ Дата: _____

Важные дистрибуторы и покупатели:

Компания: _____ Контактное лицо: _____

Отзыв: _____ Город: _____

Пожелания _____ Страна: _____

Полученные результаты: _____

Компания: _____ Контактное лицо: _____

Отзыв: _____ Город: _____

Пожелания _____ Страна: _____

Полученные результаты: _____

Рис. 18.1.

Технический уровень

При внимательном рассмотрении становится ясно, что информация должна передаваться от торгового персонала вверх по цепочке до директора, а затем вниз. В данном случае информационный поток является двунаправленным.

1. Отделу маркетинга требуется итоговая информация по продажам в регионе.
2. Региональные менеджеры должны отслеживать работу каждого сотрудника.

Финансовый уровень

Этот проект призван сэкономить деньги Widget World за счет повышения эффективности работы персонала. Именно это оправдывает вложение средств в данный

проект. На данный момент не ясно, что принесет более ощутимую экономию — снабжение сотрудников нижнего уровня более развернутой информацией или получение больших объемов информации от них.

Политический уровень

Один из региональных менеджеров компании втайне не доверяет отделу маркетинга и думает, что подаваемая им информация будет использована для поиска аргументов в пользу его преждевременного увольнения. Торговый персонал приветствует электронную отчетность. Она позволит продемонстрировать, как напряженно работают рядовые сотрудники, заставит региональных менеджеров честно оценивать труд персонала и позволит избежать кумовства. Специалисты по маркетингу хотят иметь надежные данные для публикаций об успехах компании в прессе и показывать их своим партнерам по гольфу.

Разработчик

Разработчик получил этот проект, ответив на рекламное объявление в местной газете, которое гласило: “Компании Widget World нужен разработчик средства автоматизации”. Разработчик связался с менеджером проекта, прошел собеседование (по части глубокого знания PHP 5) и прибыл на работу, исполненный желания начать карьеру в области разработки программного обеспечения.

Действительно ли дело в технологии

Средство автоматизации работы компании Widget World должно предоставлять примерно следующую информацию.

- Отчеты о контактах торгового персонала
- Отчеты о продажах
- Списки контактов менеджеров, распространяемые по электронной почте, по факсу и через Web
- Рейтинги продаж компаний
- Маркетинговые отчеты и полноцветные красочные диаграммы
- Финансовые отчеты
- Обучающие материалы

На самом деле это требования к типичному проекту, для создания которого необходимо использовать наилучшие технические приемы разработки, позволяющие удовлетворить запросы клиента. Хотя в данном случае главным потребителем является менеджер проекта, он не живет в вакууме, и результатами разработки в определенной степени будут пользоваться все сотрудники фирмы.

Подход к разработке

В процессе разработки будут использованы некоторые элементы “экстремального программирования” (eXtreme Programming). Не стоит воспринимать этот термин буквально. Экстремальное программирование — это набор методов быстрой разработки,

который получил высокую оценку специалистов. Этот процесс включает следующие элементы.

- Планирование (planning) — определение масштабов проекта с учетом приоритетов и технических оценок, предполагающее постоянную корректировку.
- Выпуск промежуточных версий (small releases) — быстрое получение рабочей версии системы, передача ее пользователям и выпуск новых версий в кратчайшие сроки.
- Простота проектного решения (simple design) — по возможности необходимо поддерживать простоту проектного решения. Все неоправданные сложности исключаются.
- Тестирование (testing) — разработчики постоянно пишут тестовые модули для проверки работы системы. Клиенты тоже участвуют в тестировании, чтобы подтвердить наличие необходимой функциональности.
- Пересмотр кода (refactoring) — реструктуризация кода без модификации поведения системы для удобства чтения, устранения повторов и упрощения программного кода.
- Постоянная связь с клиентом (on-site customer) — менеджер системы WW-SFAT является членом команды разработчиков и постоянно отвечает на вопросы, связанные с бизнес-процессами на предприятии.

Эти принципы позволяют управлять сложностью и масштабом как самого проекта, так и разрабатываемого программного обеспечения. Зависимость стоимости ошибок от стадии разработки приведена на рис. 18.2.

Стоимость изменений

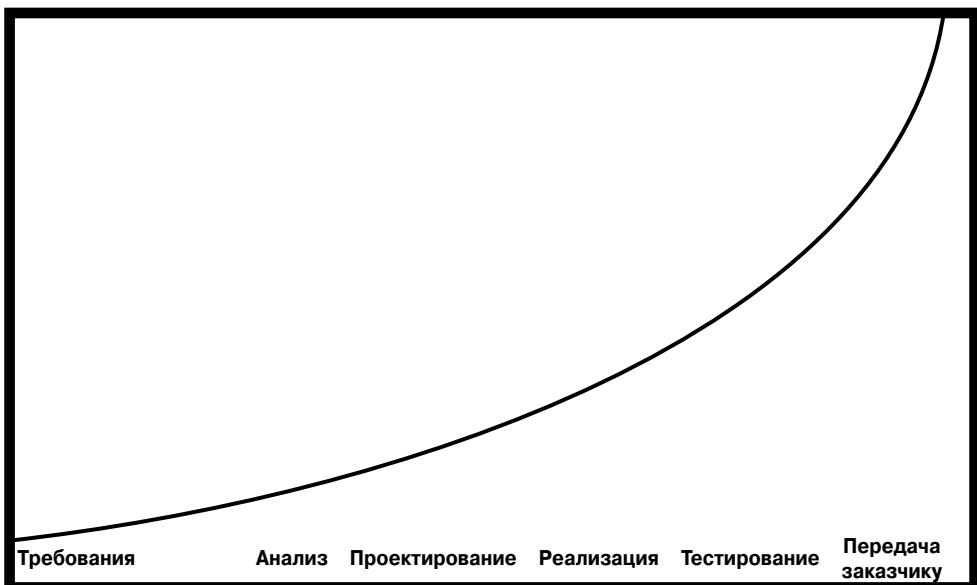


Рис. 18.2.

Рассматривая управление сложностью под другим углом зрения, мы радикально изменим эту кривую (рис. 18.3).

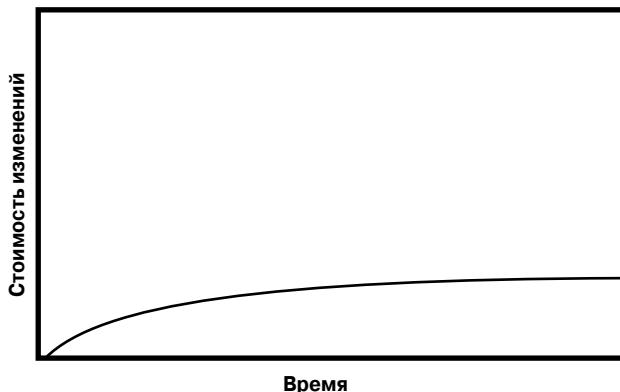


Рис. 18.3.

Реализация необходимых свойств на разных этапах разработки — мощный способ управления сложностью всей системы. Стоимость добавления новых свойств со временем будет не намного выше, чем стоимость их добавления в данный момент. Это означает, что во избежание лишних усилий по созданию ненужных элементов системы не надо гадать о том, что может понадобиться в будущем.

Традиционный подход к сложности предполагает следующие рассуждения. Добавление нового свойства сейчас обойдется в \$5.00, а его добавление в будущем будет стоить \$100.00. Поэтому надежнее добавить его сейчас.

Но если удается сделать так, что стоимость добавления нового свойства сейчас составляет \$5.00, а позднее — \$6.00, то традиционный подход к разработке оказывается неэффективным. Отложите добавление нового элемента до лучших времен. Ваша система останется простой, и вы обойдетесь только теми свойствами, которые активно используются с самого начала.

Что это означает

Рассмотрим традиционный каскадный процесс разработки программного обеспечения, схема которого показана на рис. 18.4. Реализация такого процесса требует следующих временных затрат.

- Две недели на определение требований
- Две недели на их анализ и обсуждение
- Две недели на проектирование системы
- Восемь недель на написание кода
- Четыре недели на тестирование

Хотя каскадная модель процесса разработки жестко регламентирована, она имеет несколько недостатков.

- Ей не хватает гибкости. Предполагается, что проектирование будет правильно выполнено с первого раза. Такая модель не предполагает обратной связи.

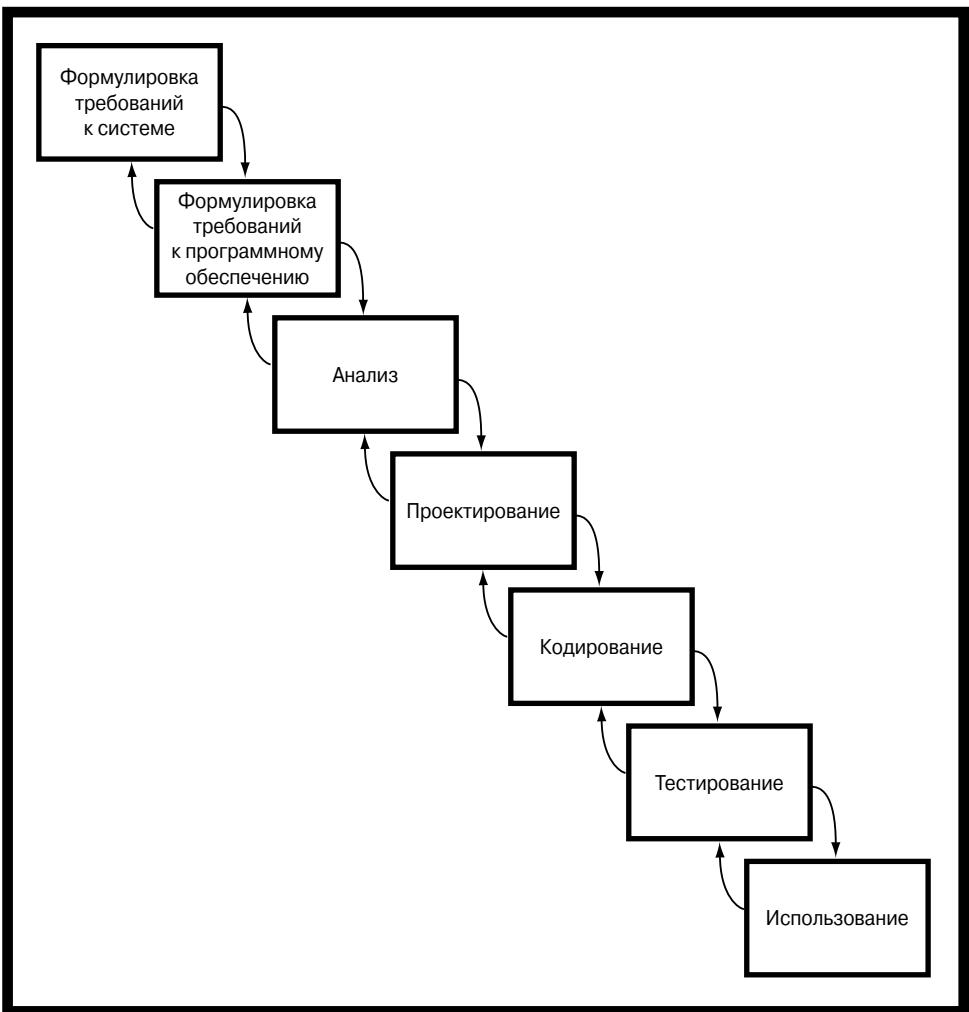


Рис. 18.4.

- Трудно заранее указать все необходимые свойства системы, потому что требования наверняка будут меняться.

Такой подход не имеет шансов на успех, поэтому целесообразно последовать принципам экстремального программирования. Это позволит разработать маленькую, гибкую систему, которая будет тщательно тестироваться по ходу написания кода. Разработчик будет тесно сотрудничать с менеджером проекта WW-SFAT и получать ответы на все заданные вопросы. У менеджера проекта есть огромный опыт работы в своей сфере деятельности, и он день за днем будет использовать систему.

Это также означает, что ошибки будут исправляться по мере их обнаружения, а не нужные функциональные возможности будут удаляться. Разработчик будет отслеживать дублирование и безжалостно его устраниять. Благодаря регулярному пересмотру кода система будет оставаться простой и мощной.

И наконец, разработчик будет выполнять модульное тестирование. Тестируя каждый модуль, вы постоянно будете иметь полностью проверенную систему. Это заставит разработчика внимательно относиться к сбоям и симптомам, возникающим вследствие наличия дефектов в программе. Если система полностью протестирована, разработчик может мгновенно выявить ошибку в новом коде.

Наличие хорошо написанного пакета для тестирования позволяет легко модифицировать свойства системы. Тестовый пакет добавляет уверенности разработчику, давая возможность расширять систему в случае необходимости.

Это только краткое введение в экстремальное программирование, которое является одной из самых быстрых методологий разработки программного обеспечения. Ничего страшного, если читателю понятен не весь материал данной главы. В последующих главах вы увидите все преимущества описанной методологии и поймете, что с использованием описанных здесь методов можно не только быстро разрабатывать небольшие гибкие программы, но и планировать процесс разработки, позволяя клиенту принимать в нем активное участие.

Технология

Хотя в этой главе речь шла о том, что не стоит создавать элементы заранее, есть некоторые технологические особенности, о которых нужно сказать с самого начала. К ним относятся следующие.

- ❑ Диаграммы классов и последовательности UML можно рассматривать как моментальные снимки системы.
- ❑ Шаблоны проектирования. Их необходимо эффективно использовать и реализовывать в коде приложения.
- ❑ Коллекции и итераторы. Нужно понимать, где имеет смысл их использовать, а где нет.
- ❑ Диспетчеры позволяют почувствовать вкус программирования на основе обработки событий.
- ❑ Протокол SOAP можно использовать для обмена данными.

Резюме

В этой главе описана компания Widget World и ее проблемы. Теперь можно приступить к использованию описанных здесь технических приемов разработки.

Вместо традиционного, жестко регламентированного каскадного процесса разработки можно воспользоваться более гибким походом, который выдвигает на первый план частое появление новых версий продукта, простоту проектного решения, тщательное тестирование основного кода, постоянное улучшение кода и обратную связь с клиентом.

На основе этого подхода удается создавать приложения, первые версии которых можно полноценно использовать уже через несколько дней и недель, а не через месяцы и годы.

19

Методологии управления проектами

Возможно, как профессиональному в области разработки Web-приложений, вам интересно узнать, может ли системный архитектор эффективно управлять проектом. На этот вопрос можно дать лишь один ответ: да! Конечно, при этом очень пригодится некоторый практический опыт.

Даже если вы никогда не занимались управлением проектами, вам будет очень полезно узнать, кто же такой менеджер проекта, с какими проблемами он сталкивается, а также какие вопросы ему приходится решать каждый день. Подобная информация позволит вам глубже разобраться с вопросами проектирования архитектуры, а также оценить, какие навыки требуются для того, чтобы успешно довести проект от начала до конца.

Если вы когда-нибудь возглавите группу молодых и инициативных разработчиков, то материал данной и последующих глав окажется весьма полезным.

В этой главе обсуждаются вопросы сбора требований и их представления в виде сжатого единого документа, а также проблемы разработки спецификации и плана выполнения проекта. Вы научитесь критически оценивать и отбирать основных специалистов, а также познакомитесь с основными принципами руководства ими на протяжении всего жизненного цикла программной системы. Кроме того, здесь будут рассмотрены некоторые подходы к программированию, вопросы оценки их применимости в конкретном проекте, а также два ключевых подхода к организации работ по разработке приложений. И наконец, вы узнаете, как обеспечить эффективное взаимодействие с заказчиком.

Следует заметить, что основной материал этой главы напрямую совсем не связан с языком PHP, тем не менее все эти сведения являются важными для любого PHP-проекта, а также для проектов, связанных с технологиями ASP, Java и любыми другими языками программирования. Авторы решили посвятить рассмотрению этих вопросов часть книги, поскольку при выполнении больших проектов эта информация может оказаться весьма кстати.

Выполните домашнее задание

Чаще всего проект начинается с того, что менеджер сообщает одному из старших разработчиков о том, что для их команды появилась новая работа. Обычно такие разговоры проходят в неформальной обстановке в комнате отдыха или еще где-нибудь в офисе. Однако на практике основная идея зарождается гораздо раньше. Очень важно отслеживать истоки появления любой идеи. Методология управления проектом будет успешной только в том случае, если она будет применяться от начала и до конца проекта.

Перед формулировкой краткого описания проекта нужно получить ответы на несколько важных вопросов. Именно в этом и заключается домашнее задание, которое стоит выполнить. Эти ответы и лягут в основу стратегии, которая будет отражена в кратком описании проекта. Первый вопрос, на который следует получить ответ, звучит следующим образом: *почему?*

Почему возникает новый проект

Этот простой вопрос оказывает огромное влияние на все последующие действия, поскольку способ реализации проекта от начала и до конца должен определяться исключительно с учетом главной целевой функции.

Причины начала нового проекта, связанного с разработкой Web-приложения, могут быть следующие.

- Начало нового бизнеса, для которого разработка приложения и является основным видом деятельности.
- Замена существующего приложения, которое функционирует не совсем правильно или больше не удовлетворяет реальным требованиям.
- Преодоление проблем с существующим поставщиком услуг, возникших из-за того, что он не вложился в сроки, выделенный бюджет или не смог удовлетворить выдвинутые требования (в некоторых случаях одновременно могут возникнуть все три проблемы).

Естественно, новый проект может инициироваться и по другим причинам, однако они наверняка окажутся не столь критичными, как приведенные выше. Например, может понадобиться реализовать задумку вице-президента вашей компании или ее коммерческого директора. В любом случае выявление причины начала проекта является задачей № 1.

Для кого проект предназначен

Ответить на этот вопрос не всегда легко, как может показаться на первый взгляд. На самом примитивном уровне проект может быть предназначен для другого отдела вашей компании или организации, нового или уже существующего клиента. Помните о том, что нужно определить, кто будет потребителем результатов проекта, а точнее, кто на самом деле является его заказчиком. Если муж идет в магазин для покупки жене лака для ногтей, то он не является реальным покупателем, а представляет собой лишь носителя поручения реального клиента.

Аналогично, ваш заказчик может поручить выполнить работу для третьих лиц, перед которым он сам несет ответственность. Этих конечных пользователей еще называют экспертами в предметной области. Вы можете игнорировать их требования на свой собственный страх и риск.

Скорее всего, конечным пользователем разрабатываемого продукта будет наиболее опытный специалист из компании-заказчика. Его требования к продукту могут существенно отличаться от потребностей, сформулированных другими заинтересованными лицами.

Определение ролей и целей

Предположим, вы являетесь руководителем технического отдела компании среднего размера и директором по персоналу поручил вам построить инфраструктуру всей компании. Под вашим руководством работают как постоянные служащие, так и специалисты по контракту. Что можно сказать о целях, которые ассоциируются с каждой ролью (вами, пользователями, экспертами в предметной области)?

Если посмотреть на ситуацию с непредвзятой точки зрения, можно выделить следующие цели.

- Вы как можно скорее и с минимальными финансовыми затратами хотите предоставить высококачественное и эффективное решение, чтобы заслужить хорошую репутацию.
- Директор по персоналу хочет, чтобы вы как можно скорее и с минимальными финансовыми затратами предоставили высококачественное и эффективное решение, чтобы заслужить хорошую репутацию.
- Обычные пользователи (эксперты в предметной области) хотят, чтобы вы как можно скорее и с минимальными финансовыми затратами предоставили высококачественное и эффективное решение, чтобы они могли его использовать для решения повседневных задач.

Обратите внимание, что единственная “чистая” мотивация есть только у экспертов в предметной области. Главный нюанс заключается в том, что должны быть достигнуты все три цели. Достигнуть первых двух очень легко. В то же время стремление к третьей цели позволит реализовать и первые две.

Перераспределение ролей и получение преимуществ

Рассмотрим следующую ситуацию. Заказчик является просто “средством связи” с экспертами в предметной области независимо от того, какую информацию они предоставляют вам напрямую. Хотя заказчик и может выразить удовлетворение от вашей работы, если полученные результаты не понравятся экспертам, то их неудовольствие сразу же отразится на нем, а, следовательно, и на вас. Заказчику очень важно представлять интересы экспертов в предметной области с самого начала проекта.

Хороший заказчик должен найти общий язык с экспертами в предметной области, проанализировать их реальные нужды, сформулировать требования и сообщить о них вам. Однако на практике вам придется его подталкивать и обманными путями склонять к предоставлению правдивых и реалистичных требований экспертов.

Если вы чувствуете, что заказчик недостаточно точно выражает интересы экспертов в предметной области, очень важно повторно инициировать процесс обмена мнениями специалистов, связанных с данным проектом. Это можно осуществить следующим образом.

- Провести совещание на основе метода “мозгового штурма”.
- Воспользоваться коробкой для анонимных предложений.
- Организовать рабочее совещание и обсудить потребности экспертов в предметной области.

Если оказалось, что с начала проекта роли не распределены, вам необходимо принять в этом процессе самое непосредственное участие. Тогда при получении краткого описания проекта вы будете уверены в том, что сформулированные требования удовлетворяют реальным потребностям заинтересованных лиц.

Какую предысторию имеет проект

Очень важно также знать о том, связана ли с проектом какая-нибудь предыстория. Новый проект — это всегда очень увлекательный вид деятельности. Однако если копнуть поглубже, он может оказаться не таким новым, как это выглядит на первый взгляд. Хорошая осведомленность о предыстории проекта позволит лучше учесть его определенные аспекты и выработать соответствующую стратегию.

Вообще говоря, у проекта может быть длинная предыстория лишь в том случае, если ранее уже предпринималась попытка его реализации. Если это так, вы должны сразу же об этом узнать как можно больше независимо от того, как давно это работа выполнялась.

- **Бывший сотрудник.** Нужно быть очень внимательным, если раньше данным проектом руководил сотрудник, который был уволен или переведен на другую должность в той же организации. Почти всегда заказчик считает, что бывший подчиненный не смог выполнить поставленную перед ним задачу. Основная задача заключается в том, чтобы получить общее представление о сильных и слабых сторонах нового руководителя, с которым вы и должны будете общаться. Этот вопрос более подробно будет обсуждаться в этой главе ниже.
- **Бывший разработчик.** Если вы взяли на себя ответственность за проект после другого менеджера (неважно, работает он в коммерческой компании или является сотрудником вашей организации), очень важно разобраться, почему заказчик отказался от его услуг. При необходимости можно поговорить с другими заказчиками данного специалиста и учесть всю полезную информацию, которую вы от них получите. Заказчик очень хочет доверять вам, так что позаботьтесь о том, чтобы у него были на это основания.
- **Сложности при работе с внутренним персоналом.** Если выполнение проекта было поручено вам какой-либо компанией, то при работе с сотрудниками ее ИТ-отдела могут возникнуть определенные трудности. Иногда сложности возникают еще до запуска проекта, когда только предполагается, что работы будут начаты. Специалистов ИТ-отдела лучше всего вовлечь в этот процесс как можно раньше. Позвоните им еще при разработке технической спецификации. Такой шаг наверняка будет хорошо воспринят, и вы даже сможете получить от них существенную помощь.

Каковы исходные условия

Следует как можно раньше осознать исходные условия проекта, поскольку они могут существенно повлиять на возможные пути его реализации. Типичными начальными условиями могут быть следующие.

- **Платформа.** Хочет ли заказчик, чтобы программный продукт был построен на основе ASP, Cold Fusion или другой технологии, а не языка PHP (как было бы для вас лучше всего)?

- Развертывание.** Требуется ли заказчику, чтобы приложение было развернуто на существующих серверах? Если да, то нужно оценить, можно ли будет при необходимости изменить их конфигурацию.
- Временные рамки.** У заказчика могут быть пожелания по срокам выполнения проекта, которые жестко связаны с его собственными целями или временными ограничениями.
- Бюджет.** Составлен ли бюджет? Если да, насколько он реален?

Очень важно узнать ответы на подобные вопросы до того момента, как будет разработан окончательный план реализации проекта. Если какие-либо из вопросов остались открытыми, перейти к следующему шагу вы вряд ли сможете.

Разработка описания проекта

После выполнения домашнего задания вы должны получить формальное описание проекта. При этом самая важная часть работы связана с созданием самого документа. К этому моменту вы уже должны иметь в своем распоряжении достаточно информации о проекте, однако его краткое описание обеспечит возможность вам и вашей команде разработчиков максимально эффективно выполнять свои обязанности.

Сначала краткое описание можно представить и в менее традиционной словесной форме или в виде письма. Однако в результате заказчик почти всегда хочет получить согласованное формальное описание проекта. Его не стоит представлять в виде отдельного документа, а лучше включить как часть в презентационные материалы, которые более подробно будут рассматриваться ниже.

Получение от заказчика четко сформулированных требований — это одна из самых трудных задач менеджера проекта. Еще хуже то, что сам заказчик, как правило, не может разобраться, когда неправильные предложения вносятся из-за отсутствия информации, а когда они возникают из-за каких-либо заблуждений. Для того чтобы избежать этой проблемы, убедитесь, что все предположения, содержащиеся в презентационных материалах, основаны на самой точной и полной информации, которая вам доступна. Для этого на самых ранних стадиях следует досконально изучить все детали бизнес-требований.

Перед тем как приступить к краткому описанию проекта, попробуйте поговорить с заказчиком по телефону или, что еще лучше, организовать специальный семинар.

Если при удачном стечении обстоятельств вам удалось договориться о телефонном разговоре или проведении семинара, в процессе этого общения с заказчиком очень важно задать правильные вопросы. Сосредоточьтесь на основных вопросах, которые описываются в этой книге. После получения всей необходимой информации можно приступать к созданию краткого описания проекта.

Формулировка бизнес-требований

На этом шаге нужно определиться с основными требованиями, которые предъявляются к разрабатываемому приложению. При этом сосредоточьтесь на вопросах, которые позволяют лучше разобраться с целями экспертов в предметной области, а не на потенциальном решении, к которому стремится заказчик.

Если предлагаемое вами решение в точности соответствует целям экспертов, то не имеет значения, насколько оно оказалось близким к ожиданиям самого заказчика. Такое соответствие формирует основу всего предлагаемого решения и должно играть ключевую роль в презентационных материалах.

Задавайте наводящие вопросы. Страйтесь больше слушать, чем говорить. Спонтанные высказывания могут оказаться весьма полезными для осознания реальных потребностей заинтересованных лиц, даже если они возникли на подсознательном уровне. Попробуйте добиться требуемого результата путем комбинирования технического и логистического подходов. Например, при формулировке бизнес-требований будут полезны следующие вопросы.

- Какие другие решения вы рассматривали?
- Почему вы от них отказались?
- Кто будет конечными пользователями системы?
- Как вы управляете данным бизнес-процессом без использования программной системы?
- Какие преимущества вы хотите получить от ее использования?
- Как вы хотели бы вернуть вложенные средства?
- Какие при этом будут использоваться критерии оценки?
- Каковы три основных требования к разработчику, который должен реализовать данную систему?
- Какие альтернативы вы рассматривали?
- Насколько вы были удовлетворены полученными предложениями?
- Накладываются ли на поиск разработчика временные ограничения? Если да, то чем они обусловлены?
- Какие специалисты должны быть вовлечены в процесс принятия решений и каковы должны быть их роли?
- Какие предложения вы можете сформулировать по дальнейшему развитию проекта?
- Учитывали ли их другие разработчики?

Стоит повторить еще раз: не опасайтесь многочисленных высказываний заказчика. В то же время пострайтесь не допустить влияния на ваше мнение. Задавайте наводящие вопросы, чтобы получить на 10% больше информации.

Например, между заказчиком и разработчиком может возникнуть следующий диалог.

Разработчик. У вас есть временные ограничения на выбор разработчика?

Заказчик. Конечно. Мы должны принять решение к тридцатому числу.

Разработчик. Тридцатое число этого месяца уже очень скоро. А чем это обусловлено?

Заказчик. Со следующего месяца у нас стартует большая маркетинговая кампания.

Разработчик. Ясно. Планируется ли использовать в ней разрабатываемый программный продукт?

Заказчик. Да. Если к тому времени уже будут получены определенные результаты, то это станет значимым событием этой кампании.

Ниже приведен аналогичный диалог, но без дополнительных вопросов.

Разработчик. У вас есть временные ограничения на выбор разработчика?

Заказчик. Конечно. Мы должны принять решение к тридцатому числу.

Разработчик. Спасибо.

Обратите внимание, сколько полезной информации можно получить благодаря дополнительным вопросам.

- У заказчика в следующем месяце начинается маркетинговая кампания.
- Клиент хочет представить незавершенный программный продукт потенциальным покупателям, как важный элемент маркетинговой кампании.

Подобная информация не является секретом, и заказчик не скрывает ее умышленно. Просто ему в голову не приходит мысль о том, что ее нужно сообщить. Дополнительная информация является для вас очень полезной. Так что не стоит ее недооценивать. Из рассмотренного примера можно сделать следующий вывод: не бойтесь проявлять интерес к высказываниям заказчика и задавать ему дополнительные вопросы, которые помогут “выудить” из него дополнительную информацию. Возможно, это позволит получить преимущество над вашими соперниками и выиграть тендер на получение заказа.

Определение границ

Выше речь шла о формулировке достаточно широких бизнес-требований к системе, которые отражали бы потребности заинтересованных лиц. Однако зачастую нужно более четко определить их *границы* (scope). Что же означает этот термин? Путем задания границ можно определить, насколько далеко нужно продвинуться в процессе выполнения проекта, чтобы решить поставленную задачу и удовлетворить все сформулированные требования.

Иногда возникает ситуация, когда бизнес-требования изменяются в процессе выполнения проекта. В результате может увеличиться объем работ, что, к сожалению, никак не повлияет на размер заработной платы. Однако этого можно избежать, если задать границы еще на ранних стадиях выполнения проекта. Созданные спецификации, по существу, являются формальным описанием этих границ, которому должны соответствовать определенные функциональные компоненты, проектируемые в процессе разработки программной системы.

Границы почти всегда однозначно зависят от накладываемых временных ограничений и бюджета проекта. Однако если определить границы различных стадий проекта, можно существенно облегчить его менеджмент. Если вы предложите заказчику две фазы развития проекта, то в девяти из десяти случаев это окажется очень полезным, поскольку он сразу же сформулирует, какую функциональность можно перенести на вторую фазу, а какая часть функций должна быть реализована с самого начала. Не оставляйте решение этой задачи на более поздний срок. Только в этом случае границы требований будут определены достаточно четко.

Если заказчика не устраивает несколько стадий выполнения проекта, придется воспользоваться более прямолинейным подходом. Объясните ему, что единственными ограничениями его требований являются деньги и время, а также то, что выделение наиболее важных компонентов системы позволит облегчить жизнь и оценить ваши предложения с точки зрения временных и финансовых затрат. Если других претендентов на выполнение проекта нет, объясните заказчику, что определение границ требований позволит впоследствии сэкономить много времени и обеспечит более динамичное развитие проекта.

График выполнения работ

Определение временных рамок выполнения работ — это гораздо больше, чем просто фиксация момента времени, к которому будет разработана система и будут удовлетворены все требования заказчика. Даже на самой ранней стадии проекта очень важно согласовать с заказчиком основные вехи (milestone) и сроки их достижения. Другими словами, необходимо оценить трудоемкость и длительность следующих видов деятельности.

- Предоставление предложений по разработке системы.
- Принятие решения о выборе разработчиков (при необходимости).
- Предоставление заказчику спецификации.
- Утверждение заказчиком разработанной спецификации. При наличии замечаний они должны быть обсуждены с заказчиком, а в спецификацию должны быть внесены соответствующие исправления.
- Завершение разработки спецификации с учетом всех внесенных замечаний.
- Начальное утверждение проектных решений.
- Утверждение проектных решений с внесенными замечаниями.
- Начало разработки системы.
- Получение бета-версии системы.
- Получение окончательной версии системы.
- Начало тестирования системы.
- Выпуск конечного программного продукта.

Безусловно, процесс согласования с заказчиком дат и соответствующих этапов может оказаться слишком трудоемким. В этом случае можно самостоятельно выделить наиболее важные этапы и соответствующие сроки их выполнения, а затем предоставить их заказчику. Такой подход может оказать на него положительный эффект. Рассмотрим следующий диалог.

Разработчик. Когда нужно получить окончательную версию системы?

Заказчик. Она должна быть готова и запущена в производственном режиме 1 июня.

Разработчик. Тогда нам нужно завершить все работы до 22 мая. Несколько оставшихся дней можно будет посвятить внесению последних модификаций и более точной настройке.

В процессе обсуждения не забудьте скрупулезно фиксировать все вехи. Кроме того, постарайтесь, чтобы у заказчика не возникло нереалистичных ожиданий, как в следующем примере.

Разработчик. Когда нужно получить окончательную версию системы?

Заказчик. Она должна быть готова и запущена в производственном режиме 1 мая.

Разработчик. Я думаю, что запланированный объем работ завершить к указанному сроку будет затруднительно. Более реально все закончить до 22 мая. Тогда систему можно будет запустить в ближайшие несколько дней. Таким образом, у нас есть две альтернативы: либо сузить функциональные требования к системе, либо немного сдвинуть срок сдачи ее окончательной версии.

Заказчик. Хорошо, мы подождем до 22 мая.

Обратите внимание, каким образом заказчик узнает о том, что его пожелания нереалистичны. Ему не просто сообщается, что так должно быть. (Тактика “мы знаем лучше” может его задеть и обидеть.) Напротив, ему предлагается несколько вариантов, как в предыдущем примере. Зачастую такой подход позволяет увидеть, что дела не так уж и плохи, и выбрать наиболее подходящее решение.

Всегда старайтесь сообщить заказчику о нереалистичных данных. Заказчик всегда ожидает получить все результаты сразу. Если что-либо нельзя завершить к требуемому сроку, об этом нужно сразу же ему сообщить. В противном случае, если в процессе обсуждения ваших предложений какие-либо даты не были упомянуты, вы рискуете потерять заказ, особенно если все другие участники проекта были об этом уведомлены. По существу, все, что не было четко оговорено, будет восприниматься как неявная договоренность. Так что будьте внимательны.

Бюджет

Обсуждение бюджета может оказаться камнем преткновения. Даже если вы разрабатываете приложение для своей компании, вам все равно придется задуматься о связанных с этим накладных расходах. Во многих больших компаниях в статью расходов включается и стоимость работ собственных специалистов. Так что обсуждение бюджета и соответствующие переговоры внутри компании могут оказаться такими же напряженными, как и определение стоимости услуг внешних исполнителей.

Не обсуждайте финансовую смету слишком рано. Это может сыграть против вас. Безусловно, заказчик ожидает вопросов на эту тему, однако последовательность задаваемых вами вопросов поможет ему сформировать мнение о ваших приоритетах как разработчика. Финансирование лучше всего обсуждать после того, как будут определены основные функциональные требования к системе, поскольку они во многом обусловлены временными и финансовыми ограничениями.

Обсуждение стоимости работ целесообразно вести в следующей форме: “Когда я еще раз проанализировал нашу реализацию аналогичных решений для компании Асте, оказалось, что она была готова вложить в разработку от \$25000 до \$45000. Вы согласны с такой приблизительной стоимостью данного проекта?”.

Во-первых, обратите внимание, что вопрос о бюджете проекта напрямую не был задан. Напротив, в разговоре с заказчиком просто было упомянуто о том, что:

- в прошлом вы выполняли аналогичные проекты — это *положительный факт*;
- раньше вы работали с такими компаниями, как Асте — это тоже *положительный факт*;
- для реализации аналогичных проектов эти компании выделяли от \$25000 до \$45000 — достаточно *нейтральная* информация.

Смещение акцента в сторону положительных моментов позволяет избежать напряжения, которое, скорее всего, возникло бы при задании вопроса о бюджете напрямую.

Во-вторых, заказчику был предложен достаточно широкий примерной стоимости проекта. Такой подход нужно использовать всегда. В данный момент разговор о слишком определенной стоимости будущих работ может свидетельствовать лишь о вашем неуважении к заказчику и его требованиям к системе, которые еще до конца не сформулированы. В свою очередь упоминание о диапазоне стоимости будет указывать заказчику на то, что сейчас вы основное внимание сосредоточили на том, что требуется сделать. Нижняя граница должна соответствовать абсолютному минимуму,

которого будет достаточно для выполнения проекта, а верхняя граница ценового диапазона должна быть на 40–50% больше этой суммы. При таком подходе можно быть уверенным в том, что вы получите выгодную работу. (Конечно, если заказчик согласится на ваши предложения.) Насколько данный проект окажется выгодным на самом деле — покажет время.

И наконец, при обсуждении бюджета была использована фраза о том, что компания “была готова вложить в разработку”, а не фраза “за выполнение проекта мы потребовали”. Другими словами, компании были готовы финансировать разработку системы еще *перед* обсуждением с вами, а не в результате переговоров.

Однако все, о чём говорилось выше, не позволяет ответить на вопрос: “Какой бюджет проекта?”, а лишь позволяет показать заказчику, что данный проект является важным и востребованным.

Если же после подобных разговоров заказчик не хочет продолжать обсуждение бюджета, он обязательно об этом скажет. Конечно, такое развитие событий нельзя назвать очень удачным. Тем не менее вы можете быстро оценить, было ли время потрачено впустую. Рассмотрим два примера. В обоих случаях для разработки системы разработчику требуется не менее \$15000, в то время как заказчик готов выделить \$5000. Вот первый сценарий.

Разработчик. Вы уже обдумывали бюджет?

Заказчик. Мне бы не хотелось называть какие-либо цифры. Мой прошлый опыт общения с разработчиками свидетельствует о том, что это не имеет особого смысла: они называли стоимость разработки, а мы — бюджет. Поэтому лучше вы подумайте и сообщите свои условия, а я вам скажу, устраивают они нас или нет.

Разработчик. Хорошо, без проблем.

А вот второй диалог.

Разработчик. Джон, я хотел бы обсудить с тобой стоимость проекта. Когда я проанализировал стоимость реализации аналогичных решений, то оказалось, что компании были готовы вложить в разработку от \$15000 до \$22000. Вы согласны с такой приблизительной стоимостью данного проекта?

Заказчик. К сожалению, это нереально. Боюсь, что мы можем предложить вам \$5000, максимум — \$7000.

Разработчик. Хорошо, Джон. К сожалению, на таких условиях выполнить работу мы не сможем. В любом случае спасибо за время, которые вы нам уделили.

В обеих ситуациях обсуждение завершилось неудачно. Однако во втором случае разработчик четко знает, что его не устраивает финансирование, тогда как в первом сценарии может быть потеряно время, прежде чем предложения разработчика будут отвергнуты.

Никогда не стесняйтесь спрашивать о финансировании. Будьте честным и открытым, но в то же время тщательно готовьтесь к разговору с заказчиком.

Коммерческие условия

При разговоре с заказчиком стоит затронуть и коммерческие условия выполнения работ, связанных с проектом. Если за его выполнение заказчик готов заплатить около \$50000, то это, безусловно, очень хорошее начало. Однако некоторые организационные вопросы, которые зачастую подробно не обсуждаются, позже могут привести к возникновению серьезных проблем.

Как и при обсуждении бюджета, коммерческие условия должны быть оговорены как можно раньше. Впоследствии это позволит избежать многих трудноразрешимых проблем. Следующие вопросы являются более важными для небольших предприятий, а не крупных корпораций, однако их стоит задать в любом случае.

- “Обычно мы просим заказчика заплатить аванс, чтобы в процессе построения системы мы могли вести свои финансовые дела. Вы не будете возражать против этого?”
- “Поскольку наша компания является небольшой, для нас очень важны условия оплаты. Все наши счета необходимо оплатить в течение 30 дней. Не возникнет ли с этим проблем в вашей бухгалтерии?”

Еще раз стоит повторить, что используемые фразы являются очень важными. В разговоре лучше использовать дружественные вопросы, например: “Вы не будете возражать против этого?”. Действительно, лучше получить ответ: “Нет, я так не думаю”, чем услышать что-нибудь более агрессивное или недоброжелательное.

Вопрос “Как вы думаете, это может привести к разрыву нашей совместной деятельности?” выглядит слишком эмоциональным. Однако следует учитывать и то, что наиболее подходящим ответом на этот вопрос является “нет”. Другими словами, подобный вопрос во многом предопределяет и ответ. Положительные ответы полезны не только потому, что они важны для разработчика, но и потому, что позволяют лучше разобраться с тем, что хочет заказчик.

Исследования показывают, что обсуждения, участники которых используют положительные слова и жесты, оставляют гораздо лучшее впечатление по сравнению с разговорами, которые сопровождаются отрицательными ответами и покачиванием головы.

Если обсуждение вызвало у заказчика положительные эмоции, то можно считать, что вы сформировали у него положительное впечатление и о себе.

Планы на будущее

Постарайтесь как можно быстрее выяснить у заказчика, какие его планы на использование системы в будущем. К этому моменту уже должны быть известны основные требования к системе. Однако не менее важно заглянуть и за их “пределы”, поскольку это во многом поможет определить функциональность системы в будущем.

Постарайтесь как можно раньше узнать о возможных сценариях использования системы в будущем. Именно они станут основой для выработки новых требований, даже если их реализацией будут заниматься другие разработчики. Проявляя интерес к вопросам, решение которых напрямую не финансируется (или их решение на данном этапе не запланировано), вы продемонстрируете заказчику стремление избежать подхода “быют — беги”. Кроме того, при таком подходе заказчик будет рассматривать вас как надежного партнера, с которым можно иметь дело.

“Внешний облик” приложения

В настоящее время даже самое простое Web-приложение должно иметь привлекательный вид. Его проектирование должно стать неотъемлемой частью общего процесса разработки программного продукта. На этом аспекте также следует акцентировать внимание заказчика.

При этом очень важно узнать, какие требования выдвигаются к внешнему виду приложения. Например, может оказаться, что его графический интерфейс должен удовлетворять корпоративному стилю или соответствовать дизайну уже существующего программного обеспечения. Кроме того, следует выяснить, может ли изменяться корпоративный стиль компании, и если да, то как часто.

Технология

Представитель заказчика, скорее всего, не разбирается в технических вопросах реализации программных систем. И для большинства проектов такая ситуация является нормальной, поскольку с ним обсуждаются сценарии использования системы, цели, требования, но никак не конкретные технологии реализации. Но даже учитывая такую ситуацию, на данном этапе было бы неплохо задать заказчику несколько вопросов о технологиях.

Самое главное — выяснить, насколько вы вольны в выборе технологий. В своем письменном предложении необходимо, по крайней мере, описать выбранную платформу и инфраструктуру, а также обосновать их выбор. Поэтому в процессе общения на данную тему необходимо выяснить, не возражает ли заказчик против использования языка программирования PHP. Следует также затронуть вопросы развертывания и безопасности.

Если в процессе общения возникнут трудности или разногласия, это может послужить для вас “красным флагом”. Если, например, клиент настаивает на использовании технологий ASP или SQL Server, то вы, как профессионал PHP, можете либо отказаться от выполнения данного проекта, либо попытаться переубедить заказчика, что PHP является наиболее подходящим решением.

В любом случае разговор подобного рода необходимо провести с заказчиком как можно раньше.

Поддержка

Если и есть вопрос, который обязательно необходимо обсудить с заказчиком, так это поддержка разработанной системы. Подобный вопрос у заказчика, возможно, даже не возникал. Однако он является очень важным.

Поэтому необходимо выяснить, кто будет отвечать за разработку пользовательской документации, кто будет обеспечивать поддержку системы после развертывания и какого рода сервисы ожидает получить клиент в процессе развертывания системы. Если же заказчик затрудняется ответить на такие вопросы, постараитесь дать ему соответствующие советы, выскажите свои предложения и получите одобрение с его стороны.

Что делать дальше

Итак, после формального общения с заказчиком необходимо переходить к “наступлению”. Если вы находитесь в благоприятной ситуации и намериваетесь разработать приложение для другого отдела компании, то можете пропустить следующий раздел.

Однако, если вам необходимо написать полноценное ценовое предложение и убедить клиента, что именно вы достойны разрабатывать систему, то вам следует внимательно изучить следующий раздел.

Написание предложения

Само по себе предложение может принимать разные формы. Это может быть презентация, модель, прототип или демонстрационная версия подобной системы, которую вы разрабатывали в прошлом. Однако каждая такая форма должна сопровождаться письменным предложением. И именно на это будет направлено наше внимание в рамках данного раздела.

Сразу же возникает вопрос: а нужно ли вообще полноценное предложение? Может, достаточно ограничиться формальным ценовым предложением (фактически счет-фактурой)?

Предложение или счет

Само собой разумеется, что заказчику необходимо предоставить счет за работу, которая будет вами выполнена. Однако написание полноценного предложения это больше, чем просто написание цифр.

Вспомните, когда вы в последний раз что-то покупали. Это, возможно, был автомобиль, телевизор или кухня. Вы получаете “ощущение покупки”, когда приобретаете телевизор за \$300. Однако почему такое ощущение пропадает, когда вы покупаете на \$300 всякого рода продукты для всей семьи?

Разница очевидна. Телевизор является долгосрочной инвестицией, в то время как продукты исчезнут в течение несколько недель.

В таком же ключе подумайте и о работе, за которую вы планируете получить деньги от заказчика. Не переусердствуйте с наличными. Лучше подумайте, что эта работа значит для заказчика. Если она будет приносить пользу на протяжении достаточно большого промежутка времени, то ее необходимо соответствующим образом продавать и готовить полноценное предложение. Если же ваша работа является краткосрочной (“расходный материал”), в таком случае достаточно предоставить счет.

Примерами работ, для которых достаточно предоставить счет, могут служить внесение небольших изменений в существующую систему или установка модулей PHP на серверы. В обоих случаях многое не скажешь. Так и не говорите!

При представлении счета очень полезно включить в него информацию о вашей компании — видах деятельности, опыте, клиентах и т.д. Это поможет представителям заказчика больше узнать о вас, как о разработчике программного обеспечения.

Далее подразумевается, что вы нацелены на создание полноценного предложения.

Предложение или спецификация

Если вы нацелены на создание серьезного предложения, не перестарайтесь, чтобы ваше предложение не превратилось в спецификацию.

Со спецификацией мы познакомимся ниже в этой главе. В общем случае спецификация представляет собой детальное описание разрабатываемой системы. На данном этапе не следует вдаваться в детали. И не только потому, что вы сделаете лишнюю работу, но и потому, что если вам не удастся получить заказ, вы фактически дадите бесплатные советы.

Поэтому тщательно подумайте, что следует включить в ваше предложение. При этом необходимо убедить заказчика в следующем:

- вы полностью понимаете, чего хочет заказчик;
- вы разработали эффективное решение, удовлетворяющее требования клиента;

- вы сможете разработать приложение в течение необходимого времени;
- за вашу работу заказчику придется заплатить \$X, и эта сумма методически обоснована;
- вы являетесь именно тем разработчиком, с которым стоит иметь дело.

Очевидно, что обсуждение технических вопросов, связанных с функциональностью или технологиями реализации системы, не приведет к выполнению приведенных выше пунктов.

Иногда заказчику приходится объяснять, почему в предложении отсутствует описание всей функциональности системы. Будьте открыты. Постарайтесь объяснить клиенту, что вы предоставите ему спецификацию системы, которая будет полностью удовлетворять выдвигаемым требованиям. На данном этапе вы предлагаете качественное и эффективное решение. При этом вы полностью понимаете требования клиента, а не специфику реализации самой системы. Если заказчик принимает ваши доводы, он фактически понимает, что вы будете способны реализовать необходимое ему приложение.

Если от вас требуют написать более подробное предложение, будьте внимательны. В этом случае вы рискуете бесплатно отдать свои ценные предложения (советы).

Кто должен участвовать в написании предложения

К этому времени, возможно, вы уже знаете, кто будет участвовать в реализации проекта. Об этом мы поговорим в конце данной главы. Однако сейчас следует задуматься, кто будет участвовать в написании предложения. Поскольку ответственность лежит на вас, последнее слово в этом процессе целиком будет за вами.

Помните, что к созданию предложения необходимо подходить творчески! Вы должны показать, что обладаете преимуществами, по сравнению с конкурентами, и способны обеспечить необходимое решение в рамках предоставленного бюджета. Поэтому в процессе создания предложения целесообразно вовлечь людей, способных внести элемент творчества.

Поскольку в предложение не будут включены технические аспекты реализации приложения, достаточно иметь одного старшего специалиста и дизайнера (безусловно, если у вас есть такая возможность). Как вы втроем будете взаимодействовать в рамках данного процесса, зависит от вас. Однако подход к созданию документа с использованием “мозгового штурма” показал большую эффективность, нежели распределение заданий для участников с последующим объединением выполненных работ.

Когда следует двигаться дальше

Некоторые заказчики заставят вас изрядно потрудиться для получения заказа. Во многих случаях дело стоит того, хотя риск может быть достаточно велик. Поэтому знание того, когда не нужно останавливаться и следует идти дальше, требует существенных навыков. Речь идет не о лени. Вы могли бы потратить свое время куда эффективней, ведя переговоры по более перспективным проектам, нежели тратить усилия на безнадежный проект.

Дополнительные шаги, которые может потребовать заказчик, могут включать демонстрацию презентаций, демонстрацию работы подобных систем, разработанных вами в прошлом, и даже отзывы и рекомендации ваших клиентов. Поэтому на данном этапе необходимо задать себе несколько вопросов.

- Насколько высоки ваши шансы на получение проекта?
- Какие усилия необходимо потратить на дополнительные шаги?
- Чем занимаются ваши конкуренты и другие разработчики?

Если есть основания предполагать, что процесс переговоров вот-вот закончится, тогда, безусловно, его необходимо продолжить. Если заинтересованность клиента не прослеживается и, возможно, он уже принял решение в пользу конкурента, то переговоры следует приостановить.

Всегда старайтесь оценивать затраты на выполнение дополнительной работы, как денежные (на командировки, расходные материалы), так и временные. Как правило, если расходы достигают 5% от стоимости проекта, определенно следует сделать шаг назад.

Если это возможно, постарайтесь проанализировать деятельность конкурентов. Наличие союзника в лагере заказчика может принести существенные дивиденды. Конечно, секретарь в приемной не принимает решений, но может держать ухо востро. Ставьте использовать любую информацию. Если ваши конкуренты все еще продолжают вести переговоры, так же поступайте и вы.

Когда следует сказать “нет”

Профессионалы знают, когда наступило время забрать предложение в ходе переговоров и отказаться от проекта. При этом важно знать, когда можно встать и уйти со своим предложением.

Кроме таких очевидных моментов, как нехватка необходимых навыков или персонала, или же недостаточный бюджет заказчика, необходимо внимательно относиться к следующим “симптомам”.

- Просто интересный разговор. Например, клиент не имеет достаточно полномочий для того, чтобы предложить вам реализацию проекта. Однако он прочитал о вас в каком-то журнале и решил поинтересоваться, сколько будет стоить интересующая его разработка. “Приметами” подобного разговора служат низкая должность, клиента, а также частые отлучки в связи с необходимостью поговорить с начальником.
- Разговоры об уменьшении бюджета проекта. Клиент уже нашел разработчика, но при этом пытается снизить стоимость проекта путем получения “более выгодных” предложений от других разработчиков.
- Проект-мечта. Обычно подобного рода разговоры ведет заказчик с достаточно высокой должностью. Однако при этом его идеи не разделяют другие сотрудники компании. Безусловно, деньги на такой проект могут найтись. Но в один прекрасный момент этот заказчик исчезнет. И так, по всей видимости, случится с вашим проектом.

Всегда помните о выгоде. Если проект принесет вам больше проблем, не бойтесь отказаться от него. Заказчик будет уважать ваше решение.

Структурирование предложения

Структура предложения может иметь произвольный вид. Однако если вы готовите предложения для одного и того же клиента, важно быть последовательным.

Выделим следующие важные аспекты предложения.

- Правильно оформляйте титульную страницу.** Указывайте дату создания предложения. Название предложения должно соответствовать имени заказчика (а не вашему имени). При этом предложение должно содержать строку “подготовлено для XXX”, где XXX — имя человека, с которым вы непосредственно контактируете.
- В первый раздел включите результаты интервью.** Если заказчик предоставил требования в письменном виде, в первый раздел следует включить краткое резюме, размером не более половины страницы. Многие заказчики принимают решения на основе первой страницы предложения и стоимости проекта. Поэтому используйте пространство разумно.
- Представьте свое решение.** В описание целесообразно включить диаграммы Visio и иллюстративный материал. Все описание должно содержать не более 2-3 страниц. Избегайте излишней детализации. Это общее описание системы — творческая интерпретация вашего понимания требований клиента.
- Обеспечивайте прозрачную финансовую политику.** Не старайтесь скрыть расчеты от заказчика. Продемонстрируйте ему, как получена названная сумма, включите некоторые детали коммерческого обоснования.
- Если вы раньше работали с этим заказчиком, не повышайте стоимость. Например, если ранее вы оценивали свою работу в \$800 в день, не завышайте стоимость следующего заказа либо детально обоснуйте повышение.
- Говорите о деталях.** Расскажите заказчику о своей компании, истории ее развития, опыте, используемом процессе разработки. По возможности назовите предыдущих заказчиков.
- Будьте точны с указанием дат.** Объясните заказчику, сколько времени потребуется для выполнения всех этапов работы. Назовите дату завершения разработки и старайтесь уложиться в срок.
- Назовите основные задачи.** Советуйтесь с заказчиком о последовательности реализации сценариев.

При личных встречах или телефонных разговорах ссылайтесь на предыдущую информацию. Напомните заказчику его слова, сказанные на предыдущей встрече, и укажите, как вы воспользовались ими при принятии решений. Например, “на прошлой встрече вы упомянули об обеспечении доступа к системе через Web-службы. Мы подготовили соответствующие предложения”.

Если вы отправляете документы по электронной почте, используйте PDF-формат, а не формат MS Word. Всегда проверяйте по телефону, получил ли заказчик отправленные документы, однако не торопите его с принятием решения.

Самое главное, не нарушайте договоренностей. Если вы пообещали подготовить документ к некоторой дате, старайтесь успеть в срок. Первое впечатление является самым главным. Вам могут не дать возможности исправить допущенную в первый раз ошибку.

Не старайтесь побеждать всех во что бы то ни стало.

Если с первого раза не достигнуто согласие с заказчиком, выслушайте его замечания и постараитесь улучшить свое видение. При таком подходе у вас скоро появится командный дух при работе над данным проектом.

Выбор персонала

Прежде чем приступать к реализации проекта, необходимо собрать достойную команду. Ее численность зависит от требований проекта и выделенных ресурсов. Рассмотрим различные роли участников проекта.

Менеджер проекта

Если вы читаете эту главу, то, вероятно, претендуете на эту роль. Менеджер проекта отвечает за его выполнение, от формулировки требований до развертывания системы. Он напрямую взаимодействует с главными разработчиками, при необходимости выполняет функции исполнительного директора и обеспечивает взаимодействие с заказчиком.

Менеджер проекта устанавливает внутренние вехи реализации проекта и соотносит их с внешними целями. Он отслеживает ход проекта, выявляет любые риски или сложности и направляет ход реализации проекта.

Это не техническая роль, но грамотный менеджер проекта должен понимать основные аспекты работы архитекторов, разработчиков и других участников проекта. Идеальный менеджер проекта — это креативный, технически грамотный специалист с высоким уровнем организации труда. Он должен обеспечить своевременную реализацию проекта в рамках выделенного бюджета. В каждом проекте должен быть лишь один менеджер проекта.

Исполнительный директор

Исполнительный директор является контактным лицом при общении с заказчиком.

Все конкретные пожелания заказчик передает исполнительному директору, который в свою очередь сообщает о них менеджеру проекта. Менеджер проекта обеспечивает реализацию требований заказчика внутри своей организации.

Исполнительный директор не отслеживает ход выполнения проекта, он лишь отвечает за взаимодействие с заказчиком. При этом он должен регулярно общаться с менеджером проекта и быть в курсе дела.

Эта роль в проекте не обязательна, однако наличие подобного специалиста повышает эффективность взаимодействия с заказчиком. Такой специалист не обременен отслеживанием состояния проекта.

Идеальный исполнительный директор — это общительный человек, который может легко войти в контакт с любым партнером.

В каждом проекте должен участвовать только один исполнительный директор.

Главный архитектор

Главный архитектор — это лицо, принимающее технические решения в течение всего жизненного цикла проекта. Именно он продумывает реализацию всех требований заказчика в рамках конкретной технологии.

Главный архитектор должен иметь опыт разработки программных систем, а также некоторый опыт управления проектом. Он также должен обладать исключительными способностями по управлению командой разработчиков, поскольку именно перед ним отчитываются его “младшие” коллеги.

Главный архитектор должен понимать принципы работы программных систем и сетей, чтобы грамотно реализовать инфраструктуру развертывания системы.

В обязанности главного архитектора входит разработка или утверждение схемы базы данных и объектной модели, поддержка стандартов написания кода и регулярное его оценивание. Учитывая объем работы главного архитектора, становится ясно, что такой специалист должен работать с одним проектом.

Разработчики и кодировщики

Над каждым серьезным приложением работает группа разработчиков и кодировщиков.

Эти две роли имеют одно существенное различие. Разработчик обладает определенной степенью свободы, в рамках которой он реализует идеи главного архитектора, принимает решения по элементам архитектуры, структуры и инфраструктуры. Кодировщик должен придерживаться рекомендаций и стандартов, выдвинутых разработчиками.

Если бюджет проекта не позволяет использовать отдельных разработчиков для клиентской части приложения (см. следующий раздел), то задача разработки шаблонов и HTML-страниц тоже ложится на плечи разработчиков.

Количество разработчиков и кодировщиков для каждого проекта ограничивается лишь бюджетом и масштабом проекта.

Разработчики клиентской части приложения

Обязанность этих специалистов состоит в создании HTML-страниц, листов стилей CSS и кода JavaScript, представляющих интерфейс пользователя данного приложения. Обычно их работа связана с созданием шаблонов, например с использованием Smarty. К подобным специалистам не выдвигается требований знания языка PHP или других серверных языков сценариев.

Разработчики клиентской части взаимодействуют с главным дизайнером или художниками проекта, но отчитываются непосредственно перед главным архитектором. Количество разработчиков клиентской части определяется лишь бюджетом и масштабом проекта.

Главный дизайнер

Главный дизайнер руководит процессом выработки дизайна и определяет общую визуальную концепцию приложения. Он советуется с заказчиком и принимает решения по выбору цветовой гаммы, шрифтов и внешнего вида приложения. В процессе принятия решений он консультируется с главным архитектором, определяющим технические ограничения для данного проекта. Обычно в проекте участвует не более двух дизайнеров.

Художники

Художники находятся в подчинении у дизайнера. Обычно они являются специалистами по Photoshop, Illustrator или другим графическим пакетам. Они отвечают за разработку конкретных визуальных компонентов системы, например панели навигации или меню.

Художники обычно работают с существующими шаблонами, стилями и цветами, определенными главным дизайнером проекта. Количество художников зависит от масштаба проекта.

Совмещение ролей

Даже в больших компаниях возникает проблема с кадровым составом. Поэтому зачастую один и тот же специалист в рамках проекта выполняет несколько ролей.

- Менеджер проекта может эффективно выступать в роли исполнительного директора.
- Главный архитектор может выполнять любые задачи разработчиков или кодировщиков.
- Разработчики могут выполнять обязанности разработчиков клиентской части.
- Дизайнеры могут выполнять работу художников.

Организация работ

По возможности участники одного проекта должны работать в одном помещении. При этом обеспечивается синергетический эффект работы коллектива, т.е. общий результат выполнения проекта гораздо больше, чем сумма результатов его отдельных участников. Такая организация работы существенно упрощает задачи менеджмента проекта.

Роль заказчика

Очень важно грамотно структурировать команду разработчиков и привлечь к выполнению проекта представителей заказчика. Очевидно, что выбор представителей заказчика находится за пределами вашей компетенции.

Однако вы должны убедить заказчика, что выбор представителя по взаимодействию с командой разработчиков во многом определяет успех всего проекта. Он должен регулярно отслеживать ход выполнения проекта, отвечать на все необходимые вопросы и высказывать свои пожелания.

Резюме

В этой главе вы познакомились с основами успешного руководства проектом по разработке программного обеспечения, а также узнали, как важно задать ключевые вопросы и на основе ответов сформулировать привлекательное предложение. В следующей главе речь пойдет о различных процессах и методологиях разработки программного обеспечения и выборе оптимального процесса в каждом конкретном случае.

20

Проектирование системы

Разработка сложного динамического бизнес-приложения на PHP с командой профессионалов — непростая задача. На вашем пути встретятся трудности не только технического характера.

Каркас, выбранный для бизнес-планирования, во многом определяет направление развития проекта по разработке программного обеспечения. Выбор процесса определяет интенсивность изменения проекта и обеспечивает “дорожную карту” его реализации. Использование более динамического подхода к планированию позволяет уменьшить риск, связанный с техническими или производственными факторами.

Выбор подхода к планированию — достаточно ответственная задача. Перед принятием решения нужно ознакомиться со всеми возможностями планирования, независимо от ваших личных или профессиональных предпочтений.

Выбор процесса

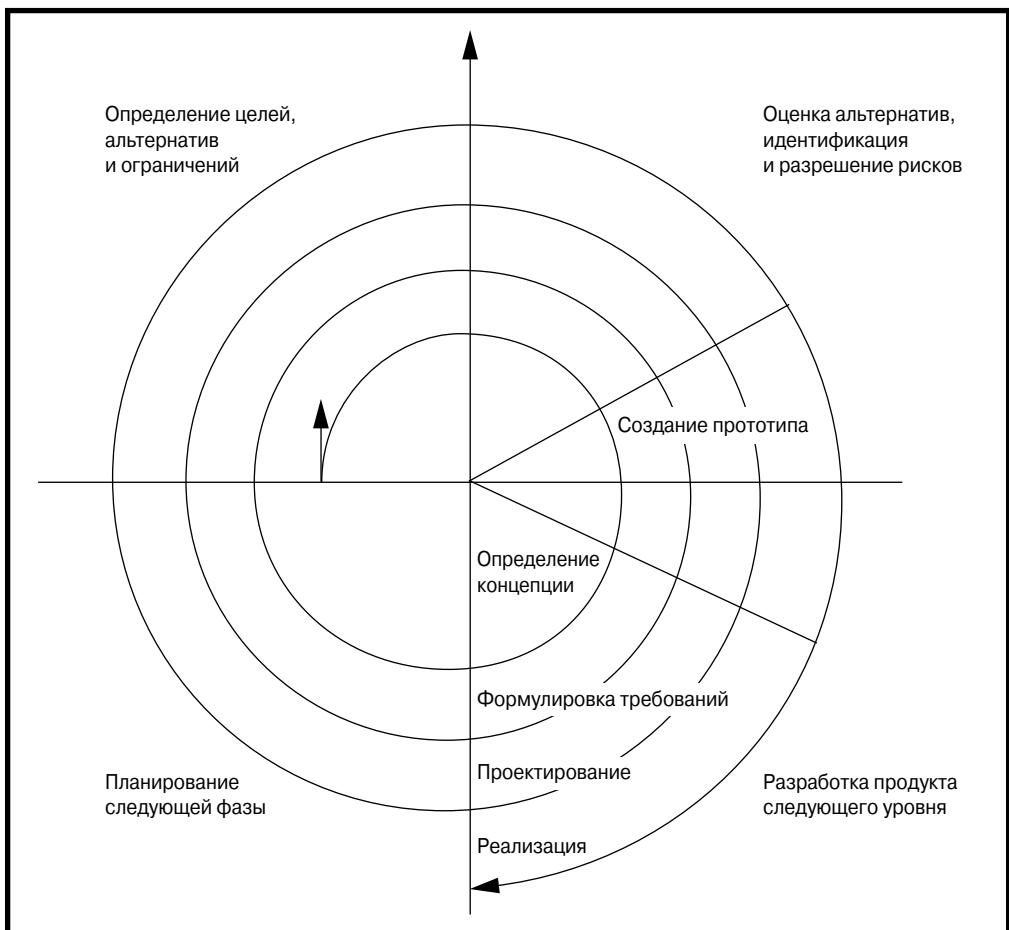
Собрав команду разработчиков, вы горите желанием приступить к выполнению задач, поставленных заказчиками. Однако сначала необходимо определить, какой процесс выбрать для создания продукта.

Существует несколько вариантов выбора, но среди них следует выделить два фундаментальных процесса, известных под названиями *каскадного* и *спирального*. В целом, эти процессы включают одинаковые основные этапы и элементы. Различие состоит лишь в порядке и способе их применения.

Каскадный процесс

В каскадном процессе весь проект рассматривается как единое решение, которое нужно реализовать. Пример такого процесса представлен на рис. 20.1.

Ключевой принцип каскадного процесса состоит в том, что ни одна фаза не может начаться, пока предыдущая полностью не завершена. Другими словами, прежде чем приступить к фазе проектирования, необходимо полностью завершить разработку и утвердить полную спецификацию проекта. Только после этого можно приступать к реализации и тестированию проекта.

*Рис. 20.1.*

Этот традиционный процесс в принципе хороший, но при его практическом применении возникают некоторые проблемы. Например, если в течение конкретной фазы допущена ошибка, она будет обнаружена только позднее. Поэтому данную и все остальные фазы придется повторить, а это требует много времени и денег.

Другая потенциальная проблема связана с тем, что проектное решение строится за один проход, поэтому в процессе разработки сложно изменить некоторые его фрагменты. Если заказчик захочет просмотреть проект, ему это вряд ли удастся до полного завершения разработки.

Спиральный процесс

Спиральный процесс включает те же основные компоненты, что и каскадный: составление спецификации, проектирование и разработку архитектуры, реализацию и тестирование. Однако в нем проект не рассматривается как целостное решение. Система делится на отдельные компоненты, к каждому из которых и применяется данный процесс.

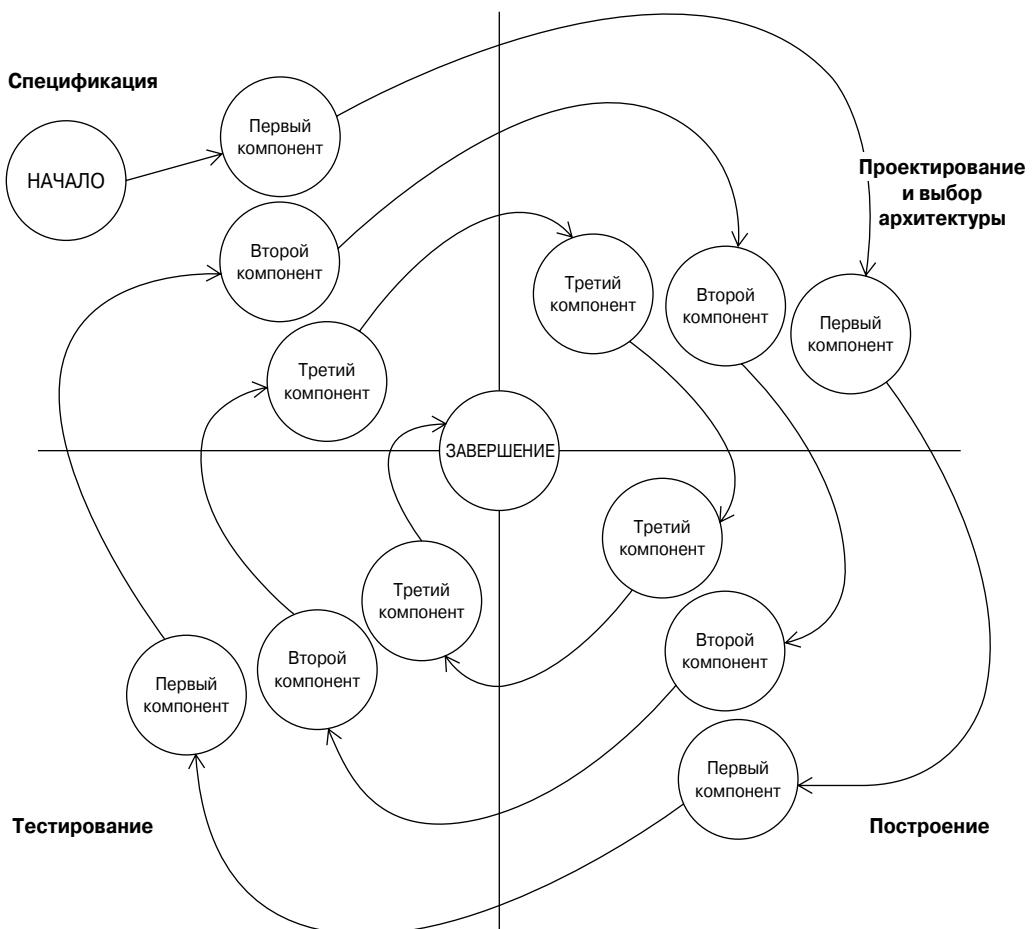


Рис. 20.2.

Компонент может представлять собой некоторую часть разрабатываемой системы или уровень с частью требуемых функций. Разработка каждого компонента проходит четыре стадии: разработку спецификации, проектирование и создание архитектуры, реализацию и тестирование.

Процесс называется спиральным, потому что этот путь (четыре стадии) многократно проходится для каждого компонента, как бы по спирали (рис. 20.2).

Большое преимущество этого процесса состоит в том, что ошибка, допущенная на одном из этапов, будет определена не в самом конце проекта, а на той же стадии развития процесса (на том же витке спирали). Затраты времени и ресурсов на ее устранение будут значительно меньше, поскольку придется иметь дело только с одним дискретным компонентом, а не с целым приложением. Кроме того, заказчик сможет воспользоваться первыми элементами системы значительно раньше, что составляет огромное преимущество.

Стоит отметить, что фаза тестирования несколько отклоняется от спирального процесса, поскольку при тестировании каждого нового компонента необходимо проводить и все предыдущие тесты разработанных ранее компонентов.

Принятие решений

В этой главе мы не будем отдавать предпочтение какому-либо процессу. В следующей главе мы остановим свой выбор на одном из процессов в контексте разработки приложения автоматизации торговли, но сейчас вопрос выбора процесса разработки мы оставляем открытым. Выбор метода в значительной степени зависит от проекта.

Хотя спиральный процесс имеет множество преимуществ, платой за них является дополнительный объем бумажной и административной работы. Поэтому при выборе процесса разработки нужно тщательно взвесить все “за” и “против”.

Только опыт позволит вам оценить и сравнить оба процесса. Чаще всего решения принимаются на основе личных предпочтений. Независимо от вашего выбора, оба процесса включают общие этапы и рекомендуемые приемы, о которых и пойдет речь в последующих разделах.

Общие рекомендации

Как было упомянуто выше, оба подхода, каскадный и спиральный, подразумевают одинаковую последовательность этапов разработки и включают фазу создания спецификации, этап проектирования, фазу реализации и стадию тестирования.

Разница состоит в том, какую долю занимает каждый из этих этапов на каждой итерации.

Термины *спецификация*, *проектирование* и *реализация* можно трактовать по-разному. Рассмотрим, как можно реализовать каждую фазу независимо от того, какой процесс был выбран для разработки проекта.

Допустим, был выбран каскадный процесс, значит, речь пойдет о фазах разработки целого проекта, а не отдельных компонентов. Если выбран спиральный подход, эту же методику можно применять к каждому витку спирали (компоненту).

Фаза разработки спецификации

Фаза разработки спецификации — во многих отношениях самая сложная часть проекта. Зачастую она значительно сложнее, чем построение самого процесса. Такая ситуация возникает при создании системы, предоставляющей доступ к базе данных. Задача разработки такой системы не вызывает трудностей, намного сложнее удовлетворить требования заказчика.

На этом этапе необходимо определить, что конкретно требуется реализовать. При этом задачу должен понимать не только сам разработчик. Гораздо важнее обеспечить взаимопонимание с заказчиком. Самый простой способ разобраться — разработать спецификацию проекта на бумаге. Обычно спецификацию проекта разделяют на функциональную и техническую.

Функциональная спецификация

Функциональная спецификация проекта в деталях описывает ожидаемое функционирование системы. Она должна быть написана на понятном языке и обеспечивать рациональное обоснование технических решений. Как правило, в процессе разработки приходится придерживаться этого документа и воспроизводить его в деталях. На этом этапе уже можно требовать одобрения проекта клиентом и утверждать сроки сдачи проекта.

В разрабатываемый документ необходимо включить все возможные детали. Удалите все неточности, чтобы впоследствии избежать лишних ошибок. Важно при-

держиваться оптимального уровня детализации. Если он завышен — документ становится нечитабельным, если занижен — появляется множество интерпретаций и нарушается функциональная однозначность.

Поэтому постарайтесь не допускать двусмысленных интерпретаций документа. Например, если используется фраза “пользователь может загрузить фотографии”, необходимо указать количество и формат снимков. В противном случае пользователь может захотеть загрузить несколько сотен фотографий, и у приложения могут возникнуть проблемы с быстродействием. Попробуйте использовать фразу “пользователь может загрузить не более 8 фотографий в JPEG-формате”.

В то же время следует избегать излишней детализации, поскольку ненужные подробности могут вызвать раздражение даже у самого терпеливого клиента. Поэтому старайтесь исключать из спецификации фразы типа: “На данной странице содержится кнопка, при щелчке на которой пользователь переходит на начальную страницу”.

При составлении функциональной спецификации старайтесь формулировать мысли как можно яснее. Можно следовать такой схеме.

- Кратко, своими словами, опишите пожелания клиента (не более чем на половину страницы).
- Поясните цель создания данного документа и перечислите другие документы, которые будут включены в спецификацию. Ставьтесь излагать идеи как можно яснее, поскольку данный документ подписывается заказчиком.
- Кратко изложите суть предложенного решения (не более чем на полстраницы).
- Для каждого компонента обеспечьте подробное описание функциональности, ориентируясь на неопытного пользователя.
- При необходимости для внесения ясности добавьте диаграммы. Как бы вы ни старались подчеркнуть, что это всего лишь картинки, заказчик будет рассматривать их как готовое проектное решение. Поэтому не злоупотребляйте диаграммами.
- Дополните документ объяснениями, необходимыми клиенту, выслушайте вопросы и внесите изменения. Объясните, какие этапы разработки последуют за подписанием документа, и укажите вехи, по которым можно отслеживать ход проекта.

На практике спецификация составляется и согласовывается за две-три итерации. Если заказчик подписал спецификацию с первого раза, с 90%-ной уверенностью можно утверждать, что он ее не читал.

Техническая спецификация

В функциональной спецификации описывается, что вы собираетесь делать, а в технической спецификации — как вы будете это реализовывать.

Уровень детализации этого документа в значительной мере зависит от заказчика. Обычно, если на предприятии у заказчика имеется отдел информационных технологий, который принимает участие в проекте, вам придется обеспечивать максимальный уровень детализации, вплоть до программного кода. В противном случае можно обойтись следующей информацией.

- Предлагаемый язык разработки с объяснением, почему он лучше других.
- Предлагаемая платформа баз данных со сравнительным анализом различных платформ.

- Предлагаемая операционная система для Web-сервера и серверов баз данных со сравнительным анализом с другими платформами.
- Обоснование плана развертывания проекта, детальное описание требований к пропускной способности каналов связи.
- Содержание данного документа и перечень дополнительных документов, включаемых в спецификацию. Помните, что техническая спецификация неразрывно связана с функциональной спецификацией, подписанной заказчиком.

Если для разработки проекта был выбран спиральный процесс, нет необходимости обеспечивать техническую спецификацию для каждого витка спирали. Разработайте одну техническую спецификацию в начале проекта и объясните заказчику, что она относится ко всем отдельным компонентам.

Для многих заказчиков, особенно тех, у которых нет своего технического отдела, этот документ может оказаться непонятным. Есть два выхода: приложить максимум усилий, чтобы подробно все объяснить и убедиться, что клиент остался доволен принятым решением, или позволить ему подписать документ вслепую. Иногда второе решение самое легкое. В конце концов клиент платит за работу, а не за обсуждение технических решений.

Фаза проектирования

Термин “проектирование” может ввести читателя в заблуждение. Он имеет отношение к разработке графического интерфейса, и к разработке программной системы. Но это разные дисциплины и нужно понимать эту разницу. Вероятнее всего, клиент как можно скорее захочет увидеть некоторые графические компоненты проекта.

Главный разработчик проекта может взять на себя ответственность и предложить проекты графических элементов. При этом он должен проконсультироваться с главным архитектором системы и обеспечить реализуемость таких решений. Опытные разработчики знают о технических ограничениях и поэтому требуют меньше рекомендаций при разработке Web-приложений.

Будьте готовы к тому, что вам потребуется создать несколько вариантов проектных решений, прежде чем вы сможете удовлетворить заказчика. У заказчика есть свое видение проекта, и он будет стараться донести его до вас. Заказчик знает, что результат не всегда совпадает с его ожиданиями, поэтому не постесняется детально разъяснить свои требования.

При согласовании проектного решения с заказчиком сначала можно обсудить графические элементы, показав их на переносном компьютере с установленным программным обеспечением для быстрой разработки. Тогда изменения можно внести непосредственно в процессе обсуждения с заказчиком. Реакция заказчика: “Это именно то, что мне нужно” — может оказаться столь же важной, как и коммерческая выгода.

Креативная спецификация

После определения проектных решений полезно разработать креативную спецификацию, описывающую стиль оформления программной системы. Этот документ обычно не подписывается заказчиком.

Цель этого документа — передать художественное видение главного разработчика. Вся выполненная работа должна соответствовать этому видению.

В документ включают детальное описание цветов, шрифтов, формата, логотипов, фона — всех компонентов, определяющих внешний вид и художественное оформление системы. Если клиент попросит добавить кнопку с конкретной функциональностью, она должна отвечать общему стилю оформления, предложенному главным разработчиком.

Карта страниц

Наиболее спорным компонентом фазы проектирования является построение карты страниц, представляющей собой черно-белую графическую модель страниц в Web-браузере пользователя. Обычно такая карта — это попытка сформировать интерфейс пользователя, не передавая какие-либо аспекты проектного решения. Это позволяет обсудить интерфейс пользователя отдельно от общего проектного решения.

Группе разработчиков можно поручить создать такую карту наряду с креативной спецификацией.

На практике создание карты страниц сопряжено с двумя возможными проблемами. Первая состоит в том, что в понимании клиента бывает трудно разделить понятия пользовательского интерфейса и проектного решения. Демонстрация карты страниц заказчику может оказаться утомительным и напряженным занятием, поскольку он воспринимает эти изображения как окончательное проектное решение, не подлежащее изменению. Ему надо четко объяснить, что собой представляет карта страниц и для чего она создается.

Вторая проблема — действительно построить карту страниц. Web-дизайнеры обычно не являются специалистами по интерфейсам и не любят использовать Visio и другие пакеты, предназначенные для выполнения этих задач. Менеджеры проекта обычно очень заняты, в результате чего задача построения карты страниц ложится на плечи главного архитектора. Поэтому общее видение системы во многом определяется позицией главного архитектора.

Архитектура программного обеспечения: план из 10 пунктов

Архитектурный план программной системы — это еще один документ, который можно представить заказчику на утверждение. Обычно его называют *планом из 10 пунктов* (*ten-point plan*), потому что он представляет собой последовательность действий, описывающих наиболее эффективный подход к созданию данного проекта или компонента.

Этот документ тоже во многом зависит от главного архитектора, поскольку именно этот специалист формирует его после утверждения технической спецификации. Архитектурный план не касается выбора технологии или аппаратных средств, а концентрируется на вопросах выбора схемы баз данных, построения объектных моделей или реализации шаблонов проектирования, таких как MVC. Главный архитектор передает этот документ своим подчиненным, которые, в свою очередь, выполняют более детальную его проработку.

Фаза реализации

Процесс реализации системы может начаться только после того, как будут согласованы все аспекты проектного решения. Для этой фазы нет четко определенной письменной документации, но можно иметь ввиду несколько пунктов.

- ❑ Поддерживайте серверы разработки (development server), промежуточный сервер, а также основной сервер. Это не обязательно должны быть физически разные машины, но вы всегда должны быть готовы продемонстрировать заказчику результаты хотя бы частично завершенных этапов разработки.
- ❑ Попытайтесь адаптировать какую-либо методологию программирования к своему проекту. Этот вопрос более подробно рассматривается ниже в этой главе.

- ❑ Главный архитектор должен управлять процессом реализации и контролировать группу разработчиков, которая должна предоставлять результаты вовремя и в необходимом формате. Это необходимо для того, чтобы главный архитектор мог предоставить руководителю проекта систему в целостном виде.
- ❑ Всегда используйте контроль версий. В последующих главах этой книги будет рассмотрено средство контроля версий, обеспечивающее бесплатное решение этого проблемного вопроса.

Фаза тестирования

В процессе разработки проекта необходимо проводить тестирование системы, позволяющее гарантировать качество полученного продукта.

Процесс тестирования всегда должен сопровождаться письменными заключениями, подробно описывающими результаты проверки. По очевидным причинам эти отчеты не следует предавать огласке, пока вы не будете на 100% уверены в успешном завершении испытаний.

Функциональное тестирование

Функциональное тестирование обеспечивает корректную работу программного продукта и точное выполнение спецификации. Будьте систематичны, протестируйте каждый аспект функциональности на различных входных данных или параметрах. Запишите результаты тестирования для каждого компонента.

Группе разработчиков нужно постоянно напоминать о необходимости регулярного тестирования разрабатываемых элементов. При этом можно использовать специализированные программы поиска ошибок, например Mantis (www.mantisbt.org).

Тестирование нагрузки

Если разрабатываемое приложение предназначено для обработки интенсивного трафика, имеет смысл оценить его предельные возможности заранее.

Тестирование нагрузки может выполняться только в реальной среде, обычно на основном сервере. Нет смысла тестировать приложение на сервере Xeon в операционной системе Linux, если известно, что в рабочем режиме оно будет функционировать на сервере Sun в операционной системе Solaris. Среди множества доступных пакетов самым популярным является LoadRunner от компании Mercury Interactive. Подобных результатов можно достичь с помощью специально разработанных сценариев тестирования.

При тестировании нагрузки нужно не бояться повышать планку. Заказчик не просто хочет знать, что его требования будут выполнены. Он будет рад услышать конкретные цифры, касающиеся производительности разрабатываемого приложения.

Передача проекта

Этап передачи разработанной системы заказчику является финалом всей работы.

На этом этапе следует убедиться, что клиент доволен конечным результатом. Возможно, понадобится провести краткий курс обучения работе с системой. Помните, что основной целью разработки является передача системы заказчику. Для подтверждения чистоты помыслов вручите заказчику компакт-диск с копией исходного кода.

В процессе передачи обычно возникает необходимость в изменении незначительных деталей. Позже мы рассмотрим, как их можно исправить или доработать.

Методология программирования

Давайте кратко рассмотрим процесс реализации. Известны несколько лучших практических методов реализации, которыми можно воспользоваться независимо от выбора процесса разработки.

Разработка на основе тестирования

В предыдущей части этой книги состоялось знакомство читателя с каркасами модульного тестирования и средством PHPUnit. Эти знания можно применить только при разработке приложения на основе тестирования.

Разработка на основе тестирования предполагает, что тест должен быть написан до создания самого кода программы. В процессе написания теста для разрабатываемого класса вы должны продумать способ его использования, его функции и интерфейс. Пакет PHPUnit (доступный через модуль PEAR по адресу <http://pear.php.net>) — хороший каркас для тестирования.

До начала разработки самого класса необходимо проверить работоспособность теста. Для проверки его работоспособности вам могут потребоваться “игрушечные” классы с таким же интерфейсом, как у настоящего.

Модульный тест для конкретного класса призван проанализировать его интерфейс и данные. Задача разработчика — писать классы так, чтобы они удовлетворяли модульному тесту.

Старайтесь обеспечить максимальную общность модульных тестов. Если несколько классов имеют похожие интерфейсы, их модульные тесты целесообразно совместить в рамках одного экземпляра PHPUnit.

Модульное тестирование имеет множество преимуществ, которые подробно описываются в главе 22. В любом случае вы сами решаете, какой подход к разработке следует использовать. Методология разработки на основе тестирования лучше подходит для больших проектов с множеством разнообразных компонентов.

Экстремальное программирование

Концепция экстремального программирования — это типичный пример практической парадигмы программирования. Это действительно парадигма, требующая проработки конкретных деталей в каждом отдельном случае. Тем не менее, как рабочая модель она включает множество аспектов, которыми можно воспользоваться в случае необходимости.

Что такое экстремальное программирование

Экстремальное программирование описывается путем перечисления различных аспектов и принципов. Их слишком много, чтобы рассматривать все, но на некоторые мы обратим внимание.

Основная цель экстремального программирования — обеспечить качественную разработку программного обеспечения, удовлетворяющего требования клиента. Достичь этой цели можно за счет управления сложностью. С ростом и усложнением системы возрастает и стоимость ее модификации или добавления новых свойств. Но верно и обратное: если система не слишком сложна, то стоимость добавления нового элемента не зависит от стадии разработки.

Это очень эффективный метод, потому что традиционная методология разработки описывается кривой, согласно которой модификация программного обеспечения постоянно дорожает и требует все больше времени для реализации.

Гораздо эффективнее и дешевле добавлять детали на стадии определения требований, а не на этапе реализации.

Предположим, что стоимость модификации системы возрастает достаточно медленно.

Если бы мы могли в любой момент добавлять, удалять и модифицировать свойства системы, то было бы целесообразно заранее определить набор необходимых свойств, устанавливаемых на начальных стадиях проекта.

Экстремальное программирование включает набор следующих принципов, сформулированных Кентом Беком (Kent Beck).

- ❑ Планирование. Определите масштаб проекта с учетом приоритетов заказчика и технических оценок. Постоянно его обновляйте.
- ❑ Регулярные выпуски новых версий. Как можно скорее, передайте части системы заказчикам и регулярно выпускайте новые версии.
- ❑ Простое проектное решение. Страйтесь обеспечить максимальную простоту проектного решения.
- ❑ Тестирование. Разработчики должны постоянно создавать модульные тесты, обеспечивая работоспособность системы на всех этапах. Заказчики тоже должны тестировать продукт, чтобы убедиться в его корректной работе.
- ❑ Рефакторинг. Модифицируйте код, не затрагивая “поведения” системы, с целью его упрощения, устранения повторов и улучшения “читабельности”.
- ❑ Связь с заказчиком. Заказчик всегда должен быть “доступен” для разработчиков. Он должен отвечать на все возникающие вопросы на протяжении всего процесса разработки.

Заказчик, на протяжении всего процесса реализации является полноценным участником разработки. По возможности это должен быть технически грамотный специалист, глубоко понимающий все бизнес-процессы, происходящие в данной предметной области. Не беспокойтесь о том, что он измучит вашу команду: чем дальше он будет вникать в работу, тем легче вам будет понять и откорректировать его требования. В экстремальном программировании особенно важен факт достижения желаемого результата. Если клиент существует в разработке с первого дня проекта, то крики: “Это не то, что я хотел!” — попросту исключены!

Создавая модульные тесты до начала разработки самих классов, вы на самом деле экономите время. При реализации крупного проекта с множеством отдельных классов такая экономия времени позволит досрочно завершить проект.

Модульное тестирование имеет и другие преимущества. Психологически легче на каждом этапе осознавать, что все созданные компоненты работают корректно. Кроме того, написание модульных тестов положительно влияет на общий стиль кодирования, что особенно важно для групповой разработки.

Кроме того, модульное тестирование обеспечивает выработку стандартов программирования на ранних стадиях проекта. При этом любой разработчик может модифицировать все части кода — корректность его работы проверяется модульным тестированием.

Самым известным аспектом экстремального программирования является возможность одновременной работы на одной рабочей станции двух специалистов: архитектора и разработчика. Один из них обычно использует клавиатуру и мышь, а второй — отслеживает его действия. Если два человека активно участвуют в разработке, то полученная система будет менее “субъективной” и более соответствующей принятым стандартам. Как говорится, одна голова хорошо, а две лучше. Наличие дополнительного разработчика вряд ли повлияет на качество программы, но значительно уменьшит количество ошибок.

Самый спорный из принципов экстремального программирования — это сверхурочная работа, когда программисты трудятся до полного изнеможения. Если проект правильно организован, достаточно профинансирован и процесс управления проектом хорошо наложен, необходимость в сверхурочной работе отпадает.

Когда использовать экстремальное программирование

Экстремальное программирование лучше всего применять для реализации проектов с высокой степенью риска. Возможно, существуют некоторые неопределённости, не позволяющие удовлетворить требования клиентов. Тогда предлагаемое решение имеет такую важность, что может поставить под вопрос саму реализацию проекта.

Группа разработчиков должна быть небольшой — не менее двух, но и не более одиннадцати человек. В процессе реализации могут быть задействованы все участники проекта.

Экстремальное программирование обеспечивает следующие преимущества.

- Повышает производительность группы разработчиков.
- Позволяет с легкостью удовлетворить все желания клиента.
- Позволяет реализовать высокие стандарты качества.
- Позволяет максимально точно выполнить требования клиента.
- Обеспечивает своевременность выполнения проекта.

Среди программистов ведутся дискуссии о преимуществах и недостатках экстремального программирования. Зачастую повышенные коммерческие требования не позволяют использовать эту методологию. Как и любая парадигма — это просто модель, созданная для того, чтобы знать, к чему стремиться. Дополнительную информацию об этом подходе и практические рекомендации по его использованию можно получить по адресу <http://www.extremeprogramming.org/>.

Управление изменениями

Практически в каждый проект по просьбе заказчика приходится вносить изменения. Управление этими изменениями является очень важным элементом реализации проекта. Не следует оценивать только коммерческое влияние изменений, необходимо также оценить неудобства, причиняемые модификацией программного обеспечения или архитектуры системы.

В этом разделе будут рассмотрены самые распространенные варианты изменений и способы их внесения.

Модификация требований

Это не только неизбежно, но и необходимо. Скорее всего, заказчик захочет внести изменения в документацию, поскольку с первого раза не всегда удается достичь компромисса.

Упорядочите все просьбы и пожелания, поступившие в течение 48 часов. Для внесения изменений используйте такие пакеты, как Microsoft Project или MrProject (<http://mrproject.codefactory.se>). Каждые 48 часов интегрируйте внесенные изменения в документацию и предоставляйте очередную версию спецификации заказчику.

Если какое-то из пожеланий не будет выполнено, не забудьте поставить в известность заказчика.

Изменение требований после подписания спецификации

Если клиент просит внести изменения после подписания спецификации, вы рискуете попасть в затруднительное положение. Для начала не переживайте. Подпись заказчика на спецификации программного продукта означает, что вашей вины здесь нет! Дополнительная работа может быть оплачена отдельно.

Прежде чем принимать решение о внесении изменений, необходимо посоветоваться со всей командой разработчиков. Постарайтесь определить, насколько вы продвинулись в процессе проектирования и реализации. Оцените объем сделанной работы и стоимость возврата к стадии составления спецификации. Эта работа должна быть оплачена дополнительно.

Даже если вы сами можете оценить работу, лучше получить подтверждение ее стоимости от независимого источника. Проявление гибкости в этой ситуации поможет сохранить хорошие отношения с заказчиком.

Если будет принято решение приступить к работе, необходимо внести все корректизы в течение короткого времени. При этом необходимо обсудить все вехи, связанные с внесением изменений. Убедитесь, что задуманное под силу вашей команде, и график ее работы окажется не очень плотным.

Конфликт из-за разницы толкования

Если заказчик усомнится в правильности толкования его требований, он может попросить переделать некоторые детали проекта. Не стоит говорить, что заказчик не прав, лучше попросить его выразить свои требования как можно точнее и по возможности избегать неоднозначности.

Если возникают проблемы, их необходимо обсудить с заказчиком. Внимательно выслушайте его и постараитесь понять его точку зрения. Проконсультируйтесь со своей командой по поводу времени, необходимого для внесения изменений.

Дефекты, обнаруженные клиентом

Детально обсуждайте с заказчиком все обнаруженные в системе дефекты. На этом этапе любое изменение, инициированное заказчиком, является дополнительной работой.

Реагируйте на все замечания заказчика, даже если окажется, что это не ошибка. Для поиска ошибок используйте систему Mantis, предоставьте доступ к этой системе заказчику, чтобы он мог самостоятельно находить ошибки в системе.

Обычно клиенты снисходительно относятся к дефектам, они понимают, что такое случается. Позаботьтесь, чтобы даже после окончания проекта вы могли быстро связаться с заказчиком и при необходимости помочь ему.

Резюме

В процессе реализации проекта приходится придерживаться некоторой методологии. В этой главе описаны два различных метода реализации проекта, а также несколько теоретических принципов, которые можно применять в процессе реализации системы.

Наконец, в этой главе было введено понятие управления изменениями и даны рекомендации по выходу из затруднительного положения.

21

Архитектура системы

Наверное, вы уже горите желанием приступить к разработке своего приложения. Вы спланировали систему, выбрали персонал, определились с методологией и составили детальную спецификацию. Однако необходимо рассмотреть еще один вопрос.

Как архитектор или менеджер проекта, вы интересуетесь в основном самой системой, а не ее развертыванием.

Однако, если хорошо отложенная система будет работать недостаточно эффективно (или вообще не будет работать), позор ляжет на ваши плечи.

В этой короткой главе вы ознакомитесь с системной архитектурой, узнаете, что означает этот термин, почему вопрос архитектуры является очень важным и как эффективно спроектировать архитектуру системы.

Что такое системная архитектура

Системная архитектура — это инфраструктура, поддерживающая функционирование разрабатываемого приложения.

Однако речь идет не только о требованиях к аппаратным средствам сервера. Архитектура охватывает вопросы настройки серверов, сети, соединений Интернет, брандмауэра, вопросы балансировки нагрузки и распределения компонентов программы между несколькими серверами.

Это достаточно обширная тема. В данной главе мы коснемся лишь верхушки айсберга, но она укажет вам правильное направление для размышлений.

Почему это важно

Правильная инфраструктура играет жизненно важную роль в работе приложения. При разработке программного продукта используется как минимум один сервер разработки. Одновременно он может служить и Web-сервером, и сервером базы данных. Если при разработке приложения используется не только язык PHP, на этой машине может быть установлен пакет Java SDK, Jakarta или любой другой сервер приложений. Если вы не первый год занимаетесь разработкой программных систем, то для поддержки прежних проектов на вашем сервере могут быть установлены средства PHP4.

Такое положение дел вас, наверное, вполне устраивает. В процессе разработки приложения доступа к Интернет не требуется, а пользователями приложения являются только разработчики.

Однако после передачи заказчику ситуация в корне изменяется. К вашему приложению могут одновременно обратиться сотни пользователей по высокоскоростному соединению. И приложение должно эффективно работать в таких условиях.

В одной из следующих глав будут рассмотрены вопросы обеспечения качества приложения, однако задачу можно существенно упростить, если спроектировать эффективную системную архитектуру, дополняющую и поддерживающую разработанную программную архитектуру приложения.

Что нужно сделать

Системная архитектура — это детальное отражение рабочей среды приложения. Вопросу системной архитектуры можно посвятить отдельный документ либо рассмотреть его в рамках существующей технической спецификации. Решения по системной архитектуре принимаются разработчиком в процессе обсуждений с заказчиком.

Если в технической спецификации указано, что приложение должно поддерживать не более x одновременных подключений, а система вышла из строя при подключении $x+1$ пользователя, вы можете смело смотреть в глаза заказчику, а ему придется выложить дополнительные деньги за расширение системы.

Подобные предположения и требования лежат в основе разработки системной архитектуры. Требования заказчика должны быть реализованы в виде эффективных системных решений.

При разработке системной архитектуры придется принимать решения по следующим вопросам.

- ❑ **Хостинг.** Где физически будет располагаться приложение? Насколько удобно помещение, оборудован ли сервер источником бесперебойного питания?
- ❑ **Соединение с Интернет.** Какая связь с Интернет требуется приложению? Кто ее будет обеспечивать? Сколько это будет стоить?
- ❑ **Серверы.** Сколько и каких серверов требуется для поддержки приложения?
- ❑ **Сеть.** Какую топологию сети использовать для поддержки серверов?
- ❑ **Надежность и избыточность.** Какой уровень доступности приложения необходим для его работы?
- ❑ **Поддержка.** Кто будет отвечать за поддержку каждого сервера и его программного обеспечения?
- ❑ **Безопасность.** Насколько безопасной является выбранная инфраструктура?

Каждый из этих вопросов будет детально рассмотрен ниже в этой главе, но сначала сосредоточимся на вопросах реализации требований заказчика к системной архитектуре.

Эффективная реализация требований

Заказчик вряд ли сможет ответить на прямые вопросы по системной архитектуре.

Иногда приходится разрабатывать проекты, в которых заказчик полностью определяет системную архитектуру, в которой должно работать приложение. Однако

гораздо чаще системную архитектуру придется разрабатывать на основе неявно сформулированных требований заказчика.

Вам придется задать заказчику ряд вопросов и использовать ответы для выработки системных решений. Какие же вопросы необходимо задать?

Хостинг, соединения, серверы и сеть

Первым вопросом должен быть следующий: “Работаете ли вы с определенным провайдером услуг Интернет?”.

К сожалению, ответ на этот вопрос в большинстве случаев утвердительный. Обычно он сопровождается объяснениями типа: “Мне бы хотелось разместить приложение на сервере этого провайдера, поскольку он нам очень помогает. Так, на прошлой неделе он удалил вирус с моего переносного компьютера”.

Затем вы спросите: “И сколько стоят его услуги?”. Если в ответ вы услышите: “Я плачу \$300 за неограниченное дисковое пространство”, то у вас, наверное, засосет под ложечкой.

Как профессионал по языку PHP, вы знаете, что размещение приложений на сервере у подобных провайдеров аналогично просьбе к местному механику настроить автомобиль для участия в гонках формулы 1.

Будьте вежливы на ранних стадиях проекта и попытайтесь убедить своего заказчика выбрать серьезного провайдера.

Обычно мы используем следующий дипломатический подход: “Хорошо, завтра я свяжусь с вашим провайдером и объясню ему требования к вашему приложению”. Через несколько дней вы с чистым сердцем можете сообщить заказчику: “Я разговаривал с вашим провайдером и, вероятно, он не сможет поддерживать наше приложение”. Вашему заказчику ничего не остается, кроме как задать встречный вопрос: “А что вы порекомендуете?”. Теперь вы можете посоветовать нужного провайдера, и заказчик прислушается к вашим рекомендациям. Такой подход срабатывает всегда.

Обсуждая вопросы хостинга, следует лишь определиться с ожидаемым трафиком. Это требование не только позволит определить ширину канала, но и примерное количество и конфигурацию серверов.

Если заказчик затрудняется ответить на этот вопрос, ему нужно помочь. Спросите его, насколько велика его база данных и как часто пользователям придется обращаться к приложению. Тогда вы сможете составить представление о количестве одновременных подключений к системе.

Следует также выяснить типы предполагаемых соединений.

Надежность и избыточность

Не следует задавать заказчику вопрос: “Хотите ли вы обеспечить избыточность приложения?”.

Лучше спросить: “Насколько критично, если сервер выйдет из строя на несколько часов?”

Для многих заказчиков это не вопрос. В таком случае можно особо не беспокоиться о надежных источниках бесперебойного питания, безотказных серверах баз данных и т.п. Однако можно переформулировать вопрос: “А что, если в результате неисправности сервера часть данных будет потеряна?”. Вы увидите, что настроение заказчика резко изменится и он будет очень озабочен потерей данных. Не забывайте об этом. Избыточность при обеспечении работы системы существенно отличается от избыточности при хранении данных. Об этом речь пойдет в следующих разделах данной главы.

Поддержка

Необходимо выяснить, кто будет владельцем серверов приложения. Речь идет не о юридической собственности, поскольку реальным владельцем останется заказчик.

Речь идет об ответственности за работоспособность сервера, перегрузку ядра операционной системы и поддержку программного обеспечения. Заказчику следует задать вопрос: “Время от времени программное обеспечение на серверах необходимо обновлять. Кто будет за это отвечать?”.

Заказчику можно предложить свои услуги по поддержке серверов.

Предлагая свои услуги, не следует пытаться продать их во что бы то ни стало. Необходимо задать заказчику вопросы, помочь ему определить слабые места и предложить очевидные решения для выявленных проблем.

Безопасность

Этот вопрос довольно важен заказчику. От своих друзей и коллег он наверняка слышал леденящие душу истории о вторжениях в систему, а может быть, и сам подвергался атакам хакеров.

Пообещайте заказчику, что система будет надежно защищена. Здесь не нужно задавать никаких вопросов, поскольку на любой вопрос о необходимости обеспечения безопасности заказчик ответит: “Да, конечно”.

Проектирование среды

Выяснив требования заказчика, необходимо сформулировать предложение по системной архитектуре.

Рассмотрим каждый из элементов системной архитектуры.

Хостинг и соединения

Практически в любой современной системе потребуется обеспечить соединение через Интернет и безопасную среду для хостинга серверов заказчика.

Несомненно, лучшим решением этого вопроса является центр данных. Такие центры предназначены для обеспечения хороших условий работы серверов и другого оборудования, а также сетевого соединения с ними.

Поскольку такие центры данных предназначены только для удаленного доступа, они не предназначены для длительного пребывания людей.

Вычисление параметров канала

Стоимость услуг провайдера Интернет определяется либо пропускной способностью канала в мегабитах (так называемый показатель CIR — Committed Information Rate), либо ежемесячным трафиком в гигабайтах. Первый способ почти всегда дороже, но именно он используется в больших системах. Он позволяет максимизировать трафик для данной полосы. Чтобы вычислить показатель CIR, необходимо учитывать число одновременных подключений к приложению.

Допустим, приложение электронной почты должно поддерживать тысячу одновременных подключений. Это значит, что в каждый момент времени данное приложение будет использовать тысяча пользователей. Это не означает тысячу одновременных соединений. Это отличие является чрезвычайно важным. Каждый сеанс состоит из множества HTTP-запросов, между которыми существуют паузы, необходимые для чтения результатов обработки запроса. То есть, соединения устанавливаются только в течение небольшого промежутка времени.

Для прочтения одной страницы типичного Web-узла пользователю нужно 30 секунд, а для работы с одной страницей Web-приложения — порядка 60 секунд.

Если предположить, что каждая страница загружается 10 секунд, то при работе Web-приложения данные передаются по сети в течение 10 секунд, за которыми следуют 60 секунд работы с приложением без передачи данных.

Несложные расчеты позволяют оценить, что передача данных в рамках сеанса занимает 15% времени. Значит, число одновременных соединений составляет примерно 15% от количества одновременных подключений к Web-приложению (а для простого Web-узла — 2%).

Значит, тысяче одновременных сеансов соответствует 150 одновременных соединений. Но как определить, какая ширина канала нужна для обеспечения этих требований?

Здесь нужно привлечь знания заказчика. Именно заказчик знает, сколько пользователей приложения работают в офисах и имеют линии связи Т1. Возможно, удаленные пользователи используют коммутируемые соединения.

В идеале нужно попросить доступ к журналам сервера для прошлых проектов и сделать соответствующие оценки.

Допустим, 30% из 200 пользователей работают с шириной канала 512К, а 70% связываются с сервером через коммутируемые соединения с пропускной способностью 48К. Тогда для обеспечения максимальной скорости работы потребуется около 40 Мбит в секунду. Это не очень дешево.

К счастью, тысяча одновременных сеансов — это достаточно жесткое требование, относящееся только к очень большим Web-приложениям. Необходимо максимально точно вычислить количество одновременных подключений. Это можно сделать на основе данных о числе постоянных пользователей приложения. Если, например, Web-приложение рассчитано на пять тысяч постоянных пользователей, каждый из которых ежедневно работает с приложением порядка 20 минут, а вероятностное распределение обращений пользователей к системе является нормальным с центром в 6 часов вечера, то число одновременных подключений составляет порядка 2% от количества пользователей — т.е. 100. Для такого количества одновременных подключений получаем более приемлемую пропускную способность, равную 4 Мбита.

Целесообразно построить графики продолжительности сеансов и их распределение в течение рабочего дня. Это позволит вычислить пиковые значения нагрузки на систему.

Вычисление ежемесячного трафика

Для заданной полосы можно вычислить максимальный объем передаваемых данных за месяц. Для некоторых приложений оплата за трафик может оказаться выгоднее, чем плата за пропускную способность канала.

Для вычисления ежемесячного трафика количество сеансов пользователей за этот период нужно умножить на передаваемый объем данных в течение каждого сеанса.

Если типичный сеанс содержит 10 запросов на страницу размером 50К, то в течение этого сеанса передается 0.5 Мбайт информации. В условиях предыдущего примера

5000 постоянных пользователей ежедневно передают 2.5 Гбайт данных. За месяц это составит 75 Гбайт.

Проконсультируйтесь у провайдера, что обойдется дешевле: 75 Гбайт в месяц или оплата канала шириной 4 Мбита?

Дополнительные затраты

Основные затраты на инфраструктуру определяются стоимостью и количеством серверов. Однако для сетевых приложений придется также приобрести и сетевое оборудование.

Если вы планируете расширять свое приложение, то при выборе оборудования необходимо учитывать возможности расширения, а не ограничиваться требованиями сегодняшнего дня.

Серверы

Требования к серверам определить несколько сложнее.

Вам понадобится как минимум один Web-сервер и один сервер баз данных. Для небольших приложений эти функции может выполнять один физический сервер, но на СУБД MySQL и PostgreSQL плохо отражается запуск других процессов. Поэтому в результате может пострадать все приложение.

Возникает вопрос: сколько серверов необходимо для работы приложения. Ответить на этот вопрос позволит тестирование нагрузки. Если окажется, что для работы приложения требуется несколько серверов, полезную информацию вы почерпнете из следующего раздела.

Использование нескольких Web-серверов

Развёртывание нескольких Web-серверов позволяет существенно повысить возможности приложения по числу обслуживаемых клиентов.

Существует несколько способов балансировки нагрузки между Web-серверами: на основе службы DNS и с использованием дополнительных устройств.

Балансировка нагрузки на основе DNS выполняется очень легко. С помощью брандмауэра каждому Web-серверу присваивается отдельный внешний IP-адрес, а затем создается запись A, указывающая не на один адрес, а на группу адресов.

Например, если обычно в службе DNS используется запись вида

```
$ORIGIN example.com.
www           IN      A      192.168.111.222
```

то для этой записи можно задать набор адресов.

```
$ORIGIN example.com.
www           IN      A      192.168.111.222
www           IN      A      192.168.111.223
www           IN      A      192.168.111.224
```

При попытке клиента обратиться к Web-серверу для него будет случайным образом выбран адрес из указанного набора. Соответственно, разные клиенты будут использовать разные Web-серверы.

Недостатком этого подхода является однократное разрешение адреса Web-браузером. Обычно Web-браузер кеширует этот адрес и в рамках одного сеанса работает с одним Web-сервером. Это неплохо, но в данном случае не выполняется реальная

балансировка нагрузки. Может оказаться, что один из Web-серверов непрерывно занят обслуживанием интенсивных сеансов, в то время как другие простоявают.

Гораздо лучше каждый запрос направлять на отдельный сервер либо использовать более сложные подходы, связанные с реальным анализом нагрузки на каждый сервер.

Недостатком второго подхода является необходимость покупки дополнительного оборудования — балансировщика нагрузки. Он принимает все HTTP-запросы и передает их на нужный Web-сервер. Такой подход обеспечивает равномерную нагрузку на все серверы системы без уведомления клиента. Однако подобные балансировщики нагрузки довольно дороги. На момент написания этой книги их стоимость составляла около \$20000.

Существуют и бесплатные альтернативы. Например, можно поэкспериментировать с системой OpenBSD.

При использовании нескольких Web-серверов необходимо продумать стратегию синхронизации данных. Подобные стратегии более подробно описаны в главе 24.

Использование нескольких серверов баз данных

На первый взгляд, эта идея кажется очевидной.

Такое решение повышает надежность системы, поскольку при использовании одного сервера базы данных он становится узким местом.

Проблема состоит в том, что базы данных предназначены не только для считывания информации. Зачастую к ним выполняются запросы на обновление, удаление или добавление данных.

Однако не все так плохо. При выполнении запроса его можно проанализировать и для запросов на чтение выполнять балансировку нагрузки.

Запросы на запись необходимо реплицировать на все серверы. Если некоторый сервер в данный момент не может выполнить запрос на запись, данные необходимо буферизировать и повторить попытку позднее.

Серьезные производители баз данных предусмотрели решение подобной проблемы. К счастью, ее позволяют решать и бесплатные СУБД PostgreSQL и MySQL. Репликация данных — это серьезный вопрос. Детальная информация по данному вопросу для СУБД PostgreSQL содержится по адресу <http://gborg.postgresql.org/project/slony1/projdisplay.php>, а для MySQL по адресу <http://dev.mysql.com/doc/mysql/en/Replication.html>.

Сеть

Реализация требований к сети, на первый взгляд, кажется простой задачей, которая позволит позднее сэкономить время и деньги.

Рассмотрим трафик через каждый интерфейс к системе и попытаемся его спланировать. Например, приложение баз данных связано с передачей больших объемов данных на сервер базы данных. Поэтому для каждого Web-сервера, связанного с сервером баз данных, можно предусмотреть отдельный интерфейс.

Такая конфигурация довольно типична. В системе устанавливается один переключатель, связанный со всеми Web-серверами и внутренним интерфейсом брандмауэра. Переключатель баз данных связывается со вторичным сетевым адаптером Web-сервера и с картой сетевого интерфейса каждого сервера базы данных. В результате трафик между Web-серверами и серверами баз данных не пересекается с реальным Web-трафиком. При этом повышается производительность работы системы.

При использовании дорогих переключателей можно даже применять виртуальные сети и разделить 24 порта на два виртуальных переключателя (по 12 в каждом). Один из них можно использовать для Web-сервера и брандмауэра, а второй — для серверов баз данных и вторых сетевых карт Web-серверов.

При такой конфигурации повышается уровень защищенности базы данных, поскольку она не имеет выхода в Интернет.

Дополнительная память

При планировании конфигурации Web-серверов и серверов баз данных необходимо обеспечить некоторую избыточность для повышения надежности работы приложения. Подобная избыточность позволит также предотвратить потерю данных, что чрезвычайно важно с коммерческой точки зрения.

Поэтому при выборе серверов отдавайте предпочтение RAID-массивам SCSI-дисков либо архитектуре IDE RAID.

Не пытайтесь максимизировать дисковое пространство с помощью архитектуры RAID 0. Отдавайте предпочтение RAID 5. Это обеспечивает необходимую избыточность данных. Вышедший из строя диск можно просто заменить, а контроллер RAID-массива быстро подхватит новый диск.

Тематика RAID практически неисчерпаема. Более подробная информация по этому вопросу содержится по адресу http://www.uni-mainz.de/~neuffer/scsi/watch_is_raid.html.

Не следует выпускать из виду политику резервного копирования базы данных и каталогов с данными на Web-серверах. При этом вполне подойдет ленточное устройство.

Поддержка

Базовую поддержку серверов зачастую обеспечивает провайдер. Однако эти функции может выполнять и разработчик системы.

Если вы беретесь за поддержку серверов, необходимо составить список задач, выполняемых еженедельно для обеспечения их эффективной работы. К этим задачам относятся следующие.

- Обеспечение работы последней версии ядра.
- Поддержка новейших версий Apache и PHP.
- Обновление библиотек, необходимых для работы модуля PHP.
- Поддержка необходимого дискового пространства, отслеживание размеров файлов журналов. Например, сервер Apache для каждого файла журнала позволяет использовать не более 2 Гбайт.

Безопасность

И наконец, системная архитектура должна обеспечивать надежную защиту всей сети от нежелательных вторжений и атак.

Для этого абсолютно необходим брандмауэр. Абсолютно недопустимо обеспечивать прямой выход серверов в Интернет. Это приведет только к возникновению проблем.

Лучше купить дополнительное устройство, а не просто воспользоваться программной реализацией брандмауэра для систем Linux или FreeBSD. Можно посоветовать брандмауэр серии Cisco PIX. Для небольших узлов подойдет PIX-501 стоимостью \$300.

Не забывайте также обеспечивать доступ к Интернет для Web-серверов через протокол NAT. Если приложение связано с отправкой электронных сообщений, на Web-сервере придется открыть порт 25 для протокола SMTP.

Резюме

В этой главе вы познакомились с понятием системной архитектуры приложения, узнали, какие вопросы следует задать заказчику для определения требований к системной архитектуре.

Затем было рассказано, как использовать эту информацию при выработке технических решений.

Документ с описанием системной архитектуры можно составить отдельно либо сделать его частью технической спецификации. Проверить теоретические расчеты можно с помощью тестирования нагрузки. Однако лучше несколько ошибиться в прогнозах, чем не делать их вообще.

В следующей главе будут рассмотрены методологии управления проектами и вопросы применения шаблонов.

22

Разработка средства автоматизации торговли

Освоив принципы работы различных технологий, можно приступать к их использованию для реализации проектов. В данной главе будет рассмотрено, как разрабатывать программное обеспечение с самого начала. И поскольку рассматриваемый проект не является очень большим, для его разработки будут использованы различные приемы экстремального программирования (XP — eXtreme Programming).

С безграничным желанием начать новый и важный проект можно очень легко перейти к вопросу о том, как следует создавать код. Однако для начала необходимо просто понять, что нужно сделать в рамках данного проекта.

Поэтому отставьте компьютер в сторону, забудьте об учетных записях, паролях, базах данных и даже PHP. На некоторое время все это вам не понадобится. Вместо всего этого вы будете пользоваться “технологией”, на совершенствование которой человечество потратило тысячелетия, — бумагой.

Да, бумага — это технология. Она является доступной, недорогой, “читабельной” (даже если намокла), не требует энергетических ресурсов (кроме как для чтения в темноте). Один и тот же “носитель” поддерживает разнообразные режимы: “только чтение”, “перезапись”, цветные и черно-белые чернила. Если лист бумаги разрезать на небольшие индексные карточки, их можно эффективно использовать для представления классов PHP. При этом их легко переносить с одного места на другое, можно прикреплять к стене, обмениваться ими, скреплять, сортировать и т.д.

Таким образом, процесс проектирования и разработки проекта значительно зависит от умения общаться с людьми, умственных способностей и тысячелетней технологии (бумаги), а не только от программных и аппаратных средств.

Начало проекта: понедельник

На работу вы приходите достаточно отдохнувшим и счастливым. И несмотря на то, что специалисты из отдела информационных технологий компании Widget World усиленно настраивают ваш компьютер, в вашем ящике имеется все необходимое для сегодняшней работы:

- ручки;
- учетные карточки размером 3×5;
- учетные карточки размером 4×6.

Менеджер по разработке автоматизированной торговой системы компании Widget World Бриджит (Bridget) сидит рядом с вами и готова начать работу.

Она поясняет, что Эдвина (Edwina), менеджер по продажам в восточном регионе, сегодня будет в офисе. Поэтому Бриджит назначила ей встречу с вами. Продавец Харольд (Harold) будет работать в офисе последующие две недели, поэтому он также будет доступен для общения. Вэйд (Wade), бухгалтер компании Widget World, также будет присутствовать на встрече. Кроме того, он доступен каждый день с 9:30 утра и до обеда.

Итак, на встречу. И не забудьте свои карточки.

Слушайте внимательно

После представления каждого из участников собрания вы объясняете, что находитесь здесь с целью собрать *сценарии* (или варианты) (*stories*) использования новой разрабатываемой системы.

Используя технологию ручки и бумаги, задокументируйте все сценарии. Каждый из них следует помещать на отдельной карточке. При этом добивайтесь того, чтобы в процессе обсуждения участвовали все представители компании Widget World — Вэнди, Вэйд и Эдвина. Безусловно, у них могут возникнуть разнообразные вопросы касательно того, что технически возможно, а что нет, кто должен подписывать документы и как осуществляется весь процесс автоматизации.

Помните, что каждый сценарий или вариант использования системы необходимо разбивать на подсценарии (подзадачи) примерно одинакового размера. Эта задача не из легких. Дело в том, что размер карточки будет пропорционален усилиям (времени), которые необходимо затратить на реализацию данного сценария.

Определение времени, которое необходимо для реализации сценария, не является тривиальной задачей. В общем случае это время варьируется от полудня до одной-двух недель. При этом желательно, чтобы все сценарии были примерно одинакового размера. Все это будет использовано впоследствии для определения объема работ и оценки затрат на решение последующих задач.

Рассмотрим, например, следующий сценарий: “Запуск ракетоносителя”. Данный сценарий можно разбить на части меньшего размера (подсценарии), такие как “Отделяемый отсек должен быть запущен в течение 10 секунд после того, как будет закрыт люк аварийного выброса”.

Таким образом, результатом вашего первого рабочего дня будет набор сценариев, которые более подробно рассмотрены ниже. В общем случае не обязательно записывать имена людей (авторов), описавших тот или иной сценарий. Однако для достоверности описания всего процесса разработки программного обеспечения приведем и авторов вариантов использования системы. Очень часто неплохой отправной точкой начала разработки проекта является анализ конкретных форм описания предметной области (рис. 22.1), которые используются в процессе продажи товаров. Именно работа с ними и соответствует рассмотренной ниже последовательности действий.

- Сценарий 1. Эдвина: “Мы отслеживаем контакты продавцов с помощью специальной формы (см. рис. 22.1), обмен которыми осуществляется по факсу. Было бы неплохо иметь доступ к этой информации в электронном формате”.

Еженедельный отчет о контактах компании Widget World

Имя сотрудника: _____ Отдел: _____
 Номер сотрудника: _____ Дата: _____

Важные дистрибуторы и покупатели:

Компания: _____ Контактное лицо: _____
 Отзыв: _____ Город: _____
 Пожелания: _____ Страна: _____

Полученные результаты:

Компания: _____ Контактное лицо: _____
 Отзыв: _____ Город: _____
 Пожелания: _____ Страна: _____

Полученные результаты:

Рис. 22.1.

- ❑ Сценарий 2. Вэйд: “Процесс сбора информации о командировочных расходах продавцов занимает много времени и подвержен ошибкам. Мы бы хотели получать эти данные еженедельно с расшифровками по дням и категориям”.
- ❑ Сценарий 3. Харольд: “Иногда мне нужно обосновать определенные расходы. Например, я трачу \$30.00 на услуги специальной службы доставки FedEx. Я хотел бы иметь возможность добавлять соответствующие комментарии в нижнюю часть отчетной формы”.
- ❑ Сценарий 4. Вэйд и Харольд: “Отчет о расходах должен содержать расшифровки по различным категориям в виде таблицы”.
- ❑ Сценарий 5. Вэйд: “Я хотел бы получать соответствующее уведомление (с итоговыми результатами) при подаче отчета о расходах”.
- ❑ Сценарий 6. Эдвина: “Я хотела бы получать соответствующее уведомление (с итоговыми результатами) при подаче отчета о контактах продавца”.

- Сценарий 7. Харольд: “Было бы неплохо, чтобы каждый продавец также получал бы такое уведомление с целью перепроверки”.
- Сценарий 8. Эдвина: “Почтовая служба (экспедиция) должна получать соответствующее уведомление, если отчет о контактах содержит запрос на документацию”.
- Сценарий 9. Эдвина и Харольд: “Каждый продавец должен иметь свою учетную запись”.
- Сценарий 10. Вэйд: “Я не выдаю аванс наличными, однако должен их отслеживать. Поэтому в отчет о расходах следует добавить графу “Исключая аванс наличными”.
- Сценарий 11. Харольд: “Аванс наличными выдают из многих источников. Поэтому было бы неплохо обеспечить возможность продавцу контролировать остаток (автоматически не позволять дебетовать определенный счет; позволять вносить остаток в отчет о расходах)”.
- Сценарий 12. Вэйд: “Я хотел бы иметь возможность сохранять и просматривать информацию о выдачах авансовых средств по каждому продавцу”.
- Сценарий 13. Вэйд: “Я хотел бы также иметь возможность получать (экспортировать) данные из отчетов о расходах продавцов в свою специальную таблицу”.

После общения с представителями компании Widget World у вас появится первая версия плана для реализации видения системы, предназначеннной для повышения производительности работы компании Widget World.

На этом первый день будем считать оконченным. Сценарии, которые были собраны в течение этого дня, являются небольшими кирпичиками плана, который основан на анализе требований клиентов. Безусловно, еще не настало время переходить к реализации программного обеспечения, поскольку разработка этого плана еще не завершена.

Оценка трудоемкости реализации сценариев

После того как вы собрали сценарии использования разрабатываемой системы и разбили их примерно на одинаковые части, настало время оценить трудоемкость их реализации. Для этого каждому сценарию можно присвоить число, которое будет характеризовать сложность его реализации. Для простого проекта можно ограничиться следующей градацией оценок (уровнями сложности):

- 1 — легкий;
- 2 — средний;
- 3 — сложный.

Другой подход заключается в том, чтобы присвоить каждому сценарию относительный вес в единицах времени *идеальной реализации* (perfect-engineering time units). Время “идеальной реализации” соответствует количеству часов, дней или недель, которые необходимо потратить для выполнения некоторого задания, с учетом того, что на каждом шагу вас поджидают трудности, но при этом ваши действия безупречны.

В качестве примера рассмотрим сценарий 9: “Каждый продавец должен иметь свою учетную запись”. Проанализируем этот сценарий. Безусловно, учетные записи не должны зависеть от встроенных средств защиты Web-сервера Apache. Однако при этом параметры учетной записи могут использоваться для инициализации сеансов PHP или изменения URL-адресов и т.п. Посмотрим на это с другой стороны: сеансы

реализовать не так трудно, особенно, если вы занимались этим ранее. Поэтому, используя простую шкалу оценивания, данный сценарий можно отнести к первому уровню — легкий. Если вести учет в единицах “идеальной реализации” (в полуднях), то и в этом случае трудоемкость реализации данного сценария можно считать легкой. Однако, если единицы “идеальной реализации” будут измеряться, например, в неделях, а некоторый сценарий можно будет реализовать в течение нескольких часов, то этот сценарий нельзя будет корректно сравнить с трудоемкостью других сценариев. Таким образом, время, необходимое для реализации сценариев в тех или иных единицах, должно быть распределено равномерно по всем сценариям.

Рассмотрим несколько других сценариев.

- Сценарий 13: “Я хотел бы также иметь возможность получать (импортировать) данные из отчетов о расходах продавцов в свою специальную таблицу”.

Вэйду необходимо иметь доступ к данным отчетов о расходах продавцов и возможность импортировать их в систему обработки таблиц.

Допустим, единицей измерения является время *идеальной реализации* в полуднях. В данном случае термин “идеальный” имеет либеральный, а не абсолютно “идеальный” характер. Так, автором данного варианта использования разрабатываемой системы является бухгалтер компании Widget World Вэйд. И данный сценарий соответствует только его роли. Это означает, что необходимо вводить разные категории пользователей, например, Вэйд/не-Вэйд, а еще лучше — бухгалтер/пользователи. Этую информацию необходимо записать на карточке.

Для импортирования данных в таблицы можно воспользоваться файлами в формате CSV (comma-separated value — значения, разделяемые запятой), в которых информация разделяется запятыми. С технической точки зрения обработка таких файлов не является сложной.

Таким образом, добавление категорий пользователей и импортирование данных из формата CSV займет примерно один день “идеальной реализации”. Или в полуднях — 2 балла.

Рассмотрим теперь сценарий 5: “Вэйд хотел бы получать соответствующее уведомление (с итоговыми результатами) при получении отчета о расходах”.

Выполнение данного сценария напрямую зависит от сценария 2: “Процесс сбора информации о командировочных расходах продавцов...”. Пометьте наличие такой зависимости на соответствующей карточке и оцените этот сценарий как “несколько более простой, чем сценарий 2”. Точная оценка будет получена только после оценки сценария 2.

Сценарий 6: “Эдвина хотела бы получать соответствующее уведомление (с итоговыми результатами) при получении отчета о контактах продавца”. Заметьте, что данный сценарий похож на сценарий 5. Пометьте это на карточке.

Проанализируйте аналогичным образом оставшиеся сценарии. Если необходимо, делайте на карточках заметки о зависимостях или о дополнительных свойствах. Кроме того, разбивайте сценарии на подсценарии в тех случаях, когда они слишком велики, или объединяйте, если они слишком малы.

В результате анализа карточек получится примерно следующий результат.

- Сценарий 1. “Отслеживать контакты продавцов с помощью специальной формы, обмен которыми осуществляется по факсу”.
- Базовая информация о работе продавцов (1 балл).
- Пара “дистрибутер/заказчик” может быть уникальной.
- Между продавцом и покупателем существует связь “один ко многим” (1 балл).

К этому следует добавить проверку работоспособности Web-форм и т.п. (1 балл). В результате данный сценарий будет оценен в 3 балла или два дня “идеальной реализации”.

- Сценарий 2. “Еженедельный сбор информации о командировочных расходах продавцов с расшифровками по дням и категориям”.
- Результатом данного процесса является отчет о командировочных расходах.
- Между продавцом и покупателем существует связь “один ко многим”.
- Вэйд указывает на то, что данный отчет должен быть составлен в виде таблицы. Это не тривиальная задача (4 балла).
- Расшифровка по категориям (1 балл).
- Проверка работоспособности (2 балла).

Всего 7 баллов.

- Сценарий 3. “Наличие комментариев в нижней части отчетной формы”.
- Должен быть включен в сценарий 2.
- Да, такой сценарий усложняет взаимосвязи. Теперь к еженедельной отчетной форме необходимо привязывать комментарии (рис. 22.2).
- Данный сценарий следует рассматривать как отдельный, но зависимый от сценария 2.

Продавец							Начало недели: 5 янв. 2004
	ВС	ПН	ВТ	СР	ЧТ	ПТ	СБ
Завтрак	1						
Ленч	2						
Обед	3						
Сумма	6						
Комментарии: ...							

Рис. 22.2.

Всего 2 балла.

- Сценарий 4. “Отчет о расходах должен содержать расшифровки по различным категориям в виде таблицы”.
- Несмотря на то, что это лишь разработка клиентской части, она займет некоторое время.

Всего 4 балла.

- Сценарий 5. “Уведомление бухгалтера (с результирующими раскладками) о том, когда подан отчет о расходах”.
- Посредством электронной почты.
- Настройка электронной почты, управление и т.д. (1 балл).

- Непосредственная работа: 1 балл.
- Зависимость от сценария 2.

Всего 2 балла.

- Сценарий 6. “Уведомление менеджера по продажам (с итоговыми результатами) о том, когда подан отчет о контактах продавца”.
- Каждый менеджер по продажам связан со многими продавцами.
- Между ними существует отношение “один ко многим”: 2 балла.
- Слабая зависимость от сценария 5: 1 балл.

Всего 3 балла.

- Сценарий 7. “Уведомление всех продавцов о деталях отчетов о расходах”.
- Если зависит от сценария 2: 2 балла.
- Если зависит от сценария 5: 1 балл.
- Сценарий 8. “Почтовая служба (экспедиция) должна получать соответствующее уведомление, если отчет о контактах содержит запрос на документацию”.
- Условное уведомление: 1 балл.
- Если зависит от сценария 2 или 5: 1 или 2 балла.
- Сценарий 9. “Каждый продавец должен иметь свою учетную запись”.
- Аутентификация: 2 балла.
- Авторизация: 2 балла.

Всего 4 балла.

- Сценарий 10. “Добавление графы “Исключая аванс наличными” в отчетную форму”.
- Те же проблемы, что и в сценарии 3 (с добавлением комментариев в нижней части отчетной формы)
- Зависимость от сценария 2.
- Отсутствие дебета счетов; нет необходимости проверять счета.
- Данные хранятся в исходном виде, а не вычисляются.

Всего 2 балла.

- Сценарий 11. “Обеспечение возможности продавцу контролировать остаток (автоматически не дебетовать определенный счет; позволять вносить остаток в отчет о расходах)”.
- Зависимость от сценария 10.
- На самом деле данный сценарий не является сценарием использования данной системы. Удалите его и добавьте заметку “отсутствие дебета счетов” на карточке для сценария 10.

Всего 0 баллов.

- Сценарий 12. “Возможность сохранения и просмотра информации о выдачах авансовых средств по каждому продавцу”.
- Зависимость от сценария 10.
- Этот сценарий является тривиальным; для него добавьте заметку “постоянный”.

Всего 0 баллов.

- Сценарий 13. “Возможность экспорта данных из отчетов о расходах продавцов в специальную таблицу”.
- Только для бухгалтера: 1 балл.
- Непосредственная работа: 2 балла.

Всего 3 балла.

Итак, проанализированы основные сценарии использования новой системы и сделан первый шаг для оценки трудоемкости их реализации. В общем случае этот шаг может состоять из нескольких итераций, в рамках которых заказчикам необходимо будет задать несколько уточняющих вопросов.

- Можно ли отправлять уведомления по электронной почте?
- Действительно ли существует три категории пользователей: менеджеры, продавцы и бухгалтеры?
- Может ли бухгалтер использовать формат CSV для получения данных из отчетов о командировочных расходах?
- По каким признакам бухгалтеру необходимо специфицировать отчеты? По имени продавца или по имени продавца и дате, или по менеджерам по продажам?

После получения ответов на эти вопросы ситуация не станет менее запутанной и туманной. Вы даже не знаете, поддерживает ли система исходящие сообщения по электронной почте? И если да, то как это делается на PHP?

Уточнение оценок

Побеседуйте со своими заказчиками и постарайтесь получить ответы на новые вопросы. Очень важно, чтобы заказчик выделил постоянных представителей для взаимодействия с командой проекта, так как не хотелось бы отрывать сотрудников от своих дел. Поскольку каждый из представителей заказчика (Вэнди, менеджер проекта, Вэйд, бухгалтер, Эдвина, менеджер по продажам, и Харольд, продавец) впоследствии будет пользоваться системой, они тоже являются участниками проекта.

Тем временем были получены ответы на новые вопросы.

- Вопрос. Можно ли отправлять уведомления по электронной почте?
- Ответ. Да, можно. В свою очередь любые результирующие документы можно отправлять во вложении.
- Вопрос. Действительно ли существует три категории пользователей: менеджеры, продавцы и бухгалтеры?
- Ответ. Да.
- Вопрос. Могут ли бухгалтеры использовать формат CSV для получения отчетов о командировочных расходах?
- Ответ. Да, они всегда так поступают.
- Вопрос. По каким признакам бухгалтер должен классифицировать отчеты? По имени продавца или по имени продавца и дате, или по менеджерам?
- Ответ. Достаточно по имени продавца и дате.

Таким образом, осталось выяснить только технические подробности.

- Как отправлять сообщения по электронной почте с использованием PHP?
- Как отправлять вложения вместе с сообщениями?

Блицзадание

Безусловно, заказчики не могут дать ответы на технические вопросы, возникающие при разработке. Это уже область вашей компетенции. И в данном случае у вас недостаточно информации для того, чтобы точно оценить все сценарии.

Поэтому настало время выполнить домашнее задание, так называемое *блицзадание* (spike). Несмотря на то, что это задание не имеет прямого отношения к рассматриваемой теме, оно поможет узнать, насколько вы смекалисты.

Отложите на время все дела и попытайтесь ответить на все возможные вопросы “как” и “если”. Это может занять некоторое время, возможно, день-два. Но в данном случае ответ найти несложно. Просмотрев соответствующую документацию по программному интерфейсу приложения PHP, можно найти следующий фрагмент.

```
mail("wade@widgetworld.com", "Email Spike Test", "One\nTwo\nThree");
```

То есть, с помощью языка PHP можно легко отправлять сообщения по электронной почте. Единственное, что необходимо сделать, — это проверить работоспособность данной функции или уточнить ее параметры.

Что касается отправки вложений, то здесь не все так просто. В PHP не существует какой-то одной функции, посредством которой можно было бы помещать вложения в сообщения. Возможно, это связано с тем, что набор стандартов MIME достаточно подробно задокументирован в серии документов RFC, в том числе RFC 822 (заголовки сообщений), 2045 (набор стандартов для передачи мультимедийной информации посредством электронной почты MIME) и 2046 (типы MIME). Тщательное изучение данных документов и спецификаций, а также небольшие усилия по программированию приведут к созданию следующих строк кода.

```
$mime_boundary = "<<<----+X[" . md5(time()) . "]";  
$headers .= "MIME-Version: 1.0\r\n";  
$headers .= "Content-Type: multipart/mixed;\r\n";  
$headers .= " boundary=\"$mime_boundary.\\"";  
  
$message .= "Это сообщение в формате MIME из нескольких частей. \r\n";  
$message .= "\r\n";  
$message .= "--".$mime_boundary."\r\n";  
  
$message .= "Content-Type" text/plain; charset=\"iso-8859-1\"\r\n";  
$message .= "Content-Transfer-Encoding: 7bit\r\n";  
$message .= "\r\n";
```

Данный фрагмент кода соответствует набору стандартов RFC, определяющих заголовки сообщений MIME (строки, заданные в переменной \$headers) и содержимое непосредственно перед отправкой сообщения почтовому агенту MTA (Mail Transfer Agent). Изучение документов RFC — занятие не из простых. Поэтому обязательно протестируйте работу данного кода в реальной системе, используя то же программное обеспечение, что и ваши заказчики.

Таким образом, выполнение блицзадания позволило “бросить” 1 балл.

Советы по оцениванию сценариев

После решения некоторых технических вопросов можно продолжить оценивать сценарии.

- Сценарий 1. “Отчет о контактах продавцов”, 3 балла
- Сценарий 2. “Отчет о командировочных расходах”, 7 баллов
- Сценарий 3. “Комментарии к отчетам о расходах продавцов”, 2 балла
- Сценарий 4. “Представление отчета о расходах в виде таблицы”, 4 балла
- Сценарий 5. “Уведомление бухгалтера”, 3 балла
- Сценарий 6. “Уведомление менеджера по продажам”, 3 балла
- Сценарий 7. “Уведомление продавцов”, 1 балл
- Сценарий 8. “Уведомление почтовой службы”, 2 балла
- Сценарий 9. “Аутентификация и авторизация”, 4 балла
- Сценарий 10. “Отчет о выдаче наличных”, 2 балла
- Сценарий 11. 0 баллов
- Сценарий 12. 0 баллов
- Сценарий 13. “Экспорт данных из отчетов о расходах продавцов”, 3 балла
- Всего: 34 балла

Только не стройте иллюзий, думая, что 34 балла составляют 17 идеальных дней реализации. Ключевым здесь является слово “идеальных”. Поскольку, кроме самой реализации, у вас будут запланированы встречи, некоторые дни будут более продуктивными, а некоторые — менее. Кроме всего прочего, неизбежны изменения в самом плане проекта и многое другое.

К тому же приведенные оценки являются только предварительными. После того как будет реализован тот или иной сценарий, необходимо записать точное время, которое ушло на его реализацию. При этом через несколько недель вы сможете подкорректировать сделанные ранее оценки на основе реально полученных данных.

Планирование процесса разработки

После общения с Вэнди, Вэйдом, Эдвиной и Харольдом вы понимаете, что перечисленные выше сценарии целесообразно объединить в три группы и реализовывать в следующем порядке.

Отчет о контактах продавцов.

- Сценарий 9. “Аутентификация и авторизация”, 4 балла
- Сценарий 1. “Отчет о контактах продавцов”, 3 балла
- Сценарий 6. “Уведомление менеджера по продажам”, 3 балла
- Сценарий 8. “Уведомление почтовой службы”, 2 балла

Отчет о командировочных расходах.

- Сценарий 2. “Отчет о командировочных расходах”, 7 баллов
- Сценарий 3. “Комментарии к отчетам о расходах продавцов”, 2 балла

- Сценарий 10. “Отчет о выдаче наличных”, 2 балла
- Сценарий 4. “Представление отчета о расходах в виде таблицы”, 4 балла

Дополнительные службы для работы с отчетами о командировочных расходах.

- Сценарий 5. “Уведомление бухгалтера”, 3 балла
- Сценарий 7. “Уведомление продавцов”, 1 балл
- Сценарий 13. “Экспорт данных из отчетов о расходах продавцов”, 3 балла

Такой порядок реализации сценариев обусловлен тем, что отчеты о контактах продавцов являются более критичными по времени, нежели отчеты о командировочных расходах. Кроме того, для формирования отчетов о контактах продавцов требуется меньше входной информации, и в нем предусмотрено более простое взаимодействие между сценариями. К тому же, данная отчетная форма будет использоваться в качестве исходной для тестирования работы всей системы в целом.

Каждую из групп сценариев будем называть *итерацией* (iteration). Реализация каждой итерации не должна занимать более трех недель и должна заканчиваться конкретным результатом, который сможет использовать клиент.

Таким образом, первоочередным заданием является реализация отчетов о контактах продавцов и возможность создания учетных записей и уведомления продавцов. После реализации первой итерации можно будет оценить, насколько оценки являются правильными в следующих аспектах:

- по сравнению друг с другом;
- по сравнению с исходными оценками;
- с учетом времени, потраченного на реализацию проекта за неделю.

После каждой итерации следует оценить проделанную работу, получить отзывы заказчиков и скорректировать план реализации проекта в соответствии с изменяющимися требованиями и целями заказчика.

Таким образом, в данном разделе было показано, как исходную задачу разбить на отдельные сценарии, а их, в свою очередь, разделить или объединить для получения сценарив приблизительно одинакового размера (в единицах реализации). При этом оценка трудоемкости реализации сценариев постоянно изменяется.

Очень важным аспектом является постоянный диалог с заказчиком с целью выявления наиболее важных требований к разрабатываемой системе. В свою очередь со стороны разработчика необходимо оценить сложность реализации тех или иных сценариев. Такой подход позволяет разработчикам постоянно отслеживать (и оценивать) трудоемкость реализации всего проекта, а заказчикам — отслеживать текущее состояние разрабатываемой системы и положение дел по проекту в целом.

Начало работы

Поскольку первоочередной задачей является реализация отчетов о контактах продавцов, рассмотрим следующие сценарии более подробно.

- Сценарий 9. “Аутентификация и авторизация”, 4 балла
- Сценарий 1. “Отчет о контактах продавцов”, 3 балла
- Сценарий 6. “Уведомление менеджера по продажам”, 3 балла
- Сценарий 8. “Уведомление почтовой службы”, 2 балла

Детали реализации сценария 9

“Каждый продавец должен иметь свою учетную запись”.

То есть, “продавец” должен существовать как логическая сущность (не обязательно как класс PHP). Поэтому рассмотрим атрибуты, которыми должен обладать продавец.

Очевидно, что каждая запись о пользователе должна содержать информацию об имени и фамилии.

First Name

Last Name

Кроме того, для каждого пользователя необходимо обеспечить идентификатор.

Employee ID

Для входа в систему каждый пользователь должен ввести имя учетной записи и пароль.

Login Name

Login Password

С каждым пользователем связана роль, определяющая уровень доступа к системе. В качестве ролей можно воспользоваться названиями должностей, однако они могут изменяться. Поэтому в нашем случае ограничимся следующими тремя уровнями доступа: продавец, бухгалтер, менеджер по продажам.

Company Role

Возникает несколько вопросов: можно ли для одного и того же пользователя определить несколько ролей одновременно? Например, может ли пользователь системы одновременно быть бухгалтером и менеджером по продажам? Ответ Вэнди на эти вопросы таков: “Скорее всего, нет. Не думаю, что обязанности Вэйда и Эдинны перекрываются.” Задав эти же вопросы Вэйду и Эдинне, получаем тот же ответ. Таким образом, роли не перекрываются.

Поскольку вы отвечаете за обеспечение аутентификации, необходимо определить, каким образом будет использоваться система.

Создание тестов

Задайте себе несколько вопросов.

- Каким образом будет осуществляться вход в систему?
- Для каких целей предназначена аутентификация? Будет ли использоваться простая аутентификация?
- Кому нужна аутентификация?

Для любого пользователя системы были определены следующие атрибуты.

Employee ID

First Name

Last Name

Company Role

Для аутентификации необходимо ввести параметры учетной записи.

Login Name

Login Password

Давайте вспомним класс `UserSession` из главы 15. Это как раз тот случай, когда им стоит воспользоваться. При этом дополнительно необходимо объявить два класса `WidgetSession` и `WidgetUser`.

```
<?php

$session = new WidgetSession(); // унаследовано от класса UserSession
$session->impress();
//автентификация
$session->login("ed", "1234");
if ($session->isLoggedIn() == false) exit;

$user = $session->getUser(); //возвращает объект класса WidgetUser
print $user->first_name;
print $user->last_name;
print $user->email;

//авторизация
print $user->role;
print $user->isSalesPerson();
print $user->isSalesManager();
print $user->isAccountant();

?>
```

Помните, что этот код содержит лишь “правдоподобный” сценарий. Это неформальный тест и неработающий код, а лишь ваше видение использования новых классов. Позднее его можно будет формализовать.

PHPUnit

В этом разделе мы разработаем тест, имитирующий способ использования программного модуля для данного сценария.

Такой подход к разработке называется *разработкой на основе тестирования* (test-driven development), поэтому по завершении создания приложения вы также получите удобный набор тестов.

Разработка тестов в процессе создания приложения позволяет внимательно взглянуть на функции объектов и их параметры, а также избежать лишней функциональности, которая не будет использоваться.

Наличие подобных тестов обеспечивает дополнительную свободу в процессе разработки приложения, поскольку функциональность каждого модуля можно модифицировать, а затем сразу же протестировать его прежние функции. Никакие нововведения не должны нарушить уже созданную функциональность.

Приведем еще несколько тестов для проверки самого типичного сценария. Их нужно сохранить в файле `test.widgetsession.php`.

```
<?php

require_once ("widgetsession.phpm");
require_once ("lib/phpunit/phpunit.php");

class TestWidgetSession extends TestCase
{
    private $_session;

    function setUp() {
        $dsn = array ('phptype' => "pgsql",
                     'hostspec' => "localhost",
                     'database' => "widgetworld",
```

```

        'username' => "wuser",
        'password' => "foobar");
    ($this->_session = new WidgetSession($dsn, true);
}

function testValidLogin() {
    $this->_session->login("ed", "12345");
    $this->assertEquals(true, $this->_session->isLoggedIn());
}

function testInvalidLogin() {
    $this->_session->login("ed", "54321"); // fail
    $this->assertEquals(false, $this->_session->isLoggedIn());
}

function testUser() {
    $user = $this->_session->getUser();
    $this->assertEquals("Lecky-Thompson", $user->last_name);
    $this->assertEquals("Ed", $user->first_name);
    $this->assertEquals("ed@lecky-thompson.com", $user->email);
}

function testAuthorization() {
    $user = $this->_session->getUser();
    $this->assertEquals("Sales Person", $user->role);
    $this->assertEquals(true, $user->isSalesPerson());
    $this->assertEquals(false, $user->isSalesManager());
    $this->assertEquals(false, $user->isAccountant());
}
}

$suite = new TestSuite;
$suite->addTest(new TestWidgetSession("testValidLogin"));
$suite->addTest(new TestWidgetSession("testInvalidLogin"));
$suite->addTest(new TestWidgetSession("testUser"));
$suite->addTest(new TestWidgetSession("testAuthorization"));

$testRunner = new TestRunner();
$testRunner->run( $suite );

?>
```

В результате запуска этого кода будет выведено следующее сообщение об ошибке.

```
Class 'WidgetSession' not found in test.widgetsession.php on line 16
```

Чтобы обеспечить работоспособность теста, создадим тестовые классы `WidgetSession` и `WidgetUser`. Для этого в начале предыдущего листинга добавим следующий код.

```

class WidgetSession {
    public function __construct ($one, $two) {}
    public function login() {}
    public function isLoggedIn() { return null; }
    public function getUser() {
        return new WidgetUser();
    }
}

class WidgetUser {
    public $first_name = "";
    public $last_name = "";
    public $email = "";
```

```

    public function isSalesPerson() { return null; }
    public function isSalesManager() { return null; }
    public function isAccountant() { return null; }
}

```

Теперь снова запустим тест `test.widgetsession.php` и получим следующий результат, сгенерированный средствами PHPUnit.

```

TestWidgetSession - testValidLogin FAIL
TestWidgetSession - testInvalidLogin FAIL
TestWidgetSession - testUser FAIL
TestWidgetSession - testAuthorization FAIL

```

```

4 tests run.
9 failures.
0 errors.
Failures

```

```

1. testValidLogin
    true   type:boolean
    null   type:NULL
2. testInvalidLogin
    false  type:boolean
    null   type:NULL
3. testUser
    Lecky-Thompson type:string
    type:string
4. testUser
    Ed     type:string
    type:string
5. testUser
    ed@lecky-thompson.com type:string
    type:string
6. testAuthorization
    Sales Person type:string
    type:string
7. testAuthorization
    true   type:boolean
    null   type:NULL
8. testAuthorization
    false  type:boolean
    null   type:NULL
9. testAuthorization
    false  type:boolean
    null   type:NULL

```

Теперь ни один из тестов не содержит синтаксических ошибок, но некоторые из них не могут быть пройдены. Это связано с отсутствием тестируемой функциональности. Поэтому при первом запуске теста не стоит удивляться его невыполнению.

Теперь необходимо реализовать классы и провести реальные тесты. Процесс регистрации и обработки сеансов реализуется в классе `UserSession`.

Не забудьте о классе `WidgetSession`. Поскольку он расширяет функциональность класса `UserSession`, его целесообразно унаследовать от этого класса.

```

class WidgetSession extends UserSession {
    public function getUser() {
        return new WidgetUser();
    }
}

```

Снова запустим тест `test.widgetsession.php`.

```

TestWidgetSession - testValidLogin ok
TestWidgetSession - testInvalidLogin ok

```

```
TestWidgetSession - testUser FAIL
TestWidgetSession - testAuthorization FAIL

4 tests run.
7 failures.
0 errors.
```

Поскольку функциональность класса UserSession реализована, половина тестов пройдена успешно.

Теперь приступим к работе с объектами WidgetUser. Для этого сначала перекроем функцию getUserObject() класса UserSession, чтобы она возвращала объект класса WidgetUser.

```
class WidgetSession extends UserSession {

    public function getUserObject() {
        $uid = $this->GetUserID(); // вызов из объекта UserSession
        if ($uid == false) return null;

        // извлечение информации из базы данных
        $stmt = "select * FROM \"user\" WHERE id = ".$uid;
        $result = $this->getDatabaseHandle()->query($stmt);
        return new WidgetUser($result->fetchRow());
    }
}
```

Необходимо каким-то образом сохранять состояние объекта WidgetUser. Данные объекта WidgetUser заполняются из базы данных, поэтому состояние этого объекта целесообразно тоже хранить в ассоциативном массиве.

При этом необходимо переопределить функции __set() и __get().

```
class WidgetUser {

    protected $contentBase = array();

    function __construct($initdict) {
        $this->contentBase = $initdict; // копирование
    }

    function __get ($key) {
        if (array_key_exists ($key, $this->contentBase)) {
            return $this->contentBase[$key];
        }
        return null;
    }

    function __set ($key, $value) {
        if (array_key_exists ($key, $this->contentBase)) {
            $this->contentBase[$key]=$value;
        }
    }

    public function isSalesPerson() { return null; }
    public function isSalesManager() { return null; }
    public function isAccountant() { return null; }
}
```

При следующем запуске теста test.widgetsession.php большинство тестов будет пройдено (за исключением одного).

1. testuser
edolecky-thompson.com type:string
null type:NULL

Рассмотрим версию таблицы user для базы данных MySQL.

```
CREATE TABLE "user" (
    id serial PRIMARY KEY,
    username varchar(32) default NULL,
    md5_pw varchar(32) default NULL,
    first_name varchar(64) default NULL,
    last_name varchar(64) default NULL
);
```

Отсюда видно, что в таблице user отсутствует поле адреса электронной почты, которое необходимо добавить.

```
CREATE TABLE "user" (
    id serial PRIMARY KEY,
    username varchar(32) default NULL,
    md5_pw varchar(32) default NULL,
    first_name varchar(64) default NULL,
    last_name varchar(64) default NULL,
    email varchar(255) default NULL
);
```

Снова запустим тест test.widgetsession.php.

```
TestWidgetSession - testValidLogin ok
TestWidgetSession - testInvalidLogin ok TestWidgetSession - testUser ok
TestWidgetSession - testAuthorization FAIL
```

```
4 tests run.
4 failures.
```

Теперь при тестировании не будет выявлено никаких ошибок, поэтому обратимся к следующему тесту testAuthorization. Поскольку в базе данных еще не введено поле роли, добавим его и присвоим ему по умолчанию значение s (продавец).

```
CREATE TABLE "user" {
    id serial PRIMARY KEY,
    username varchar(32) default NULL,
    md5_pw varchar(32) default NULL,
    first_name varchar(64) default NULL,
    last_name varchar(64) default NULL,
    email varchar(255) default NULL,
    role char(1) NOT NULL default 's'
};
```

Добавим несколько методов получения состояния в класс WidgetUser.

```
public function isSalesPerson() {
    if ($this->role == "s") return true;
    return false;
}

public function isSalesManager() {
    if ($this->role == "m") return true;
    return false;
}

public function isAccountant() {
    if ($this->role == "a") return true;
    return false;
}
```

Однако тест проверки роли пользователя завершается ошибкой, поскольку ожидается вывод осмысленного значения, а не просто символа s, m или a.

1. testAuthorization


```
Sales Person type:string
s type:string
```

Вспомним, что мы тестируем.

```
function testAuthorization () {
    $user = $this->_session->getUser();

    $this->assertEquals("Sales Person", $user->role);

    $this->assertEquals(true, $user->isSalesPerson());
    $this->assertEquals(false, $user->isSalesManager());
    $this->assertEquals(false, $user->isAccountant());
}
```

Отсюда видно, что оператор `$user->role` должен возвращать значение `Sales Person`. Значение роли выбирается из базы данных, поэтому придется изменить либо программный код, либо представление роли в базе данных.

Воспользуемся следующей идеей. С помощью функции `__get()` по-прежнему будем извлекать значение поля `role` из базы данных, а затем будем его преобразовывать к нужному виду в функции `getRole()`.

Приведем код класса `WidgetUser`.

```
class WidgetUser {

    protected $contentBase = array();
    protected $dispatchFunctions = array ("role" => "getrole");

    function __construct($initdict) {
        $this->contentBase = $initdict; // копирование
    }

    function __get ($key) {

        // вызов функции
        if (array_key_exists ($key, $this->dispatchFunctions)) {
            $funcname = $this->dispatchFunctions[$key];
            return $this->$funcname();
        }

        // в противном случае возвращаем состояние
        if (array_key_exists ($key, $this->contentBase)) {
            return $this->contentBase[$key];
        }
        return null;
    }

    function __set ($key, $value) {
        if (array_key_exists ($key, $this->contentBase)) {
            $this->contentBase[$key]=$value;
        }
    }

    public function getRole() {
        switch ($this->contentBase["role"]) {
            case "s": return ("Sales Person");
            case "m": return ("Sales Manager");
            case "a": return ("Accountant");
            default: return ("");
        }
    }

    public function isSalesPerson() {
        if ($this->contentBase["role"] == "s") return true;
        return false;
    }
}
```

```

public function isSalesManager() {
    if ($this->contentBase["role"] == "m") return true;
    return false;
}

public function isAccountant() {
    if ($this->contentBase["role"] == "a") return true;
    return false;
}
}

```

Обратите внимание, что функции `isSalesPerson()`, `isSalesManager()` и `isAccountant()` обращаются к массиву `contentBase`, а не к полю `$this->role`. В противном случае им бы пришлось иметь дело с длинными строками типа `Sales Manager`. А для таких строк выполнять операцию сравнения гораздо менее удобно.

Создание страницы регистрации

Теперь созданные и протестированные классы `WidgetUser` и `WidgetSession` можно использовать в сценарии регистрации.

Хотя сценарий регистрации не является слишком сложным, для удобства его создания воспользуемся шаблоном Smarty.

Приведем исходный файл `index.php`.

```

<?php

require_once ("Smarty.class.php");
require_once ("widgetsession.phpm");

$session = new WidgetSession (array (
    'phptype' => "pgsql",
    'hostspec' => "localhost",
    'database' => "widgetworld",
    'username' => "wuser",
    'password' => "foobar"));
$session->Impress();

$smarty = new Smarty;

if ($_REQUEST["action"] == "login") {
    $session->login($_REQUEST["login_name"], $_REQUEST["login_pass"]);
    if ($session->isLoggedIn()) {
        $smarty->assign_by_ref ("user", $session->getUserObject());
        $smarty->display ("main.tpl");
        exit;
    } else {
        $smarty->assign('error', "Неверные данные, попробуйте еще раз.");
        $smarty->display ("login.tpl");
        exit;
    }
} else {
    if ($session->isLoggedIn() == true) {
        $smarty->assign_by_ref ("user", $session->getUserObject());
        $smarty->display ("main.tpl");
        exit;
    }
}

$smarty->display ("login.tpl");

?>

```

Это сценарий работает следующим образом. Если переменная `action` принимает значение `login`, предпринимается попытка регистрации с помощью вызова функции `$session->login`. В случае успешной регистрации отображается главное меню, а в случае неудачной — генерируется ошибка, и пользователь снова возвращается на страницу регистрации.

Если попытка регистрации не предпринималась, то статус данного сеанса проверяется с помощью метода `$session->isLoggedIn()`. Если пользователь зарегистрирован, отображается главное меню. В противном случае пользователь остается на странице регистрации.

Шаблоны `login`, `main`, а также шаблоны верхнего и нижнего колонтитула довольно просты.

Шаблон `login.tpl` имеет следующий вид.

```
{include file= "header.tpl" title="Регистрация в приложении Widget World"}

<h3>Введите данные:</h3>
<p>
{section name=one loop=$error}{sectionelse}
  <font color="#FF0000">{$error}</font><p>
{/section}

<form action="index.php" method="post">
<table border="0">
<tr><td width="20"></td><td>Пользователь:</td>
  <td><input name="login_name" type="text" size="20" maxsize="50"></td></tr>
<tr><td width="20"></td><td>Пароль:</td>
  <td><input name="login_pass" type="password" size="20" maxsize="50"></td></tr>
<tr><td width="20"></td><td></td>
  <td><input type="submit" value=" Зарегистрироваться "></td></tr> </table>
<input type="hidden" name="action" value="login">
</form>

{include file="footer.tpl"}
```

Шаблон `header.tpl` имеет следующий вид.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta HTTP-EQUIV="content-type" CONTENT="text/html; charset=ISO-8859-1">
<title>{$title|default:"no title"}</title>
</head>
<h1>Widget World</h1>
<hr><p>
```

Шаблон `footer.tpl` имеет следующий вид.

```
<br/><br/>
<hr/>
Только для приложения Widget World.
</body>
</html>
```

Шаблон `main.tpl` имеет следующий вид.

```
{include file="header.tpl" title="Меню приложения Widget World"}

Добро пожаловать {$user->first_name} в роли {$user->role}!

<table border="0" cellspacing="8" cellpadding="8">
```

```

{strip}
{section name=security show=$user->isAccountant()}
    <tr><td><h3>Здесь реализуется функциональность для бухгалтера.</h3></td></tr>
{/section}

    <tr><td valign="top"><h3><a href="travel-expenses.php">Новый отчет
о командировочных расходах</a></h3></td></tr>
    <tr><td valign="top"><h3><a href="customer-contacts.php">Новый отчет
о контактах</a></h3></td></tr>

{/strip}
</table>

{include file="footer.tpl"}

```

Результирующая страница показана на рис. 22.3.

The screenshot shows a login form titled "Widget World". The title is at the top center. Below it is a horizontal line. The form has a label "Ведите данные:" followed by two input fields: "Пользователь:" containing "ed" and "Пароль:" containing "*****". Below the inputs is a button labeled "Зарегистрироваться". At the bottom of the form is another horizontal line, followed by the text "Только для приложения Widget World.".

Рис. 22.3.

После успешной регистрации отображается страница с главным меню, показанная на рис. 22.4.

Если роль пользователя равна а (бухгалтер (Accountant)), то для него открывается несколько иная страница (рис. 22.5).

Поздравляем! Первый сценарий реализован. Для его создания пришлось воспользоваться навыками работы с сеансами PHP, базами данных, пакетами PEAR и PHPUnit.

Теперь можно немного отдохнуть и перейти к следующему сценарию.

Следующий сценарий

Предыдущий сценарий аутентификации и авторизации был оценен в 4 балла. Запишите, сколько дней понадобилось на решение этой задачи.

Напомним выбранную последовательность реализации сценариев.

- ❑ Сценарий 9. “Аутентификация и авторизация” — 4 балла (реализован).
- ❑ Сценарий 1. “Отчет о контактах с покупателями” — 3 балла.

Widget World

Добро пожаловать Эд в роли Sales Person!

Новый отчет о командировочных расходах

Новый отчет о контактах

Только для приложения Widget World.

Рис. 22.4.

Widget World

Добро пожаловать Эд в роли Accountant!

Здесь реализуется функциональность для бухгалтера.

Новый отчет о командировочных расходах

Новый отчет о контактах

Только для приложения Widget World.

Рис. 22.5.

- Сценарий 6. “Уведомление менеджера” — 3 балла.
 - Сценарий 8. “Уведомление службы рассылки (канцелярии)” — 2 балла.
- Теперь вернемся к работе.

Требования к отчету о контактах

Напомним (см. рис. 22.1), какие данные заносятся в еженедельный отчет о контактах. Их можно описать с помощью следующей таблицы.

```
CREATE TABLE contact_visits (
    emp_id integer NOT NULL,
    week_start date NOT NULL,
```

```

seq integer NOT NULL,
company_name varchar(40) default NULL,
contact_name varchar(40) default NULL,
city varchar(40) default NULL,
state varchar(40) default NULL,
accomplishments text,
followup text,
literature_request text
);
CREATE UNIQUE INDEX cv_pk ON contact_visits (emp_id,week_start,seq);
CREATE INDEX cv_emp_id ON contact_visits (emp_id);
CREATE INDEX cv_week_start ON contact_visits (week_start);
CREATE INDEX cv_seq ON contact_visits (seq);

```

Эти данные обновляются еженедельно и связаны с конкретным сотрудником фирмы. Таких данных может быть очень много.

В отчете о контактах указывается отдел, в котором работает сотрудник. Данные об этом отделе еще не внесены в таблицы user. Добавим их.

```

CREATE TABLE "user" (
id serial PRIMARY KEY,
username varchar(32) default NULL,
md5_pw varchar(32) default NULL,
first_name varchar(64) default NULL,
last_name varchar(64) default NULL,
email varchar(255) default NULL,
role char(1) NOT NULL default 's',
department varchar(40) NOT NULL default ''
);

```

Тесты для модуля отчета о контактах

Давайте подумаем, как будут использоваться данные о контактах с покупателями. Эти данные будут передаваться через Web-сервер, поэтому для их хранения необходимо обеспечить доступ к идентификатору сотрудника для класса WidgetSession.

Рассмотрим, какая минимальная информация может понадобиться для теста PHPUnit. Это данные emp_id, week_start и seq.

```

function testValidContactVisit() {
    $cv = new ContactVisit (
        array ('emp_id'          => "1",
               'seq'              => "1",
               'week_start'       => "1980-01-01",
               'company_name'     => "test one",
               'contact_name'     => "Big One",
               'city'             => "Columbus",
               'state'            => "OH",
               'accomplishments' => "phone call",
               'followup'         => "",
               'literature_request' => ""));
    $this->assertEquals(true, $cv->isValid(), "valid log");
}

```

Этот тест проверяет наличие минимальной информации, необходимой для класса ContactVisit. Поэтому если поля emp_id, week_start и seq содержат любые значения, то функция isValid() возвращает значение true. Необходимо проверить и противоположную ситуацию.

```

function testInvalidContactVisit() {
    $cv = new ContactVisit (

```

```

array ('emp_id'          => "1",
      'week_start'     => "", // требуется дата
      'company_name'   => "test one",
      'contact_name'   => "Big One",
      'city'           => "Columbus",
      'state'          => "OH",
      'accomplishments' => "phone call",
      'followup'        => "",
      'literature_request' => ""));
$this->assertEquals(false, $cv->isValid(), "invalid visit");
}

```

Поскольку с каждым визитом связан уникальный номер seq, то порядок следования можно определять автоматически.

```

function testSequence() {
    $cv1 = new ContactVisit(array());
    $this->assertEquals(1, $cv1->seq);
    $cv2 = new ContactVisit(array());
    $this->assertEquals(2, $cv2->seq);
}

```

Номер визита (значение переменной seq) автоматически инкрементируется для каждого нового объекта ContactVisit, поэтому нет необходимости создавать специальный контейнер для хранения номера визита.

Проверка базы данных является тривиальной, но ее тоже нужно реализовать.

```

function testPersistence() {
    $this->_session->getDatabaseHandle()->query(
        "delete FROM contact_visits WHERE
         emp_id = 1 and week_start = '1980-01-01'");
    // удаление повторов
    $cv = new ContactVisit (
        array ('emp_id'          => "1",
              'week_start'     => "1980-01-01",
              'seq'            => 1,
              'company_name'   => "test one",
              'contact_name'   => "Big One",
              'city'           => "Columbus",
              'state'          => "OH",
              'accomplishments' => "phone call",
              'followup'        => "",
              'literature_request' => ""));
    $result = $this->_session->getDatabaseHandle()->
        query("select * FROM contact_visits WHERE
         emp_id = 1 and week_start = '1980-01-01'");
    $this->assertEquals(0, $result->numRows());
    $cv->persist();
    $result = $this->_session->getDatabaseHandle()->
        query("select * FROM contact_visits WHERE
         emp_id = 1 and week_start = '1980-01-01'");
    $this->assertEquals(1, $result->numRows());
}

```

Перед запуском теста не забудьте создать класс ContactVisit (пока что пустой) и сохраните его в файле contact.phpm.

```

class ContactVisit {

    function __construct ($results) { }

    public function isValid() { return null; }
}

```

```
public function persist() { }
public function getSequence() { return null; }
}
```

Процесс тестирования

Поскольку визиты совершаются последовательно, создайте статическую переменную, чтобы последовательная нумерация визитов поддерживалась для всех экземпляров данного класса. Реализуйте конструктор класса ContactVisit в файле contact.phpm.

```
class ContactVisit {
```

```
function __construct ($results, $dbh = null) {
    static $sequence = 0;
    $this->dbh = $dbh;
    $this->contentBase = $results; // копирование
    $sequence = $sequence +1; // инкрементирование
    $this->contentBase["seq"] = $sequence;
}

public function isValid() { return null; }
public function persist() { }
public function getSequence() { return null; }
```

```
}
```

В классе ContactVisit необходимо хранить не только обычные данные (имя компании, город и т.д.), но и ссылку на базу данных. Поэтому функция __get() класса ContactVisit должна возвращать состояние объекта данного класса. Добавим ее в файл contact.phpm.

```
function __get($key) {
    if (array_key_exists ($key, $this->contentBase)) {
        return $this->contentBase[$key];
    }
    return $this->$key;
}
```

Теперь добавим некоторые функции генерации SQL-запросов. Файл contact.phpm примет следующий вид.

```
class ContactVisit {

    protected $contentBase = array();
    protected $dbh = null; // дескриптор базы данных

    function __get ($key) {
        if (array_key_exists ($key, $this->contentBase)) {
            return $this->contentBase[$key];
        }
        return $this->$key;
    }

    function __construct ($results, $dbh = null) {
        static $sequence = 0;
        $this->dbh = $dbh;
```

```

$this->contentBase = $results; // копирование
$sequence = $sequence +1; // инкрементирование
$this->contentBase["seq"] = $sequence;
}

private function isEmpty($key) {
    if (array_key_exists($key, $this->contentBase) == false) return true;
    if ($this->contentBase[$key] == null) return true;
    if ($this->contentBase[$key] == "") return true;
    return false;
}

public function isValid() {
    if ($this->isEmpty("emp_id") == true) return false;
    if ($this->isEmpty("week_start") == true) return false;
    if ($this->isEmpty("company_name") == true) return false;
    return true;
}

private function implodeQuoted(&$values, $delimiter) {
    $sql = "";
    $flagIsFirst = true;
    foreach ($values as $value) {
        if ($flagIsFirst) {
            $flagIsFirst = false;
        } else {
            $sql .= $delimiter;
        }

        if (gettype ($value) == "string") {
            $sql .= "'".$value."'";
        } else {
            $sql .= $value;
        }
    }
    return $sql;
}

private function generateSqlInsert ($tableName, &$metas, &$values) {
    return "insert into ".$tableName.
        " (" . implode           ($metas,
", ") . " ) ".
        " values (" . $this->implodeQuoted   ($values,
", ") . " ) ";
}

public function persist() {
    if ($this->isValid() == false) return false;
    $sql = $this->generateSqlInsert ("contact_visits",
array("emp_id",
"week_start",
"seq",
"company_name",
"contact_name",
"city",
"state",
"accomplishments",
"followup",
"literature_request"),
array ( $this->emp_id,
$this->week_start,
$this->seq,
$this->company_name,

```

```

        $this->contact_name,
        $this->city,
        $this->state,
        $this->accomplishments,
        $this->followup,
        $this->literature_request );
    if (DB::isError ($this->dbh->query($sql))) return false;
    return true;
}
}

```

Таким образом, реализованы следующие методы.

- Метод `isValid()` проверяет корректность данных. Он основывается на вызове функции `isEmpty()`, которая добавлена для удобства.
- Метод `persist()`, обеспечивающий генерацию SQL-запросов. В нем вызывается метод `generateSqlInsert()`, генерирующий SQL-запрос `INSERT` на основе имени таблицы, метаинформации и добавляемых значений. В свою очередь метод `generateSqlInsert()` вызывает метод `implodeQuoted()`, который добавляет кавычки к строковым значениям.

В методах `generateSqlInsert()` и `implodeQuoted()` предполагается, что массивы передаются по ссылке, а не по значению. По умолчанию в PHP 5 объекты передаются по ссылке, но массивы — по значению.

Создание окна

Рассмотрим форму, содержащуюся в файле `customer-contacts.tpl`.

```

{include file="header.tpl" title="Widget World - Контакты"}

<h3>Отчет о контактах</h3>

<form action="customer-contacts.php" method="post">
<table border="0" width="100%">
<tr><td><b>Имя сотрудника:</b></td><td>{$user->first_name}<br>{$user->last_name}</td>
<td><b>Отдел:</b></td><td>{$user->department }</td></tr>
<tr><td><b>Номер:</b></td><td>{$user->id}</td><td><b>Начальная неделя:</b></td>
<td><SELECT NAME="week_start">{$html_options values=$start_weeks output=$start_weeks selected=$current_start_week}</SELECT></td></tr>
</table>

<br><br><hr>

<p><font size="+1"><b>Важные дистрибуторы и покупатели:</b></font><br>
( также перспективы )<p>
<table border="0">
{section name=idx loop=$max_weekly_contacts}{strip} <tr><td
width="20"></td><td><b>Компания:</b></td><td><b>Контакт:</b></td><td><b>Город:</b>
</td><td><b>Страна:</b></td><td><b>Ответ:</b></td><td><b>Запрос на
документацию:</b></td></tr>
<tr>
<td width="20"></td>
<td><input name="company_name_{$smarty.section.idx.index}" size="20"
maxlength="50"></td>
<td><input name="contact_name_{$smarty.section.idx.index}" size="20"
maxlength="50"></td>
<td><input name="city_{$smarty.section.idx.index}" size="20"
maxlength="50"></td>
<td><input name="state_{$smarty.section.idx.index}" size="10"
maxlength="50"></td>

```

```
maxlength="50">></td>
<td><input name="followup_{$smarty.section.idx.index}" size="20" maxlength="2000"></td>
<td><input name="literature_request_{$smarty.section.idx.index}" size="20" maxlength="2000"></td>
</tr>
<tr>
<td width="20">></td>
<td colspan=" 7">><b>Результаты:</b></td>
</tr>
<tr>
<td width="20">></td>
<td colspan="7">><TEXTAREA NAME="accomplishments_{$smarty.section.idx.index}" ROWS=4 COLS=95></TEXTAREA><br><br>
</td>
</tr>

{/strip}{/section}

</table>

<br><hr>
<input type="hidden" name="action" value="persist_contact">
<br><br>

<center>
<input type="submit" name="submit" value=" Сохранить " onclick="return checkInputs(this.form); ">
</center>

</form>

{include file="footer.tpl"}
```

В главном цикле {section name=idx loop=\$max_weekly_contacts} создаются уникальные имена для входных значений company_name_{\${smarty.section.idx.index}}, над которыми затем выполняются действия.

Smarty обеспечивает удобный способ заполнения раскрывающихся списков на основе массивов дней текущей недели.

```
<SELECT NAME="week_start">{html_options values=$start_weeks output=$start_weeks selected=$current_start_week}</SELECT>
```

Эта страница выглядит несколько спартанской, поскольку создана на основе обычного кода HTML, а не более сложных средств типа XHTML или CSS. На данном этапе нас больше интересует работоспособность системы, а внешний лоск можно обеспечить позднее.

Добавление функциональности

Хотя класс `ContactVisit` умеет записывать информацию в базу данных, для получения сведений за несколько недель потребуется немного попотеть. Возложим эти обязанности на функции файла `customer-contacts.php`.

```

'database' => "widgetworld",
'username' => "uwuser",
'password' => "foobar"));

$session->Impress();

$smarty = new Smarty;

$GLOBALS["max-weekly-contacts"] = 5;

function getStartDateOffset ($i) {
    if ($i < 0) $i = 5;
    $dates = array("Воскресенье" => 0, "Понедельник" => -1,
                  "Вторник" => -2, "Среда" => -3,
                  "Четверг" => -4, "Пятница" => -5,
                  "Суббота" => -6);
    return date("Y-m-d", mktime (0,0,0,date("m"),
                                 date("d")+$dates[date("1")]-
                                 (($i-5)*7),date("Y")));
}

function getCurrentStartWeek () {
    if (strlen($_REQUEST["week_start"])) >= 8) return $_REQUEST["week_start"];
    return getStartDateOffset(-1); // Это воскресенье
}

function getStartWeeks () {
    $sundayArray = array();
    for ($i=20; $i > 0; $i--) {
        array_push($sundayArray, getStartDateOffset($i));
    }
    return ($sundayArray);
}

function persistContactvisits (&$dbh, $emp_id) {
    $dbh->query("delete from contact_visits where emp_id = ".$emp_id."
                  and week_start = '".getCurrentStartWeek()."')");
    $seq = 0;
    for ($i = 0; $i < $GLOBALS["max-weekly-contacts"]; $i++) {
        $cv = new ContactVisit (
            array ("emp_id" => $emp_id,
                   "week_start" => getCurrentStartWeek(),
                   "company_name" => $_REQUEST["company_name_".$i],
                   "contact_name" => $_REQUEST["contact_name_".$i],
                   "city" => $_REQUEST["city_".$i],
                   "state" => $_REQUEST["state_".$i],
                   "accomplishments" => $_REQUEST["accomplishments_".$i],
                   "followup" => $_REQUEST["followup_".$i],
                   "literature_request" => $_REQUEST["literature_request_".$i]), $dbh);
        $cv->persist();
    }
}

$user = $session->getUserObject();

// отображение
if ($_REQUEST["action"] != "persist_contact") {
    $smarty->assign_by_ref ("user", $user);
    $smarty->assign('start_weeks', getStartWeeks());
    $smarty->assign('current_start_week', getCurrentStartWeek());
    $smarty->assign("max_weekly_contacts", $GLOBALS["max-weekly-contacts"]);
    $smarty->display('customer-contacts.tpl');
    exit();
}

```

```
// сохранение данных о контактах
require_once ("contact.phpm");
persistContactVisits ($session->getDatabaseHandle(), $user->id);

$smarty->display('thankyou.tpl');
?>
```

Самой интересной здесь является функция `getStartDateOffset()`, которая заполняет раскрывающийся список дат.

```
function getStartDateOffset ($i) {
    if ($i < 0) $i = 5;
    $dates = array("Воскресенье" => 0, "Понедельник" => -1,
                  "Вторник" => -2, "Среда" => -3,
                  "Четверг" => -4, "Пятница" => -5,
                  "Суббота" => -6);
    return date("Y-m-d", mktime (0,0,0,date("m"),
                                 date("d")+$dates[date("1")]-
                                 (($i-5)*7),date("Y")));
}
```

В ней день недели связывается с массивом `$dates`, в котором каждому дню недели ставится в соответствие целое число.

Функция `persistContactVisits()` удаляет устаревшие контакты, создает новые объекты `ContactVisit` и сохраняет их в базе данных с помощью функции `persist()`.

Результаты работы этих функций показаны на рис. 22.6.

Widget World

Отчет о контактах

Имя сотрудника:	Ed Lecky-Thompson	Отдел:	продажи
Номер:	1	Начальная неделя:	<input style="border: 1px solid black; padding: 2px; width: 150px; height: 20px;" type="button" value="2004-06-13"/> <div style="border: 1px solid black; padding: 5px; width: 150px; height: 150px; overflow: auto; vertical-align: top;"> 2004-02-29 2004-03-07 2004-03-14 2004-03-21 2004-03-28 2004-04-04 2004-04-11 2004-04-18 2004-04-25 2004-05-02 2004-05-09 2004-05-16 2004-05-23 2004-05-30 2004-06-06 2004-06-13 2004-06-20 2004-06-27 2004-07-04 2004-07-11 </div>

Важные дистрибуторы и покупатели:
(а также перспективы)

Компания	Контакт	Город	Страна	Ответ	Запрос на документацию

Результаты:

Компания	Контакт	Город	Страна	Ответ	Запрос на документацию

Результаты:

Рис. 22.6.

Теперь после реализации двух сценариев можно себе позволить немного отдохнуть.

Повторная оценка проекта

Заказчикам приложение понравилось. Однако они тоже люди, поэтому сделали некоторые замечания.

- Добавить кнопку выхода из системы.
- Запретить доступ к сценарию `customer-contacts.php` без предварительной регистрации.
- При изменении недели должны сохраняться данные за предыдущую неделю.
- В отчете о контактах необходимо отслеживать такие данные: количество звонков, количество звонков по техническим вопросам, километраж и обслуженная территория.
- Число контактов может случайно измениться. Можно ли его настраивать?

Выполнив несложные подсчеты, можно получить следующие оценки.

- Добавить кнопку выхода из системы (полбалла).
- Запретить доступ к сценарию `customer-contacts.php` без предварительной регистрации (ошибка, полбалла).
- При изменении недели должны сохраняться данные за предыдущую неделю (1 балл).
- В отчете о контактах необходимо отслеживать такие данные: количество звонков в магазин, количество звонков по техническим вопросам, число звонков дистрибуторам, примерный километраж, “ожиженная” территория и комментарии относительно территории (1 балл).
- Число контактов может случайно измениться. Можно ли его настраивать (полбалла)?

Вспомним исходный план.

- Сценарий 9. “Аутентификация и авторизация” — 4 балла (реализован).
- Сценарий 1. “Отчет о контактах с покупателями” — 3 балла (реализован).
- Сценарий 6. “Уведомление менеджера” — 3 балла.
- Сценарий 8. “Уведомление службы рассылки (канцелярии)” — 2 балла.

В процессе обсуждения с заказчиком можно несколько изменить порядок реализации сценариев.

- Сценарий 9. “Аутентификация и авторизация” — 4 балла (реализован).
- Сценарий 1. “Отчет о контактах с покупателями” — 3 балла (реализован).
- Сценарий 14. “Сохранение результатов работы при изменении недели” (новый).
- Сценарий 15. “Еженедельное добавление данных в отчет о контактах с покупателями” (новый).

Затем можно реализовать оставшиеся сценарии.

- Сценарий 6. “Уведомление менеджера” — 3 балла.
- Сценарий 8. “Уведомление службы рассылки (канцелярии)” — 2 балла.

Теперь вернемся к работе.

Чистка кода

Прежде чем переходить к реализации новых сценариев, внимательно рассмотрим существующий код.

Он полностью функционален, хотя и не совсем совершенен. Некоторые фрагменты кода скопированы и вставлены в другие места.

Не бойтесь вносить изменения в полностью оттестированный продукт. В реальной жизни мы иногда спешим, поэтому впоследствии все нужно расставить на свои места.

Уделите немного времени и почистите код.

О чём идет речь? Взгляните на следующие рабочие файлы.

```
index.php
customer-contacts.php
contact.phpm
test.contact.php
widgetsession.phpm
test.widgetsession.php
templates/customer-contacts.tpl
templates/footer.tpl
templates/header.tpl
templates/login.tpl
templates/main.tpl
templates/thankyou.tpl
```

Задайте себе вопрос: “Все ли эти файлы должны размещаться в корневом каталоге?”.

Обратите внимание на PHP-модули и тесты. Разместите их в следующей иерархии.

```
index.php
customer-contacts.php
lib/contact.phpm
lib/test.contact.php
lib/widgetsession.phpm
lib/test.widgetsession.php
templates/customer-contacts.tpl
templates/footer.tpl
templates/header.tpl
templates/login.tpl
templates/main.tpl
templates/thankyou.tpl
```

Теперь в корневом каталоге содержатся только два файла, запускаемые Web-браузером. Файлы с тестами можно переместить в отдельный каталог.

Затем проверьте все директивы включения и удостоверьтесь в том, что все тесты по-прежнему работают.

Рефакторинг кода

При разработке кода следует придерживаться некоторых важных принципов.

- Код должен быть простым.
- Не создавайте ненужные классы.
- Избегайте дублирования.
- Удаляйте лишнюю функциональность, включая классы и функции.

Последний пункт реализовать особенно сложно, поскольку трудно расставаться со сделанной работой. Однако удаление мертвых деревьев только способствует оздоровлению леса.

Надеемся, что вы поместили исходный код в соответствующее хранилище. Используйте для этого любую систему контроля версий, например CVS или Microsoft SourceSafe.

Не забывайте об этих рекомендациях при реализации простых пожеланий пользователей, в частности при добавлении кнопки выхода из системы, поскольку файл footer.tpl используется и в других местах.

```
<br><br>
<a href="index.php?action=logout">Выход из системы</a><br>
<hr>
Только для приложения Widget World.
</body>
</html>
```

Теперь обратите внимание на одинаковые фрагменты в файлах customer-contacts.php и index.php:

```
require_once ("Smarty.class.php");
require_once ("widgetsession.phpm");

$session = new WidgetSession(array ('phptype' => "pgsql",
                                    'hostspec' => "localhost",
                                    'database' => "widgetworld",
                                    'username' => "wuser",
                                    'password' => "foobar"));
$session->Impress();
$smarty = new Smarty;
```

Такое положение дел противоречит принципу недублирования, поэтому удалите данный фрагмент и поместите его в отдельный файл, например lib/common.php.

Настройки следует поместить в новый файл ../../../../../../settings.php, достаточно удаленный от корневого каталога Web-сервера.

```
<?php

/*
 * параметры базы данных
 */
$GLOBALS ["db-type"]      = "pgsql";
$GLOBALS ["db-hostname"]   = "localhost";
$GLOBALS ["db-username"]   = "wuser";
$GLOBALS ["db-password"]   = "foobar";
$GLOBALS ["db-name"]       = "widgetworld";

/*
 * Параметры окружения
 */
$GLOBALS ["smarty-path"]   = "/usr/lib/php/smarty/";

/*
 * системные настройки
 */
$GLOBALS ["max-weekly-contacts"] = 5;

?>
```

Файл lib/common.php должен содержать следующую информацию.

```
<?php
```

```

require_once ("../../../../settings.php");
require_once ("./lib/widgetsession.phpm");

require_once ($GLOBALS["smarty-path"] . 'Smarty.class.php');
$smarty = new Smarty;

$session = new WidgetSession(
    array ('phptype' => $GLOBALS ["db-type"],
           'hostspec' => $GLOBALS ["db-hostname"],
           'database' => $GLOBALS ["db-name"],
           'username' => $GLOBALS ["db-username"],
           'password' => $GLOBALS ["db-password"]));
$session->Impress();

?>

```

Возможно, регистрационные данные придется вводить не только на странице index.php. Добавьте проверку регистрационных данных в файле common.php.

```

/*
 * требуется регистрация
 */
$scriptname = end(explode("/", $_SERVER["REQUEST_URI"]));
if ($scriptname <> "index.php") {
    if ($session->isLoggedin() == false) {
        Header ("Location: index.php");
    }
}

```

Теперь обратимся к файлу index.php, который после внесенных изменений выглядит следующим образом.

```

<?php

require_once ("lib/common.php");

if (array_key_exists("action", $_REQUEST)) {
    switch ($_REQUEST["action"]) {
        case "login":
            $session->login($_REQUEST["login_name"], $_REQUEST["login_pass"]);
            if ($session->isLoggedin()) {
                $smarty->assign_by_ref("user", $session->getUserObject());
                $smarty->display ("main.tpl");
                exit;
            } else {
                $smarty->assign('error', "Неверные данные. Попробуйте еще раз.");
                $smarty->display ("login.tpl");
                exit;
            }
            break;
        case "logout";
            $session->logout();
            $smarty->display ("login.tpl");
            exit;
            break;
        default:
            $smarty->display ("login.tpl");
            exit;
    }
} else {
    if ($session->isLoggedin() == true) {
        $smarty->assign_by_ref("user", $session->getUserObject());
        $smarty->display ("main.tpl");
    }
}

```

```

    exit;
}
}

$smarty->display ("login.tpl");

?>

```

При попытке регистрации вы либо перейдете к шаблону main.tpl (в случае ввода правильных данных), либо вернетесь к странице login.tpl при неудачной регистрации. При выходе из системы осуществляется переход на страницу регистрации.

Хотя этот код с технической точки зрения является вполне работоспособным, он далек от совершенства. Он выглядит слишком сложным, поэтому стоит потратить некоторое время и сделать его более читабельным. Другими словами, целесообразно выполнить рефакторинг кода.

Рефакторинг — процесс субъективный, но в результате вы получаете более эстетичный код.

Один из методов рефакторинга предполагает устранение дублирования за счет добавления дублирования.

Просмотрите код на предмет схожих фрагментов. Например, следующий код проходит все тесты и запускается браузером.

```

<?php

require_once ("lib/common.php");

if (array_key_exists("action", $_REQUEST)) {
    switch ($_REQUEST["action"]) {
        case "login":

            $session->login($_REQUEST["login_name"],
                            $_REQUEST["login_pass"]);
            if ($session->isLoggedin()) {
                $smarty->assign_by_ref("user", $session->getUserObject());
                $smarty->display ("main.tpl");
                exit;
            } else {
                if (array_key_exists("login_name", $_REQUEST)) {
                    $smarty->assign('error', "Неверные данные, попробуйте еще раз.");
                }
                $smarty->display ("login.tpl");
                exit;
            }

        break;
        case "logout":
            $session->logout();

            if ($session->isLoggedin()) {
                $smarty->assign_by_ref("user", $session->getUserObject());
                $smarty->display ("main.tpl");
                exit;
            } else {
                if(array_key_exists("login_name", $_REQUEST)) {
                    $smarty->assign('error', "Неверные данные, попробуйте еще раз.");
                };
                $smarty->display ("login.tpl");
                exit;
            }
    }
}

```

```

    exit;
    break;
default:

    if ($session->isLoggedIn()) {
        $smarty->assign_by_ref("user", $session->getUserObject());
        $smarty->display ("main.tpl");
        exit;
    } else {
        if(array_key_exists("login_name", $_REQUEST)) {
            $smarty->assign('error', "Неверные данные, попробуйте еще раз.");
        }
        $smarty->display ("login.tpl");
        exit;
    }

    exit;
}
else {

    if ($session->isLoggedIn()) {
        $smarty->assign_by_ref("user", $session->getUserObject());
        $smarty->display ("main.tpl");
        exit;
    } else {
        if (array_key_exists("login_name", $_REQUEST)) {
            $smarty->assign('error', "Неверные данные, попробуйте еще раз.");
        }
        $smarty->display ("login.tpl");
        exit;
    }
}

$smarty->display ("login.tpl");

?>
```

В этом коде добавлены одинаковые фрагменты (они выделены серым фоном). В данном случае мы воспользовались технологией копирования и вставки, но здесь она оправдана.

Теперь скопируйте общий фрагмент и поместите его в отдельную функцию. Этот прием рефакторинга называется *извлечением метода* (extract method) (см. книгу *Refactoring* Мартина Фовлера (Martin Fowler), которая вышла в издательстве Addison-Wesley).

```
function displaySmartyPage (&$smarty, &$session, $pageToDisplay) {
    if ($session->isLoggedIn()) {
        $smarty->assign_by_ref("user", $session->getUserObject());
        $smarty->display ($pageToDisplay);
        exit;
    } else {
        if (array_key_exists("login_name", $_REQUEST)) {
            $smarty->assign('error', "Неверные данные, попробуйте еще раз.");
        }
        $smarty->display ("login.tpl");
        exit;
    }
}
```

Теперь внесите правки в файл `index.php` с учетом новой функции `displaySmartyPage()`. Файл `index.php` примет следующий вид.

```
if (array_key_exists("action", $_REQUEST)) {
    switch ($_REQUEST["action"]) {
        case "login":
            $session->login($_REQUEST["login_name"],
                $_REQUEST["login_pass"]);

            displaySmartyPage ($smarty, $session, "main.tpl");

            break;
        case "logout":
            $session->logout();

            displaySmartyPage ($smarty, $session, "main.tpl");

            break;
        default:
            displaySmartyPage ($smarty, $session, "main.tpl");
    }
} else {

    displaySmartyPage ($smarty, $session, "main.tpl");
}

displaySmartyPage ($smarty, $session, "main.tpl");
```

Код стал более прозрачным. Очевидно, что логику управления можно привести к следующему виду.

```
if (array_key_exists("action", $_REQUEST)) {
    switch ($_REQUEST["action"]) {
        case "login":
            $session->login($_REQUEST["login_name"],
                            $_REQUEST["login_pass"]);
            break;
        case "logout":
            $session->logout();
            break;
    }
}

displaySmartyPage ($smarty, $session, "main.tpl");
```

Поскольку функция `displaySmartyPage()` теперь вызывается лишь один раз, вы можете усомниться в ее необходимости.

Действительно, функция `displaySmartyPage()` теперь не нужна, и ее можно удалить, воспользовавшись приемом рефакторинга “создание встроенной функции”.

```
<?php  
  
require_once ("lib/common.php");  
  
if (array_key_exists("action", $_REQUEST)) {  
    switch ($_REQUEST["action"]) {  
        case "login":  
            $session->login($_REQUEST["login_name"] ,  
                            $_REQUEST["login_pass"]);  
    }  
}
```

```

        break;
    case "logout":
        $session->logout();
        break;
    }

if ($session->isLoggedIn()) {
    $smarty->assign_by_ref("user", $session->getUserObject());
    $smarty->display ("main.tpl");
} else {
    if (array_key_exists(login_name", $_REQUEST)) {
        $smarty->assign('error', "Неверные данные, попробуйте еще раз.");
    }
    $smarty->display ("login.tpl");
}

?>

```

Приложив значительные усилия, мы существенно упростили код, избавились от лишних обращений к шаблону Smarty и уменьшили общее количество строк кода более чем на треть.

Поэтому не бойтесь заниматься чисткой кода и удалять излишнюю функциональность.

Завершение итерации

Получив навыки рефакторинга, примените их для усовершенствования остального кода.

Сценарий 14 — “Сохранение данных при изменении недели”

Сохранить предыдущее состояние несложно, поскольку оно хранится в объекте ContactVisit. Поэтому добавление следующей функции в файл customer-contacts.php позволит решить половину этой задачи.

```

function gatherContactVisits ($dbh, $emp_id) {
    $result = $dbh->query ("select * from contact_visits where
                           emp_id = ".$emp_id." and week_start =
                           '".getCurrentStartWeek()." order
                           by seq");
    if (DB::isError($result)) {
        return array();
    }
    $visits = array();
    while ($row =& $result->fetchRow()) {
        array_push ($visits, new ContactVisit($row));
    }
    return $visits;
}

```

Теперь присвоим данные визита по ссылке объекту ContactVisit и выведем их на экран с помощью шаблона customer-contacts.tpl.

```

<input name="company_name_{$smarty.section.idx.index}" size="20"
maxlength="50" value="{$contactVisits[idx]->company_name}">

```

Теперь осталось изменить содержимое раскрывающегося списка при переходе на следующую неделю. Это нетривиальная задача.

Для ее решения создадим скрытую форму, содержащую поле week_start. При изменении даты будем вызывать функцию reload() на языке JavaScript, заполняющую переменную week_start новым значением.

```
{literal}
<SCRIPT TYPE="text/javascript">
<!--

function reload () {
    window.document.forms[0].week_start.value =
window.document.forms[1].week_start.value // скрытая форма
    window.document.forms[0].submit(); // скрытая форма
}

// -->
</SCRIPT>
{/literal}
```

<h3>Отчет о контактах</h3>

```
<form action="customer-contacts.php" method="post">
<input type="hidden" name="action" value="reload_contact">
<input type="hidden" name="week_start" value="">
</form>
```

```
<form action="customer-contacts.php" method="post">
<table border="0" width="100%">
<tr><td><b>Имя сотрудника:</b></td><td>{$user->first_name}<br>{$user->last_name}</td>
<td><b>Отдел:</b></td><td>{$user->department}</td></tr>
<tr><td><b>Номер:</b></td><td>{$user->id}</td>
<td><b>Начальная неделя:</b></td>
```

```
<td><SELECT NAME="week_start" onchange="reload()">{$html_options
values=$start_weeks
output=$start_weeks selected=$current_start_week}</SELECT></td></tr>
```

</table>

<hr>

Сценарий 15 — “Еженедельное добавление данных в отчет о контактах с покупателями”

Напомним, что заказчик просит еженедельно обновлять следующие данные: количество звонков в магазин, количество звонков по техническим вопросам, число звонков дистрибуторам, примерный километраж, “ожваченная” территория и комментарии относительно территории.

Отметим, что визиты связаны с новой структурой contact отношением “многие к одному”.

```
CREATE TABLE contact (
    emp_id integer NOT NULL default '0',
    week_start date NOT NULL,
    shop_calls integer default NULL,
```

```

distributor_calls integer default NULL,
engineer_calls integer default NULL,
mileage decimal(9,2) default NULL,
territory_worked varchar(60) default NULL,
territory_comments text
);
CREATE UNIQUE INDEX co_pk ON contact (emp_id, week_start);
CREATE INDEX co_emp_id ON contact (emp_id);
CREATE INDEX co_week_start ON contact (week_start);

```

Это не проблема, поскольку класс Contact аналогичен классу ContactVisit. Создадим тест для проверки информации о контактах в базе данных.

```

function testContactPersistence() {
    $this->session->getDatabaseHandle()->query(
        "delete FROM contact WHERE emp_id = 1 and
         week_start = '1980-01-01'"); // удаление повторов
    $c = new Contact (
        array ("emp_id" => "1",
               "week_start" => "1980-01-01",
               "shop_calls" => 2,
               "distributor_calls" => 3,
               "engineer_calls" => 4,
               "mileage" => 50,
               "territory_worked" => "Central Ohio",
               "territory_comments" => "Buckeyes are great. " ),
        $this->_session->getDatabaseHandle());
    $result = $this->_session->getDatabaseHandle()->query(
        "select * FROM contact WHERE emp_id = 1 and
         week_start = '1980-01-01'");
    $this->assertEquals(0, $result->numRows());
    $c->persist ();
    $result = $this->_session->getDatabaseHandle()->query(
        "select * FROM contact WHERE emp_id = 1 and
         week_start = '1980-01-01'");
    $this->assertEquals(1, $result->numRows());
}

```

Класс Contact во многом аналогичен классу ContactVisit. Это важное замечание, которое можно будет использовать в процессе рефакторинга. Действительно, существующие классы ContactVisit, WidgetUser и вновь создаваемый класс Contact имеют одинаковую функциональность.

Напомним содержимое классов ContactVisit, WidgetUser.

```

class ContactVisit {

    protected $contentBase = array();

    protected $dbh = null; // дескриптор базы данных

    function __get ($key) {}

    function __construct ($results, $dbh = null) {}
    private function isEmpty($key) {}
    public function isValid() {}
    private function implodeQuoted ()
    private function generateSqlInsert ($tableName, &$metas, &$values) {}
    public function persist() {}

}

class WidgetUser {

```

```

protected $contentBase = array ();

protected $dispatchFunctions = array ("role" => "getrole");
function __construct($initdict) {}

function __get ($key) {}

function __set ($key, $value) {}
public function getRole() {}
public function isSalesPerson() {}
public function isSalesManager() {}
public function isAccountant() {}

}

```

Содержимое этих классов можно объединить, введя новый класс `PersistableObject`, реализующий функциональность связи с базой данных. Этот метод рефакторинга называется *извлечением класса* (extract class).

```

class PersistableObject {

    protected $contentBase = array();

    protected $dbh = null; // дескриптор базы данных
    protected $dispatchFunctions = array ("role" => "getrole");

```

```

function __get ($key) {
    // вызов функции
    if (array_key_exists ($key, $this->dispatchFunctions)) {
        $funcname = $this->dispatchFunctions[$key];
        return $this->$funcname();
    }

    // получение состояния
    if (array_key_exists ($key, $this->contentBase)) {
        return $this->contentBase[$key];
    }

    // значение данного ключа
    return $this->$key;
}

```

```

function __construct ($results, $dbh = null) {
    $this->dbh = $dbh;
    if ($results <> null) {
        $this->contentBase = $results; // копирование
    }
}

```

```

public function implodeQuoted(&$values, $delimiter) {
    $sql = "";
    $flagIsFirst = true;
    foreach ($values as $value) {
        if ($flagIsFirst) {
            $flagIsFirst = false;
        } else {
            $sql .= $delimiter;
        }

        if (gettype ($value) == "string") {
            $sql .= "'".$value."'";
        }
    }
}

```

```

    } else {
        $sql .= $value;
    }
}
return $sql;
}

public function generateSqlInsert ($tableName, &$metas, &$values) {
    return " insert into ".$tableName.
        " (" .implode ($metas, ", ") .") ".
        " values (" . $this->implodeQuoted ($values, ", ") .") ";
}

public function generateSqlUpdate ($tableName, &$metas, &$values) {
    $sql = " update ".$tableName." set ";
    for ($i=0; $i<count($metas); $i++) {
        $sql .= $metas[$i]." = ".$values [$i].", ";
    }
    return $sql;
}

public function generateSqlDelete ($tableName) {
    return " delete from \"\".$tableName."\" where ".$this->getSqlWhere();
}

```

```

// примечание: необходимо реализовать
// в конкретных классах
public function getSqlWhere() {
    return "";
}

```

```

public function isValid() {
    return true;
}

public function persistWork ($tablename, $meta) {
    if ($this->isValid() == false) return false;
    $values = array();
    foreach ($meta as $mvalue) {
        array_push ($values, $this->$mvalue);
    }
}

```

```

if (strlen($this->getSqlWhere()) > 0) {
    $sql = $this->generateSqlDelete ($tablename);
    $this->dbh->query($sql);
}

```

```

$sql = $this->generateSqlInsert ($tablename, $meta, $values);
if (DB::isError ($this->dbh->query($sql))) return false;
return true;
}

```

Функции `generateSqlUpdate()` и `generateSqlDelete()` отвечают за обновление информации о данном объекте в базе данных и его удалении, соответственно.

В обеих функциях используется SQL-оператор `WHERE`. Поэтому в дочерних классах (в частности, `ContactVisit`) необходимо выделять характерную для них информацию. Это делается с помощью реализации функции `getSqlWhere()`.

После создания объекта `PersistableObject` и перемещения в него соответствующих функций (`implodeQuoted()`, `generateSqlInsert()`, `isValid()`, `PersistWork()`), а также данных о состоянии (`contentBase`, `dispatchFunctions`) классы `ContactVisit`, `WidgetUser` и `Contact` существенно упростятся.

```
class Contact extends PersistableObject {
    function __construct ($results, $dbh = null) {
        parent::__construct ($results, $dbh);
    }

    public function persist() {
        return $this->persistWork ("contact", array (
            "emp_id",
            "week_start",
            "shop_calls",
            "distributor_calls",
            "engineer_calls",
            "mileage",
            "territory_worked",
            "territory_comments"));
    }

    public function getSqlWhere() {
        return " emp_id = ".$this->emp_id."
                and week_start = '".$this->week_start."'";
    }
}
```

```
class ContactVisit extends PersistableObject {

    function __construct ($results, $dbh = null) {
        parent::__construct ($results, $dbh);
        static $sequence = 0;
        $sequence = $sequence + 1; // инкрементирование
        $this->contentBase ["seq"] = $sequence;
    }

}
```

```
private function isEmpty($key) {
    if (array_key_exists($key, $this->contentBase) == false) return true;
    if ($this->contentBase[$key] == null) return true;
    if ($this->contentBase[$key] == "") return true;
    return false;
}

public function isValid() {
    if ($this->isEmpty("emp_id") == true) return false;
    if ($this->isEmpty("week_start") == true) return false;
    if ($this->isEmpty("company_name") == true) return false;
    return true;
}
```

```
public function persist() {
    return $this->persistWork("contact_visits", array (
        "emp_id",
        "week_start",
        "seq",
        "company_name",
```

```

        "contact_name",
        "city",
        "state",
        "accomplishments",
        "followup",
        "literature_request" ));
    }
}

```

```

class WidgetUser extends PersistableObject {

    function __construct($initdict) {
        parent::__construct ($initdict);
        $this->dispatchFunctions = array ("role" => "getrole");
        $this->contentBase = $initdict; // копирование
    }

    public function getRole() {
        switch ($this->contentBase["role"] ) {
            case "s": return ("Sales Person");
            case "m": return ("Sales Manager");
            case "a": return ("Accountant");
            default: return ("");
        }
    }

    public function isSalesPerson() {
        if ($this->contentBase["role"] == "s") return true;
        return false;
    }

    public function isSalesManager() {
        if ($this->contentBase["role"] == "m") return true;
        return false;
    }

    public function isAccountant() {
        if ($this->contentBase["role"] == "a") return true;
        return false;
    }
}

```

Как хорошо, что для всех этих модулей существуют тесты. После подобных преобразований необходимо снова протестировать все классы и удостовериться в их корректной работе.

Обратите внимание, что в классе Contact реализована функция `getSqlWhere()`. Поэтому объекты данного класса можно добавлять и удалять. При этом перед вставкой удаляется объект `ContactVisit`, а объект `WidgetUser` только считывает информацию из базы данных. Поэтому в классах `ContactVisit` и `WidgetUser` функция `getSqlWhere()` не требуется.

Различие между классами `Contact` и `ContactVisit` проявляется в процессе их использования, поэтому для каждого из этих классов необходимо создать отдельный модульный тест.

В шаблоне `customer-contacts.tpl` необходимо добавить новый раздел, отражающий информацию об объектах `Contact`.

```

<table border="0">
<tr>
<td>Число звонков в магазины:</td><td><input name="shop_calls" size="7" maxlength="17" value="{$contact->shop_calls}"></td><td width="20"></td>
<td>Число звонков по техническим вопросам:</td><td><input name="engineer_calls" size="7" maxlength="17" value="{$contact->engineer_calls}"></td>
</tr>

<tr>
<td>Число звонков дистрибуторам:</td><td><input name="distributor_calls" size="7" maxlength="17" value="{$contact->distributor_calls}"></td>
<td width="20"></td>
<td>Примерный километраж:</td><td><input name="mileage" size="7" maxlength="17" value="{$contact->mileage}"></td>
</tr>

<tr>
<td>Территория:</td><td colspan="2"><input name="territory_worked" value="{$contact->territory_worked}"></td>
</tr>

<tr>
<td colspan="7">Комментарии:<br><TEXTAREA NAME="territory_comments" ROWS=4 COLS=95>{$contact->territory_comments}</TEXTAREA></td>
</tr>
</table>

```

Для считывания и записи информации об объектах Contact добавьте в файл customer-contacts.php следующие функции.

```

function persistContact (&$dbh, $emp_id) {
    $c = new Contact (
        array ("emp_id" => $emp_id,
               "week_start" => getCurrentStartWeek(),
               "shop_calls" => $_REQUEST["shop_calls"],
               "distributor_calls" => $_REQUEST["distributor_calls"],
               "engineer_calls" => $_REQUEST["engineer_calls"],
               "mileage" => $_REQUEST["mileage"],
               "territory_worked" => $_REQUEST["territory_worked"],
               "territory_comments" => $_REQUEST["territory_comments"]),
        $dbh);
    $c->persistent();
}

function gatherContact (&$dbh, $emp_id) {
    $result = $dbh->query ("select * from contact where emp_id = ".$emp_id."
                           and week_start = '".getCurrentStartWeek()."');
    if (DB::isError($result)) return array();
    return new Contact {$result->fetchRow()};
}

$user = $session->getUserObject();

// отображение
if ($_REQUEST["action"] != "persist_contact") {
    $smarty->assign_by_ref ("user", $user);
    $smarty->assign_by_ref ("contact", gatherContact ($session->
        getDatabaseHandle(), $user->id));
    $smarty->assign_by_ref ("contactVisits",
        gatherContactVisits ($session->getDatabaseHandle(), $user->id));
    $smarty->assign ('start_weeks', getStartWeeks ());
    $smarty->assign ('current_start_week', getCurrentStartWeek ());
}

```

```

$smarty->assign("max_weekly_contacts", $GLOBALS["max-weekly-contacts"]);
$smarty->display('customer-contacts.tpl');
exit;
}

// сохранение данных о визитах
require_once ("lib/contact.phpm");
persistContact ($session->getDatabaseHandle(), $user->id);
persistContactVisits ($session->getDatabaseHandle(), $user->id);

```

Результаты ваших усилий показаны на рис. 22.7.

Widget World

Отчет о контактах

Имя сотрудника:	Ed Lecky-Thompson	Отдел:	продажи
Номер:	1	Начальная неделя:	2004-06-13

Важные дистрибуторы и покупатели:
(а также перспективы)

Компания	Контакт	Город	Страна	Ответ	Запрос на документацию
<input type="text"/>					

Результаты:

<input type="text"/>

Число звонков в магазины: Число звонков по техническим вопросам:
 Число звонков дистрибуторам: Примерный километраж:
 Территория:

Комментарии:

Рис. 22.7.

Скорость реализации последнего сценария свидетельствует о том, что исходный код был хорошо оптимизирован.

Однако рефакторинг не следует прекращать. Некоторые разделы все еще требуют пристального внимания, особенно фрагменты кода, содержащие функции классов Contact и ContactVisit.

Однако не следует увлекаться. Процесс рефакторинга должен приводить к упрощению кода. Если в процессе рефакторинга вам приходится тратить массу времени на написание сложных каркасов, следует остановиться и задуматься о необходимости подобных изменений.

Отчет о командировочных расходах

Составим дальнейший план разработки.

Для формирования отчета о командировочных расходах нужны следующие сценарии.

- Сценарий 2. “Отчет о командировочных расходах” — 7 баллов.
- Сценарий 3. “Комментарий к отчету о расходах” — 2 балла.
- Сценарий 10. “Отчет об авансовых платежах” — 2 балла.
- Сценарий 4. “Представление отчета о расходах в виде электронной таблицы” — 4 балла.

Дополнительно нужно реализовать следующие сервисы.

- Сценарий 5. “Уведомление бухгалтера” — 3 бала.
- Сценарий 7. “Уведомление продавца” — 1 балл.
- Сценарий 13. “Экспорт данных из отчета о расходах” — 3 балла.

И наконец, придется реализовать еще два сценария.

- Сценарий 6. “Уведомление менеджера” — 3 балла.
- Сценарий 8. “Уведомление службы рассылки и канцелярии” — 2 балла.

Как видно из рис. 22.2, отчет о расходах содержит не слишком много деталей. Однако в процессе обсуждений с заказчиком выяснилось, что он хотел бы придать ему форму, показанную на рис. 22.8.

Необходимо продумать, с какой частотой нужно обновлять данные в базе. По возможности значения полей необходимо вычислять. Для начала не будем усложнять себе жизнь, а реализуем самые простые вещи. К ним относятся сумма в долларах, дата и описание платежа. Эти данные можно хранить в следующей таблице.

```
CREATE TABLE travel_expense_item (
    emp_id      integer      NOT NULL,
    expense_date date        NOT NULL,
    description  varchar(40)  NOT NULL,
    amount       decimal(9,2) NOT NULL
);
```

Добавим к ним информацию о текущей неделе.

```
CREATE TABLE travel_expense_week (
    emp_id      integer      NOT NULL,
    week_start   date        NOT NULL,
    comments     text,
    territory_worked varchar(60),
    cash_advance decimal(9, 2),
    mileage_rate decimal(3,2) NOT NULL );
```

Необходимо также добавить используемые по умолчанию значения, индексы и другие данные, позволяющие отслеживать выданный аванс и пройденное расстояние. Итоги будут подводиться еженедельно. Их можно вычислить в соответствующем поле на форме, но сохранять в базе данных необязательно.

Элемент командировочных расходов

Создайте следующие тесты. В простейшем случае они просто проверяют наличие минимально необходимой информации.

```
function testValidTravelExpenseItem() {
    $tvi = new TravelExpenseItem (
        array ('emp_id'      = > "1",
```

в	с	в	е	г	ч	п	т	с	н
Имя сотрудника:	XXXXXX XXXXXXXX			Отдел:	XXXXXX				
Номер:	X			Начальная неделя:	YYYY-MM-DD (drop down)				
ТERRитория:	XXXXXXXXXXXXXX								
	Вс	Пн	Вт	Ср	Чт	Пт	Сб	Всего	
Проживание									
Проживание & Отель	1	1	1	1	1	1	1	1	7
Другое		1	1	1	1	1	1	1	7
Чаевые		1	1	1	1	1	1	1	7
Всего	3	3	3	3	3	3	3	3	21
Питание									
Завтрак		1	1	1	1	1	1	1	7
Ланч		1	1	1	1	1	1	1	7
Обед		1	1	1	1	1	1	1	7
Чаевые		1	1	1	1	1	1	1	7
Развлечения		1	1	1	1	1	1	1	7
Всего	6	6	6	6	6	6	6	6	42
Транспортные расходы									
Самолет		1	1	1	1	1	1	1	7
Аренда авто		1	1	1	1	1	1	1	7
Бензин		1	1	1	1	1	1	1	7
Местный проезд и парковка	1	1	1	1	1	1	1	1	7
Всего	4	4	4	4	4	4	4	4	28
Километраж	1	1	1	1	1	1	1	1	7
Стоимость / км	1	1	1	1	1	1	1	1	7
Итого	6	6	6	6	6	6	6	6	42
Разное									
Подарки		1	1	1	1	1	1	1	7
Телефон & Факс		1	1	1	1	1	1	1	7
Заказы		1	1	1	1	1	1	1	7
Почта		1	1	1	1	1	1	1	7
Другое		1	1	1	1	1	1	1	7
Всего	5	5	5	5	5	5	5	5	35
Комментарии:					Всего				147
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX					Аванс				47
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX					По сотруднику				100
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX					По компании				

Рис. 22.8.

```

        'expense_date' => "1980-01-01",
        'description'  => "one",
        'amount'        => "1.0"));
$this->assertEquals(true, $v->isValid(), "valid expense");
}

```

```

function testInvalidTravelExpenseItem() {
    $tvi = new TravelExpenseItem (
        array ('emp_id'      => "1",
              'expense_date' => "", // требуется дата
              'description'  => "one",
              'amount'       => "1.0"));
    $this->assertEquals(false, $tvi->isValid(), "valid expense");
}

```

Что еще нужно сделать? Эти данные могут понадобиться для заполнения массива, поэтому нужно предусмотреть их сохранение в базе данных. Добавим еще один тест.

```

function testTravelExpenseItemPersistence() {
    $this->_session->getDatabaseHandle()->query(
        "delete FROM travel_expense_item WHERE emp_id=1
         and expense_date='1980-01-01'"); // удаление повторов
    $tvi = new TravelExpenseItem (
        array ('emp_id'      => "1",
              'expense_date' => "1980-01-01",
              'description'  => "one",
              'amount'       => "1.0"),
        $this->_session->getDatabaseHandle());
    $result=$this->_session->getDatabaseHandle()->query(
        "select * FROM travel_expense_item WHERE
         emp_id = 1 and expense_date='1980-01-01'");
    $this->assertEquals(0, $result->numRows());
    $tvi->persist();
    $result=$this->_session->getDatabaseHandle()->query(
        "select * FROM travel_expense_item WHERE
         emp_id = 1 and expense_date = '1980-01-01'");
    $this->assertEquals(1, $result->numRows());
}

```

При выполнении этих тестов необходимо установить соединение с тестовой базой данных.

В новом файле lib/expense.php создайте шаблон для класса элемента командировочных расходов.

```

class TravelExpenseItem {
    public function isValid() { }
    public function persist() { }

}

```

Удостоверьтесь в работоспособности теста.

Теперь можно приступить к реализации класса TravelExpenseItem по образцу класса ContactVisit. Это чрезвычайно просто.

```

class TravelExpenseItem extends PersistableObject {
    function __construct ($results, $dbh = null) {
        parent::__construct ($results, $dbh);
    }

    private function isEmpty($key) {
        if (array_key_exists($key, $this->contentBase) == false) return true;
        if ($this->contentBase[$key] == null) return true;
        if ($this->contentBase[$key] == "") return true;
        return false;
    }
}

```

```

public function isValid() {
    if ($this->isEmpty("emp_id") == true) return false;
    if ($this->isEmpty("expense_date") == true) return false;
    if ($this->isEmpty("description") == true) return false;
    if ($this->isEmpty("amount") == true) return false;
    return true;
}

public function persist() {
    return $this->persistWork (
        "travel_expense_item",
        array ("emp_id",
            "expense_date",
            "description",
            "amount" ));
}

public function getSqlWhere() {
    return " emp_id = ".$this->emp_id." and
        expense_date = '".$this->expense_date."' and
        description = '".$this->description."'";
}
}

```

Вспомним определение класса ContactVisit.

```

class ContactVisit extends PersistableObject {
    function __construct ($results, $dbh = null) {}
    private function isEmpty($key) {}
    public function isValid() {}
    public function persist() {}
}

```

Теперь функция isEmpty() дублируется в двух классах, поэтому ее необходимо перенести в базовый класс PersistableObject. Не забудьте сделать ее защищенной, иначе подклассы не смогут получить к ней доступ.

Удостоверьтесь, что класс ContactVisit по-прежнему удовлетворяет всем тестам. Полученные классы имеют следующий вид.

```

class PersistableObject {
    protected $contentBase = array();
    protected $dbh = null; // дескриптор базы данных
    protected $dispatchFunctions = array ("role" => "getrole");
    function __get ($key) {}
    function __construct ($results, $dbh = null) {}
    public function implodeQuoted (&$values, $delimiter) {}
    public function generateSqlInsert ($tableName, &$metas, &$values) {}
    public function generateSqlUpdate ($tableName, &$metas, &$values) {}
    public function generateSqlDelete ($tableName) {}
    public function getSqlWhere() {}
    public function isValid() {}
    protected function isEmpty($key) {}
    public function persistWork ($tablename, $meta) {}
}

class ContactVisit extends PersistableObject {
    function __construct ($results, $dbh = null) {}
    public function isValid() {}
    public function persist() {}
}

class TravelExpenseItem extends PersistableObject {
    function __construct ($results, $dbh = null) {}

```

```

public function isValid() {}
public function persist() {}
public function getSqlWhere() {}
}

```

Теперь можно сделать небольшой перерыв.

Вычисление командировочных расходов за неделю

Теперь можно разработать простые тесты для еженедельного отчета о командировочных расходах.

```

function testValidTravelExpenseWeek() {
    $tvi = new TravelExpenseWeek (
        array ('emp_id'          => "1",
              'week_start'      => "1980-01-01",
              'comments'        => "comment",
              'mileage_rate'   => "0.31",
              'territory_worked' => "Midwest" ));
    $this->assertEquals(true, $tvi->isValid(), "valid expense");
}

function testInvalidTravelExpenseWeek() {
    $tvi = new TravelExpenseWeek (
        array ('emp_id'          => "1",
              'week_start'      => "", // требуется дата
              'comments'        => "comment",
              'mileage_rate'   => "0.31",
              'territory_worked' => "Midwest" ));
    $this->assertEquals(false, $tvi->isValid(), "valid expense");
}

function testTravelExpenseWeekPersistence() {
    $this->_session->getDatabaseHandle()->query(
        "delete FROM travel_expense_week WHERE
         emp_id = 1 and week_start = '1980-01-01'"); // удаление повторов
    $tvi = new TravelExpenseWeek (
        array ('emp_id'          => "1",
              'week_start'      => "1980-01-01",
              'comments'        => "comment",
              'territory_worked' => "Midwest",
              'mileage_rate'   => "0.31",
              'cash_advance'   => "0"),
        $this->_session->getDatabaseHandle());
    $result=$this->_session->getDatabaseHandle()->query(
        "select * FROM travel_expense_week WHERE
         emp_id = 1 and week_start = '1980-01-01'");
    $this->assertEquals (0, $result->numRows(), "pre check");
    $tvi->persist();
    $result=$this->_session->getDatabaseHandle()->query(
        "select * FROM travel_expense_week WHERE
         emp_id = 1 and week_start = '1980-01-01'");
    $this->assertEquals(1, $result->numRows(), "persist");
}

```

Какой класс должен отвечать за реализацию жизненного цикла элементов `TravelExpenseItem`? Существует два класса-кандидата: `TravelExpenseWeek` и сам класс `TravelExpenseItem`. Какой из этих классов обеспечит лучший контейнер для хранения итоговых значений за неделю?

Тест для класса-контейнера `TravelExpenseWeek` будет выглядеть следующим образом.

```
function testTravelExpenseWeekContainerRead() {  
    // очистка тестовой базы данных  
    $this->_session->getDatabaseHandle()->query(  
        "delete FROM travel_expense_item WHERE  
        emp_id = 1 and expense_date >= '1980-01-06'  
        and expense_date <= '2001-09-15'");  
  
    $dbh = $this->_session->getDatabaseHandle();  
  
    // понедельник  
    $item1 = new TravelExpenseItem (  
  
        array ('emp_id' => "1", 'expense_date' => "1980-01-06",  
              'description' => "lodging_and_hotel", 'amount' => "1.1"),  
  
        $dbh);  
    $item2 = new TravelExpenseItem (  
        array ('emp_id' => "1", 'expense_date' => "1980-01-06",  
  
              'description' => "meals_breakfast", 'amount' => "2.2" ),  
  
        $dbh);  
  
    $item3 = new TravelExpenseItem (  
        array ('emp_id' => "1", 'expense_date' => "1980-01-06",  
  
              'description' => "misc_supplies", 'amount' => "3.3" ),  
  
        $dbh);  
  
    // вторник  
    $item4 = new TravelExpenseItem (  
  
        array ('emp_id' => "1", 'expense_date' => "2001-09-10",  
              'description' => "lodging_and_hotel", 'amount' => "4.4" ),  
  
        $dbh);  
    $item5 = new TravelExpenseItem (  
        array ('emp_id' => "1", 'expense_date' => "2001-09-10",  
  
              'description' => "meals_breakfast", 'amount' => "5.5"),  
  
        $dbh);  
    $item6 = new TravelExpenseItem (  
        array ('emp_id' => "1", 'expense_date' => "2001-09-10",  
  
              'description' => "misc_supplies", 'amount' => "6.6"),  
  
        $dbh);  
  
    // среда  
  
    $item7 = new TravelExpenseItem (  
  
        array ('emp_id' => "1", 'expense_date' => "1980-01-01",  
              'description' => "lodging_and_hotel", 'amount' => "7.7"),  
  
        $dbh);  
    $item8 = new TravelExpenseItem (  
        array ('emp_id' => "1", 'expense_date' => "1980-01-01",  
  
              'description' => "lodging_and_hotel", 'amount' => "7.7"),  
  
        $dbh);
```

```
'description' => "meals_breakfast", 'amount' => "8.8") ,  
  
$dbh);  
$item9 = new TravelExpenseItem (  
    array ('emp_id' => "1", 'expense_date' => "1980-01-01",  
  
        'description' => "misc_supplies", 'amount' => "9.9") ,  
  
    $dbh);
```

Данные для первых трех дней недели заполняются с помощью класса `TravelExpenseItem`. Для каждого из этих дней создаются различные типы элементов: `meals_breakfast`, `misc_supplies` и `lodging_and_hotel`.

```
$item1->persist();  
$item2->persist();  
$item3->persist();  
$item4->persist();  
$item5->persist();  
$item6->persist();  
$item7->persist();  
$item8->persist();  
$item9->persist();  
  
$week = new TravelExpenseWeek (  
    array ('emp_id' => "1" ,  
        'week start' => "1980-01-06") ,  
    $this->_session->getDatabaseHandle());  
  
$week->readWeek();  
  
// понедельник  
$this->assertEquals(1.1, (float)  
    $week->getExpenseAmount(0, 'lodging_and_hotel'));  
$this->assertEquals (2.2, (float)  
    $week->getExpenseAmount (0 , 'meals_breakfast'));  
$this->assertEquals(3.3, (float)  
    $week->getExpenseAmount(0, 'misc_supplies'));  
  
// вторник  
$this->assertEquals(4.4, (float)  
    $week->getExpenseAmount (1 , 'lodging_and_hotel'));  
$this->assertEquals(5.5, (float)  
    $week->getExpenseAmount(1, 'meals_breakfast'));  
$this->assertEquals(6.6, (float)  
    $week->getExpenseAmount(1, 'misc_supplies'));  
  
// среда  
$this->assertEquals(7.7, (float)  
    $week->getExpenseAmount(2, 'lodging_and_hotel'));  
$this->assertEquals(8.8, (float)  
    $week->getExpenseAmount(2, 'meals_breakfast'));  
$this->assertEquals(9.9, (float)  
    $week->getExpenseAmount(2, 'misc_supplies'));  
}
```

После записи в базу данных элементов `TravelExpenseItem` считывать эту информацию может объект `TravelExpenseWeek`, если указан идентификатор сотрудника и определен день начала недели.

Этот тест важен и в других аспектах. Дело в том, что он определяет интерфейс создаваемой Web-страницы. Напомним, что элементы `TravelExpenseItem` пополняются каждый день, а Web-страница обновляется еженедельно.

Поскольку объект TravelExpenseWeek “знает”, какой день недели является первым (например, в Америке это воскресенье, а в России — понедельник), дни недели можно нумеровать по порядку, не указывая конкретного наименования дня.

Последние штрихи

Рассмотрим, как следует выводить информацию на экран.

Вернемся к рис. 22.8. Заметим, что некий объект должен позаботиться о выводе промежуточных итоговых результатов.

Кроме того, необходимо обеспечить соответствие осмысленных названий и элементов, хранящихся в базе данных.

Вряд ли для этих целей следует использовать жесткий код HTML. Следует подумать о том, как можно упростить себе жизнь.

Чтобы избежатьочных кошмаров, связанных с поддержкой HTML-кода, рассмотрим следующую структуру данных, которую можно использовать для генерации и отображения полей данных на экране.

```
array(
    array('name' => 'Проживание',
          'code' => 'lodging',
          'data' => array('Проживание & Отель', 'Другое', 'Чаевые'),
          'persist' => array('lodging_and_hotel', 'lodging_other',
                             'lodging_tips')),

    array('name' => 'Питание',
          'code' => 'meals',
          'data' => array('Завтрак', 'Ланч',
                          'Обед', 'Чаевые', 'Развлечения'),
          'persist' => array('meals_breakfast', 'meals_lunch',
                             'meals_dinner', 'meals_tips',
                             'meals_entertainment')),

    array('name' => 'Транспортные расходы',
          'code' => 'trans',
          'data' => array('Самолет', 'Оренда авто', 'Бензин',
                          'Местный проезд', 'Парковка'),
          'persist' => array ('trans_airfare', 'trans_auto_rental',
                             'trans_auto_maint', 'trans_local',
                             'trans_tolls', 'trans_miles_traveled')),

    array('name' => 'Разное',
          'code' => 'misc',
          'data' => array('Подарки', 'Телефон & Факс', 'Заказы',
                          'Почта', 'Другое'),
          'persist' => array('misc_gifts', 'misc_phone',
                             'misc_supplies', 'misc_postage',
                             'misc_other')));
```

Это многомерный массив, первым измерением которого является массив разделов, т.е. каждый раздел может вычислять сумму своих элементов.

В каждом разделе содержится массив `data`, содержащий осмысленное описание элементов и массив `persist`, в котором хранятся имена соответствующих элементов базы данных. Элемент `code` позволяет автоматизировать создание переменных JavaScript, вычисляемых для каждого раздела. Эта структура данных позволяет интегрировать HTML-код, логику JavaScript, взаимодействие с базой данных и объекты PHP.

Дополнительные тесты для отчета о еженедельных расходах

Важным этапом является взаимодействие объектов PHP с базой данных. На этой стадии определяется, какую информацию необходимо получать с данной страницы. Этую обязанность целесообразно возложить на контейнер `TravelExpenseWeek`.

Напомним соглашения об именовании, позволяющие определить уникальное значение для каждой отдельной ячейки. Элемент формы представляется в виде конката-нации названия раздела, дня недели и номера элемента. Такое соглашение является вполне адекватным.

Приведем тест для анализа запроса.

```
function testTravelExpenseWeekContainerParseRequest() {
    $response = array (
        'lodging_sun_0' => "1.1",
        'meals_sun_0'   => "2.2",
        'misc_sun_2'    => "3.3",
        'lodging_mon_0' => "4.4",
        'meals_mon_0'   => "5.5",
        'misc_mon_2'    => "6.6",
        'lodging_tue_0' => "7.7",
        'meals_tue_0'   => "8.8",
        'misc_tue_2'    => "9.9");
}

$week = new TravelExpenseWeek (
    array ('emp_id'      => "1",
           'week_start' => "1980-01-06"));

$week->parse($response);

$this->assertEquals(1.1,
    $week->getExpenseAmount(0, 'lodging_and_hotel'));

$this->assertEquals(2.2,
    $week->getExpenseAmount(0, 'meals_breakfast'));
$this->assertEquals(3.3,
    $week->getExpenseAmount(0, 'misc_supplies'));

$this->assertEquals(4.4,
    $week->getExpenseAmount(1, 'lodging_and_hotel')); $this->assertEquals(5.5,
    $week->getExpenseAmount(1, 'meals_breakfast')); $this->assertEquals(6.6,
    $week->getExpenseAmount(1, 'misc_supplies'));

$this->assertEquals(7.7,
    $week->getExpenseAmount(2, 'lodging_and_hotel'));
$this->assertEquals(8.8,
    $week->getExpenseAmount(2, 'meals_breakfast'));
$this->assertEquals(9.9,
    $week->getExpenseAmount(2, 'misc_supplies'));
}
```

Массив `response` эмулирует входные данные. Заметим, что значения этого массива и соответствующие им элементы базы данных отражены в приведенной выше структуре данных.

Создадим еще один тест для хранения данных контейнера. Он аналогичен предыдущему тесту для элементов командировочных расходов.

```

function testTravelExpenseWeekContainerWrite() {
    $this->_session->getDatabaseHandle()->query(
        "delete FROM travel_expense_item WHERE
        emp_id = 1 and expense_date >= '1980-01-06'
        and expense_date <= '2001-09-15'");

    $response = array (
        'lodging_sun_0'=>"1.1", 'meals_sun_0'=>"2.2", 'misc_sun_2'=>"3.3",
        'lodging_mon_0'=>"4.4", 'meals_mon_0'=>"5.5", 'misc_mon_2'=>"6.6",
        'lodging_tue_0'=>"7.7", 'meals_tue_0'=>"8.8", 'misc_tue_2'=>"9.9");
}

$week = new TravelExpenseWeek {
    array ('emp_id'          => "1",
           'week_start'       => "1980-01-06",
           'territory_worked' => "Midwest",
           'comments'         => "comment",
           'cash_advance'     => "0",
           'mileage_rate'     => "0.31"),
    $this->_session->getDatabaseHandle());
}

$week->parse($response);
$this->assertEquals(true, $week->persist());

$week = new TravelExpenseWeek {
    array ('emp_id'          => "1",
           'week_start'       => "1980-01-06",
           'territory_worked' => "Midwest",
           'comments'         => "comment",
           'cash_advance'     => "0",
           'mileage_rate'     => "0.31"),
    $this->_session->getDatabaseHandle());
}

$week->readWeek();

$this->assertEquals(1.1, (float)
    $week->getExpenseAmount(0, 'lodging_and_hotel'));
$this->assertEquals(2.2, (float)
    $week->getExpenseAmount(0, 'meals_breakfast'));
$this->assertEquals(3.3, (float)
    $week->getExpenseAmount(0, 'misc_supplies'));

$this->assertEquals(4.4, (float)
    $week->getExpenseAmount(1, 'lodging_and_hotel'));
$this->assertEquals(5.5, (float)
    $week->getExpenseAmount(1, 'meals_breakfast'));
$this->assertEquals(6.6, (float)
    $week->getExpenseAmount(1, 'misc_supplies'));

$this->assertEquals(7.7, (float)
    $week->getExpenseAmount(2, 'lodging_and_hotel'));
$this->assertEquals(8.8, (float)
    $week->getExpenseAmount(2, 'meals_breaKfast'));
$this->assertEquals(9.9, (float)
    $week->getExpenseAmount(2, 'misc_supplies'));
}
}

```

В этом тесте объект TravelExpenseWeek анализирует объект response, записывает элементы отчета в базу данных, считывает их и сверяет полученные результаты.

Заметим, что при создании данного кода интенсивно использовался метод копирования и вставки. Поэтому он должен стать объектом рефакторинга. Любой код необходимо приводить к виду, удобному для чтения и модификации.

Реализация класса еженедельного отчета о командировочных расходах

Теперь можно реализовать сам класс отчета о командировочных расходах.

```
class TravelExpenseWeek extends PersistableObject {
    public $items = array();

    function __construct ($results, $dbh = null) {
        parent::__construct ($results, $dbh);
    }

    public function isValid() {
        if ($this->isEmpty("emp_id") == true) return false;
        if ($this->isEmpty("week_start") == true) return false;
        if ($this->isEmpty("territory_worked") == true) return false;
        if ($this->isEmpty("mileage_rate") == true) return false;
        return true;
    }

    public function persist() {
        return $this->persistWork ("travel_expense_week",
            array ( "emp_id",
                    "week_start",
                    "comments",
                    "territory_worked",
                    "cash_advance",
                    "mileage_rate"));
    }

    public function getSqlWhere() {
        return " emp_id = ".$this->emp_id."
               and week_start = '".$this->week_start."'";
    }

    public function parse(&$request) { }
    public function readWeek() { }
    public function getExpenseAmount($offset, $description) { }
}
```

Функции parse(), readWeek() и getExpenseAmount() требуются для тестирования и являются просто заглушками.

Реализация функций анализа запросов

Тест TravelExpenseWeekContainerParseRequest несколько сложнее, поскольку он связан с операциями над данными. В первую очередь следует позаботиться о зависимостях, связанных с определенным выше метамассивом. Эту функциональность можно реализовать в следующем методе класса TravelExpenseWeek, который необходимо поместить в файл lib/expense.php.

```
public function getExpensesMetaArray() {
    array(
```

```

array('name' => 'Проживание',
      'code' => 'lodging',
      'data' => array('Проживание & Отель', 'Другое', 'Чаевые'),
      'persist' => array('lodging_and_hotel', 'lodging_other',
                          'lodging_tips')),

array('name' => 'Питание',
      'code' => 'meals',
      'data' => array('Завтрак', 'Ланч',
                      'Обед', 'Чаевые', 'Развлечения'),
      'persist' => array('meals_breakfast', 'meals_lunch',
                          'meals_dinner', 'meals_tips',
                          'meals_entertainment')),

array('name' => 'Транспортные расходы',
      'code' => 'trans',
      'data' => array('Самолет', 'Оренда авто', 'Бензин',
                      'Местный проезд', 'Парковка'),
      'persist' => array ('trans_airfare', 'trans_auto_rental',
                          'trans_auto_maint', 'trans_local',
                          'trans_tolls', 'trans_miles_traveled')),

array('name' => 'Разное',
      'code' => 'misc',
      'data' => array('Подарки', 'Телефон & Факс', 'Заказы',
                      'Почта', 'Другое'),
      'persist' => array('misc_gifts', 'misc_phone',
                          'misc_supplies', 'misc_postage',
                          'misc_other'))),
}

```

Помимо метамассива необходимо определить способ извлечения элементов отчета из базы данных. Напомним, что в тесте `TravelExpenseWeekContainerParseRequest` вызывается функция `getExpenseAmount()` объекта `TravelExpenseWeek`.

```
$this->assertEquals(8.8, $week->getExpenseAmount(2, 'meals_breakfast'));
```

Пока эта функция реализована для каждого дня недели. Создайте себе напоминание о необходимости добавить ее в класс `TravelExpenseWeek` в файле `lib/expense.phpm`.

```
/**
 * todo: поместить в ассоциативный массив
 */
public function getExpenseAmount($offset, $description) {

    $targetDate = $this->addDays($this->week_start, $offset);
    foreach ($this->items as $item) {

        if ($item->expense_date == $targetDate &&
            $item->description == $description) {
            return $item->amount;
        }
    }
    return "";
}
```

Вызов функции `addDays()` обеспечивает получение строкового представления данных. Добавьте в эту функцию значение времени в секундах ($86400 = 60 \times 60 \times 24$).

Теперь класс `TravelExpenseWeek` будет содержать следующий фрагмент.

```
/***
 * todo: будет ли работать этот код при переходе
```

```

*      на зимнее время?
*/
public function addDays($start, $days) {
    return date("Y-m-d", strtotime($start)+$days*86400);
}

```

Однако этот код содержит ловушку. Как PHP будет возвращать конкретное значение дня? По местному или абсолютному времени? Пока запомните эту проблему и продолжайте реализацию класса.

Ключевой функцией объекта TravelExpenseWeek является функция анализа запроса на командировочные расходы с целью создания ассоциативного массива элементов TravelExpenseItem.

Напомним, что мы используем следующую структуру данных

```

array(
    array('name' => 'Проживание',
          'code' => 'lodging',
          'data' => array('Проживание & Отель', 'Другое', 'Чаевые'),
          'persist' => array('lodging_and_hotel', 'lodging_other',
                             'lodging_tips'))
)

```

для формирования имен объектов вида lodging_sun_0.

Необходимо создать элементы TravelExpenseItem и хранить их в массиве TravelExpenseWeek->items. Добавьте функцию синтаксического анализа в класс TravelExpenseWeek в файл /lib/expense.phpm.

```

/**
 * Эта функция устраняет противоречия между ежедневными
 * данными из базы и их отображением по неделям
 */
public function getWeekArray() {
    return array ('sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat');
}

public function parse (&$request) {

    // цикл по разделам
    foreach ($this->getExpensesMetaArray() as $sectionlist) {

        // цикл по строкам
        for ($i=0; $i<count ($sectionlist['persist']); $i++) {

            $daynum = 0;

            // цикл по дням
            foreach ($this->getWeekArray() as $day) {

                $index = $sectionlist['code']."_".$day."_".$i;
                if (array_key_exists($index, $request) and
                    $request[$index] <> null and
                    $request[$index] <> "") {

                    // создаем новый элемент и сохраняем в
                    // переменной $this->items
                    array_push (
                        $this->items,

```

Эти пять функций составляют основную функциональность класса `TravelExpense-Week`, выполняющего нетривиальную задачу конвертирования структуры данных для Web-представления в базу данных.

Реализация функций записи и считывания

Напомним, что в тесте, предназначенном для проверки взаимодействия с базой данных, содержится следующий фрагмент кода.

```
$week = new TravelExpenseWeek (
array ('emp_id'          => "1",
      'week_start'       => "1980-01-06",
      'territory_worked' => "Midwest",
      'comments'         => "comment",
      'cash_advance'    => "0",
      'mileage_rate'    => "0.31"),
$this-> session->getDatabaseHandle());
```

```
$week->readWeek();
```

Добавьте функцию `readWeek()` в класс `TravelExpenseWeek` (в файл `/lib/travelExpense.php`).

```
public function readWeek() {
    $sql = "select * from travel_expense_week where";
    $sql .= " emp_id = ".$this->emp_id." and ";
    $sql .= " week_start = '".$this->week_start."'";
    $result = $this->dbh->query($sql);
    if (DB::isError($result) <> true and $result->numRows() > 0) {
        $row = $result->fetchRow();
        $this->contentBase['comments'] = $row['comments'];
        $this->contentBase['territory_worked'] =
            $row['territory_worked'];
        $this->contentBase['cash_advance'] =
            $row['cash_advance'];
        $this->contentBase['mileage_rate'] =
            $row['mileage_rate'];
    }
    $sql = "select * from travel_expense_item where";
    $sql .= " emp_id = ".$this->emp_id." and ";

```

```

$sql .= " expense_date >= '". $this->week_start ."' and";
$sql .= " expense_date <= '". $this->addDays($this->week_start, 6) ."'";
$this->items = array();
$result = $this->dbh->query ($sql);
if (DB::isError($result) or $result->numRows() == 0) return;

while ($row = $result->fetchRow()) {
    array_push ($this->items, new TravelExpenseItem($row));
}

}

```

Эта функция, по существу, состоит из двух частей. В первой части из базы данных возвращается состояние объекта `TravelExpenseWeek`, а во второй — соответствующие элементы `TravelExpenseItem`, которые сохраняются в соответствующем массиве.

Обратите внимание на небольшую проблему. В конструкторе объекта `TravelExpenseItem` ему передается целый объект `$row`. Напомним реализацию конструктора базового класса `PersistableObject`.

```

function __construct ($results, $dbh = null) {
    $this->dbh = $dbh;
    if ($results <> null) {
        $this->contentBase = $results; // копирование
    }
}

```

Таким образом, объект `TravelExpenseItem` слепо хранит любую передаваемую ему информацию. Это достаточно расточительно. Хотя в базе данных содержатся только объекты `TravelExpenseItem`, они наследуют свое поведение от `PersistableObject`, для которого `TravelExpenseWeek`, `Contact` и `ContactVisit` являются наследниками. Передача любого из этих объектов в массив `$_REQUEST` приводит к неоправданной трате ресурсов.

Чтобы предотвратить хранение любой информации в объекте `PersistableObject` и ограничить его лишь хранением нужных данных, вернемся на шаг назад и создадим новый тест.

```

function testIgnoreExtra() {
    $response = array ('emp_id'      => "1",
                      'expense_date' => "1980-01-01",
                      'description'  => "one",
                      'amount'        => "1.0",
                      'extra'         => "extra bits");

    $tvi = new TravelExpenseItem($response);

    $this->assertEquals(null, $tvi->extra);
}

}

```

Очевидно, этот тест не будет пройден. Очевидно также, что класс `TravelExpenseWeek` окажется чрезмерно раздутым.

Одним из способов решения проблемы является априорные знания о том, какая информация требуется данному объекту. Поэтому прежнюю функцию `persist()` вида

```

public function persist() {
    return $this->persistentWork ("travel_expense_item",
                                  array ("emp_id",
                                         "expense_date",
                                         ...
                                         )

```

```

        "description",
        "amount"));
}
}
```

можно сделать более согласованной за счет передачи ей некоторого внутреннего состояния.

```

public function persist() {
    return $this->persistWork (
        $this->contentMetaTable,
        $this->contentMetaOnly);
}
```

Итак, мы переписали класс TravelExpenseItem.

```
class TravelExpenseItem extends PersistableObject {
```

```

protected $contentMetaTable = null;
protected $contentMetaOnly = null;
```

```

function __construct ($results, $dbh = null) {
    $this->contentMetaTable = "travel_expense_item";
    $this->contentMetaOnly = array ( "emp_id",
                                    "expense_date",
                                    "description",
                                    "amount");

    $content = array();
    foreach ($this->contentMetaOnly as $key) {
        if (array_key_exists($key, $results)) {
            $content[$key] = $results[$key];
        }
    }
    parent::__construct ($content, $dbh);
}

public function isValid() {
    if ($this->isEmpty("emp_id") == true) return false;
    if ($this->isEmpty ("expense_date") == true) return false;
    if ($this->isEmpty("description") == true) return false;
    if ($this->isEmpty("amount") == true) return false;
    return true;
}

public function getSqlWhere() {
    return " emp_id = ".$this->emp_id." and
           expense_date ='".$this->expense_date."' and
           description = '".$this->description."'";
}

public function persist() {

    return $this->persistWork (
        $this->contentMetaTable,
        $this->contentMetaOnly);
}
```

В данном случае мы контролируем передаваемую в конструктор информацию. Проверьте выполнение основных тестов для класса `TravelExpenseWeek` и выполните рефакторинг.

Функция записи класса `TravelExpenseWeek` несколько проще, поскольку каждый элемент `TravelExpenseItem` “знает”, как сохранить себя в базе данных. Поэтому необходимо лишь вызвать функцию `persist()` для каждого из этих элементов.

Для этого просто добавьте небольшой цикл в конец функции сохранения класса `TravelExpenseWeek` в файле `/lib/expense.phpm`.

```
public function persist() {
    $this->persistWork ("travel_expense_week",
        array ("emp_id",
            "week_start",
            "comments",
            "territory_worked",
            "cash_advance",
            "mileage_rate"));

    // сохранение каждого элемента в базе данных
    foreach ($this->items as $item) {
        $item->persistent();
    }

    return true;
}
```

Поздравляем! Теперь все тесты пройдены. Осталось добавить некоторые свойства и выполнить быстрый рефакторинг.

Быстрый рефакторинг

Теперь классы `TravelExpenseWeek` и `TravelExpenseItem` обладают схожей функциональностью. Однако в них использована разная степень проверки ошибок. Переместим эту функциональность в класс `PersistableObject`.

По существу, в класс `PersistableObject` необходимо добавить следующий фрагмент.

```
protected $contentMetaTable = null;
protected $contentMetaOnly = null;

public function persist() {
    return $this->persistWork (
        $this->contentMetaTable,
        $this->contentMetaOnly);
}
```

В этот же класс можно также переместить цикл, используемый в обоих конструкторах.

```
foreach ($this->content as $key) {
    if (array_key_exists($key, $results))
        $this->content[$key] = $results[$key];
}
```

Класс `PersistableObject` примет следующий вид.

```
class PersistableObject {
    protected $contentBase = array ();

    protected $contentMetaTable = null;
    protected $contentMetaOnly = null;
```

```
protected $dbh = null; // дескриптор базы данных
protected $dispatchFunctions = array ("role" => "getrole");

function __get ($key) {
    // содержимое удалено для краткости
}

function __construct ($results, $dbh = null) {
    $this->dbh = $dbh;

    if ($this->contentMetaOnly <> null) {
        foreach ($this->contentMetaOnly as $key) {
            if (array_key_exists($key, $results)) {
                $this->contentBase[$key] = $results[$key];
            }
        }
    } elseif ($results <> null) {
        $this->contentBase = $results; // копирование
    }
}

public function implodeQuoted(&$values, $delimiter) {
    // содержимое удалено для краткости
}

public function generateSqlInsert ($tableName, $metas, $values) {
    // содержимое удалено для краткости
}

public function generateSqlInsert ($tableName, $metas, $values) {
    // содержимое удалено для краткости
}

public function generateSqlUpdate ($tableName, $metas, $values) {
    // содержимое удалено для краткости
}

public function generateSqlDelete ($tableName) {
    // содержимое удалено для краткости
}

public function getSqlWhere() {
    // содержимое удалено для краткости
}

protected function isEmpty($key) {
    // содержимое удалено для краткости
}

public function isValid() {
    // содержимое удалено для краткости
}

public function persistWork ($tablename, $meta) {
    // содержимое удалено для краткости
}

public function persist() {
    return $this->persistWork (
```

```

    $this->contentMetaTable,
    $this->contentMetaOnly);

}
}

```

Конструктор отвечает за определение необходимости хранения данных. Обратите также внимание на обратную совместимость этого кода. Если массив contentMetaOnly пуст, то PersistableObject просто копирует массив \$results в элемент contentBase.

При этом класс TravelExpenseItem еще больше упростится.

```

class TravelExpenseItem extends PersistableObject {

    function __construct ($results, $dbh = null) {

        $this->contentMetaTable = "travel_expense_item";
        $this->contentMetaOnly = array (
            "emp_id",
            "expense_date",
            "description",
            "amount");
        parent::__construct ($results, $dbh);
    }

    public function isValid() {
        if ($this->isEmpty("emp_id") == true) return false;
        if ($this->isEmpty("expense_date") == true) return false;
        if ($this->isEmpty("description") == true) return false;
        if ($this->isEmpty("amount") == true) return false;
        return true;
    }

    public function getSqlWhere() {
        return " emp_id = ".$this->emp_id." and
               expense_date = '". $this->expense_date."' and
               description = '". $this->description."'";
    }
}

```

Обратите внимание, что конструктор класса TravelExpenseItem устанавливает состояние соответствующего объекта PersistableObject, а затем передает этому объекту необходимые параметры.

Хотя в данном случае выполняется явное обращение к конструктору, при этом вы можете произвольным образом устанавливать состояние родительского объекта PersistableObject.

Теперь внесем аналогичные модификации в класс TravelExpenseWeek.

```

class TravelExpenseWeek extends PersistableObject {

    public $items = array();

    function __construct ($results, $dbh = null) {

        $this->contentMetaTable = "travel_expense_week";
        $this->contentMetaOnly = array (
            "emp_id",
            "week_start",
            "comments",
            "territory_worked",
            "cash_advance",
            "mileage_rate");
        parent::__construct ($results, $dbh);
    }
}

```

```

}

public function isValid() {
    // содержимое удалено для краткости
}

public function persist() {
    if (parent::persist() == false) return false;

    // сохранение каждого элемента в базе данных
    foreach ($this->items as $item) {
        if ($item->persist() == false) return false;
    }
    return true;
}

public function getSqlWhere() {
    return " emp_id = ".$this->emp_id." and
           week_start = '". $this->week_start."'";
}

public function getExpensesMetaArray () {
    // содержимое удалено для краткости
}

public function getWeekArray() {
    // содержимое удалено для краткости
}

public function addDays($start, $days) {
    // содержимое удалено для краткости
}

public function parse (&$request) {
    // содержимое удалено для краткости
}

public function readWeek() {
    // содержимое удалено для краткости
}

public function getExpenseAmount($offset, $description) {
    // содержимое удалено для краткости
}
}
}

```

Обратите внимание, что в классе TravelExpenseWeek переопределена функция PersistableObject->persist(). В ней сначала вызывается аналогичная функция родительского класса, а затем функция persist() для каждого элемента TravelExpenseItem. Логика добавления и удаления данных из базы реализована в функции PersistableObject->persistWork(). Поэтому в конкретных классах, таких как TravelExpenseWeek, необходимо реализовать лишь функцию getSqlWhere(). Проверим выполнение тестов. Они выполняются, хотя все еще не учитываются авансовые платежи.

```

function testValidTravelExpenseWeek() {
    $tvi = new TravelExpenseWeek (
        array ('emp_id'          => "1",
              'week_start'      => "1980-01-01",
              'comments'        => "comment",

```

```

'mileage_rate'      => "0.31",
'territory_worked' => "Midwest"));
$this->assertEquals(true, $tvi->isValid(), "valid expense");
}

```

Во всех остальных тестах переменной `cash_advance` присваивается значение 0. А что будет, если эту переменную удалить из теста? Он не будет пройден. Дело в том, что в классе `PersistableObject` слепо создается SQL-запрос на основе всех полей базы данных.

Модифицируем тест, исключив из рассмотрения поля `cash_advance` и `comments`.

```

function testTravelExpenseWeekPersistence() {
    $this->_session->getDatabaseHandle()->query(
        "delete FROM travel_expense_week WHERE
            emp_id = 1 and week_start = '1980-01-01'");
    // удаление повторов
    $tvi = new TravelExpenseWeek (
        array ('emp_id'          => "1",
               'week_start'     => "1980-01-01",
               'territory_worked' => "Midwest",
               'mileage_rate'   => "0.31"),
        $this->_session->getDatabaseHandle());
    $result=$this->_session->getDatabaseHandle()->query(
        "select * FROM travel_expense_week WHERE
            emp_id = 1 and week_start = '1980-01-01'");
    $this->assertEquals(0, $result->numRows(), "pre check");
    $this->assertEquals(true, $tvi->persist(), "save");
    $result=$this->_session->getDatabaseHandle()->query(
        "select * FROM travel_expense_week WHERE
            emp_id = 1 and week_start = '1980-01-01'");
    $this->assertEquals (1, $result->numRows(), "persisted ok");
    $row = $result->fetchRow();
    $this->assertEquals(0.0, (float)
        $row['cash_advance'], "cash advance default");
}

```

Значение переменной `cash_advance` можно задать по умолчанию. Для этого просто модифицируем конструктор класса `TravelExpenseWeek`, присвоив ему по умолчанию нулевое значение и пустую строку.

```

class TravelExpenseWeek extends PersistableObject {

    function __construct ($results, $dbh = null) {
        $this->contentMetaTable = "travel_expense_week";
        $this->contentMetaOnly = array ("emp_id",
                                         "week_start",
                                         "comments",
                                         "territory_worked",
                                         "cash_advance",
                                         "mileage_rate");

        $this->contentBase['comments'] = "";
        $this->contentBase['cash_advance'] = "0.0;

    parent::__construct ($results, $dbh);
}

```

Тесты выполняются. Значит, модификации прошли успешно. Просмотрим код на предмет поиска комментариев со строкой `todo`.

```
/**
```

```
* todo: будет ли функция работать при переходе
* на летнее время?
```

```
 */
public function addDays($start, $days) {
    return date("Y-m-d", strtotime($start)+$days*86400);
}
```

Это действительно вопрос. Во многих странах переход на зимнее время осуществляется в последнее воскресенье октября. При этом стрелки переводятся на один час назад, и сутки содержат более чем 86400 секунд. Напишем тест, подтверждающий, что данная функция не будет работать при таких переходах.

```
function testDaylightSavingTime() {
    $tvw = new TravelExpenseWeek (array());
    $this->assertEquals("2004-10-30", $tvw->addDays("2004-10-29", 1));
    $this->assertEquals("2004-10-31", $tvw->addDays("2004-10-29", 2));
    $this->assertEquals("2004-11-01", $tvw->addDays("2004-10-29", 3),
    "no DST");
}
```

Это действительно проблема. Через три дня после 29 октября должно наступить 1 ноября. Поэтому воспользуемся функцией `strtotime()` и откажемся от явного задания числа секунд в сутках.

```
public function addDays($start, $days) {
    return date("Y-m-d", strtotime($start." ".$days." days"));
}
```

Еще один комментарий `todo` связан с оптимизацией функции `getExpenseAmount()`. Но с этим можно подождать.

Итак, в результате напряженной работы мы получили хорошую реализацию объектов для взаимодействия с базой данных. Продолжим реализацию отчетов по командировочным расходам.

Окончательный отчет по командировочным расходам

Несколько часов работы, и вы получите форму, показанную на рис. 22.9.

Воспользуемся шаблоном Smarty в файле `travel-expenses.php`, с помощью которого осуществляется переход из главного меню на первую страницу отчета.

```
<?php
require_once ("lib/common.php");
require_once ("lib/expense.php");

// зарегистрировался ли пользователь?
if (!$session->isLoggedIn()) {
    redirect ("index.php");
}

$user = $session->getUserObject();

$week = new TravelExpenseWeek (
    array ('emp_id'          => $user->id,
          'week_start'      => getCurrentStartWeek(),
```

Widget World

Отчет о командировочных расходах

Имя сотрудника: Ed Lecky-Thompson Отдел: продажи
 Номер: 1 Начальная неделя: 2004-07-04
 Территория: worked

	Вс	Пн	Вт	Ср	Чт	Пт	Сб	Всего
--	----	----	----	----	----	----	----	-------

Проживание

Проживание & Отель	1.00	2.00	3.00					
Другое	2.00							
Чаевые	3.00							
Всего								

Питание

Завтрак	4.00							
Ланч	5.00							
Обед	6.00							
Чаевые	7.00							
Развлечения	8.00							
Всего								

Транспортные расходы

Самолет	9.00							
Аренда авто	10.00							
Бензин	11.00							
Местный проезд	12.00							
Парковка	13.00							
Всего								
Километраж	19.00							
Стоимость/км								
Итого								

Разное

Подарки	14.00							
Телефон & Факс	15.00							
Заказы	16.00							
Почта	17.00							
Другое	18.00							
Всего								

Всего	
Аванс	20.00
По сотруднику	
По компании	

Комментарии:

comment

[Выход из системы](#)

Только для приложения Widget World.

```

'territory_worked' => $_REQUEST["territory_worked"],
'comments'           => $_REQUEST["comments"],
'cash_advance'       => $_REQUEST["cash_advance"],
'mileage_rate'       => $GLOBALS["expense-mileage-travelrate"]),
$session->getDatabaseHandle();

// отображение

if ($_REQUEST["action"] != "persist_expense") {

    $week->readWeek();

    $smarty->assign_by_ref ("user",      $user);
    $smarty->assign_by_ref ("week",      $week);
    $smarty->assign('start_weeks',      getStartWeeks());
    $smarty->assign('current_start_week', getCurrentStartWeek());
    $smarty->assign_by_ref ('expenses',   $week->getExpensesMetaArray());
    $smarty->assign('travelrate',      $GLOBALS["expense-mileage-travelrate"]);
    $smarty->display('travel-expenses.tpl');
    exit();
}

// накопление и сохранение данных за неделю
$week->parse($_REQUEST);
$week->persist();

print "сохранено, спасибо";

?>

Этот сценарий работает в двух режимах: в режиме отображения расходов при чтении их из базы данных (функция readWeek()) и в режиме сохранения данных формы (функция persist()). Выбор режима определяется значением поля action.

Заметим, что все сложные объекты передаются в шаблон Smarty по ссылке, поэтому их временные копии не создаются.

Теперь рассмотрим шаблон Smarty, предназначенный для отображения страницы с данными о командировочных расходах. Его необходимо сохранить в файле templates/travel-expenses.tpl.
```

```

{include file="header.tpl" title="Widget World - Командировочные расходы"}
{literal}
<SCRIPT TYPE="text/javascript">
<!--

function reloadCalc () {
    window.document.forms[0].week_start.value =
        window.document.forms[1].week_start.value // скрытая форма
    window.document.forms[0].submit(); // скрытая форма
}

// -->
</SCRIPT>
{/literal}

<h3>Отчет о командировочных расходах</h3>
```

```

<form method="post">
<input type="hidden" name="action" value="reload_expense">
<input type="hidden" name="week_start" value="">
</form>

<form id="calc" name="calc" action="travel-expenses.php" method="post">

<table border="0" width="100%">
<tr>
<td><b>Имя сотрудника:</b></td>
<td>{$user->first_name} {$user->last_name}</td>
<td><b>Отдел:</b></td><td>{$user-department}</td>
</tr>

<tr>
<td><b>Номер:</b></td>
<td>{$user->id}</td>
<td><b>Начальная неделя:</b></td>
<td><SELECT NAME="week_start" onchange="reloadCalc()">{html_options
values=$start_weeks output=$start_weeks selected=$current_start_week}</SELECT></td>
</tr>

<tr>
<td><b>Территория:</b></td>
<td colspan=3><input name="territory_worked" size=20 maxsize=60
value="{$week->territory_worked}"></td>
</tr>
</table>

<br><br>

```

Первая форма этой страницы аналогична форме отчета о контактах, в котором реализована скрытая форма, предназначенная для заполнения значений начальной недели (на JavaScript). Эта форма автоматически заполняется при выборе новой недели.

Вторая форма просто содержит основную информацию, включая значение `territory_worked`.

Следующий раздел содержит электронную таблицу.

Код просто повторяется для всех дней недели. Напомним, что это лишь первая попытка создания формы, для которой еще потребуется рефакторинг. Но пока ограничимся реализацией базовых элементов без проверки корректности вводимых данных.

```

<table border="0">
<tr><td></td><td>Вс</td><td>Пн</td><td>Вт</td><td>Ср</td><td>Чт</td>
<td>Пт</td><td>Сб</td><td>Всего</td></tr>

```

```
{section name=idx loop=$expenses}{strip}
```

```
<tr><td><b>{$expenses[ idx ].name}</b></td><td></td><td></td><td></td><td></td><td></td>
<td></td><td></td><td></td><td></td></tr>
```

```
{section name=idx2 loop=$expenses[ idx ].data}{strip}
{assign var="p" value=$expenses[ idx ].persist[ idx2 ]}
```

```
<tr bgcolor="#eeeeee, #dddddd">
<td>{$expenses[ idx ].data[ idx2 ]}</td>
```

```
<td><input name="{$expenses[ idx ].code}_sun_{$smarty.section.idx2.index}"
type="text" size="7" maxsize="17" value="{$week->getExpenseAmount(0, $p)}"></td>
```

Напомним, что мы создаем HTML-код вида

```
<input name="lodging_sun_0" type="text" size="7" maxsize="17" value="1.00">
```

Здесь используется два главных цикла по элементам массива расходов.

```
return array(
    array('name' => 'Проживание',
          'code' => 'lodging',
          'data' => array('Проживание & Отель', 'Другое', 'Чаевые'),
          'persist' => array('lodging_and_hotel', 'lodging_laundry', 'lodging_tips'))
```

Переменная Smarty \$p автоматически создается, чтобы обеспечить лучшую читаемость кода. Две строки, соответствующие первому дню недели, копируются шесть раз для каждого из оставшихся дней. К счастью, эти данные не отображаются.

```
<td><input readonly
name="{$expenses[idx].code}_week_sub_{$smarty.section.idx2.index}"
type="text" size="7" maxsize="17"></td>
</tr>
{/strip}{/section}
```

Все хорошо за исключением одного момента. Транспортные расходы должны вычисляться с учетом километража.

Данные о километраже не содержатся в метамассиве расходов, поэтому их необходимо вводить отдельно.

```
{if $expenses[idx].code == 'trans'}
```

```
<tr><td>{$expenses[idx].name} Итого</td>
<td><input readonly name="{$expenses[idx].code}_sun_sub" type="text" size="7"
maxsize="17"></td>
```

Снова скопируем этот неотображаемый код шесть раз.

```
<td><input readonly name="{$expenses[idx].code}_week_sub" type="text" size="7"
maxsize="17"></td></tr>

<tr><td>Километраж</td>
    <td><input name="mitr_sun" type="text" size="7" maxsize="17"
value="{$week->getExpenseAmount(0, 'trans_miles_traveled')}"></td>
```

Для простоты данные остальных дней недели в коде не приведены.

```
<td><input readonly name="mitr_tot" type="text" size="7"
maxsize="17"></td></tr>
<tr><td>Стоимость {$travelrate} / км </td><td><input readonly
name="mitot_sun" type="text" size="7" maxsize="17"></td>
```

Это же касается данных о километраже.

```
<td><input readonly name="mitot_tot" type="text" size="7"
maxsize="17"></td></tr>
<tr><td>{$expenses[idx].name} Всего</td><td><input readonly
name="{$expenses[idx].code}_sun_sub2" type="text" size="7" maxsize="17"></td>
```

```
{else}
```

```
<tr><td>{$expenses[idx].name} Итого</td><td><input readonly
name="{$expenses[idx].code}_sun_sub" type="text" size="7" maxsize="17"></td>
```

Теперь продолжим стандартную обработку отчета. Для начала не будем добавлять дополнительную строку в нижнюю часть раздела о транспортных расходах.

```

<td><input readonly name="${expenses[idx].code}_week_sub" type="text" size="7" maxsize="17"></td></tr>
{/if}

{/strip}{/section}

Остальная часть формы связана с вычислением промежуточных итоговых сумм и вывода комментариев.

<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="3">Всего</td><td><input readonly name="subtotal" type="text" size="7" maxsize="17"></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td colspan="3" rowspan="2">Авансы</td><td><input name="cash_advance" type="text" size="7" maxsize="17" value="${week->cash_advance}"></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td colspan="3">По сотруднику</td><td><input readonly name="totaldueemployee" type="text" size="7" maxsize="17"></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td colspan="3" rowspan="2">По компании</td><td><input readonly name="totalduecompany" type="text" size="7" maxsize="17"></td></tr>
</table>

<br><br>
Комментарии:<br><TEXTAREA NAME="comments" COLS=80 ROWS=6>${week->comments}</TEXTAREA>
<br><br><center>

<input type="submit" name="submit" value=" Сохранить отчет " >
</center>

<input type="hidden" name="action" value="persist_expense">
</form>

{include file="footer.tpl"} 
```

Этот код технически можно использовать в своем приложении, хотя в нем еще не реализована проверка ошибок.

Теперь код можно передавать заказчикам и выслушивать их пожелания.

Реализация отчета о командировочных расходах в виде электронной таблицы

Ваши заказчики остались довольны отчетом. Они его активно используют и вносят дополнительные корректизы.

К счастью для вас, заказчик просит изменить функциональность отчета только в одном аспекте: реализовать его в виде электронной таблицы. Что это означает?

В JavaScript определено одно событие опкейпур, которое можно применить к полям ввода для выполнения вычислений “на лету” при вводе новой информации.

Однако моделировать функции электронной таблицы не так просто, как кажется на первый взгляд. Приведем содержимое обновленного файла template/travel-expenses.tpl.

```

{include file="header.tpl" title="Widget World - Командировочные расходы"}
{literal}
<SCRIPT TYPE="text/javascript">
<!--
function subtotal(thisForm, totalcell, cellArray) {
    var subtot = 0;
    for (var i=0; i < cellArray.length; i++) { 
```

```

if(isNaN(thisForm[cellArray[i]].value))
    thisForm[totalcell].value = 0
else

    subtotal = Math.round(subtot*100 +
        thisForm[cellArray[i]].value*100)/100;

}
thisForm[totalcell].value = subtotal;
return subtotal;
}

```

Параметрами функции `subtotal()` являются форма, ячейка, в которую заносится результат, и массив суммируемых чисел.

Язык JavaScript поддерживает числа с плавающей точкой, но не поддерживает вычисления с произвольной точностью. Поэтому все расчеты следует производить не в долларах, а в центах.

Следующие несколько функций вычисляют итоговые значения столбцов в каждом из разделов.

```

function subday(thisForm, totalcell, prefix, maxindex){

var cellArray = new Array (maxindex);
for (var i=0; i < maxindex; i++) {
    cellArray[i] = thisForm[prefix+i].name;
}

return subtotal(thisForm, totalcell, cellArray);
}

function subweek(thisForm, totalcell, prefix, postfix){

return subtotal (thisForm, totalcell,
    new Array (prefix+'sun'+postfix, prefix+'mon'+postfix,
        prefix+'tue'+postfix, prefix+'wed'+postfix,
        prefix+'thr'+postfix, prefix+'fri'+postfix,
        prefix+'sat'+postfix));
}

function daycalc(thisForm, day, code, thisindex, maxindex) {
    subday (thisForm, code+"_"+day+"_sub", code+"_"+day+"-", maxindex);
    subweek(thisForm, code+"_week_sub_"+thisindex, code+"-", '_'+thisindex) ;
    subday (thisForm, code+"_week_sub", code+"_week_sub_", maxindex);
    totalcalc (thisForm);

    return true;
}

```

Функция `daycalc()` вызывается для каждой из ячеек HTML-формы при вводе информации пользователем. Вызов осуществляется следующим образом.

```
onkeyup="return daycalc(this.form, 'sun', 'lodging', '0', 3)"
```

Напомним, что данная форма заполняется информацией из метамассива расходов.

```
array('meals_breakfast', 'meals_lunch',
      'meals_dinner', 'meals_tips',
      'meals_entertainment')
```

Однако JavaScript “не знает” об использовании PHP или шаблона Smarty. Поэтому интерпретатору JavaScript нужно явно указать, с какой ячейкой мы работаем и данные каких ячеек суммируются. Эта информация передается через два последних параметра функции `daycalc(): '0'` означает текущую строку, а `3` — максимальное число суммируемых строк.

Километраж необходимо вычислять отдельно на основе полей `mitr_sun`, `mitr_mon` и т.д. Промежуточные результаты, представляющие собой произведения количества километров на стоимость одного километра, хранятся в полях `mitot_sun`, `mitot_mon` и т.д. При этом с помощью функции `subweek()` сначала вычисляются промежуточные суммы по горизонтали, а затем вычисляется общий итоговый результат по вертикали. На этом обработка раздела транспортных расходов завершается.

```
function micalc (thisForm, day, travelrate) {
    var totalcell = 'mitot_'+day;
    var sourcecell = 'mitr_'+day;

    // километраж
    thisForm[totalcell].value = Math.round(
        thisForm[sourcecell].value*100*travelrate)/100;

    subweek (thisForm, 'mitr_tot', 'mitr_', '');
    subweek (thisForm, 'mitot_tot', 'mitot_', '');

    // расходы на транспорт за день
    thisForm["trans_"+day+"_sub2"].value = Math.round(
        (thisForm[totalcell].value * 100) +
        (thisForm["trans_"+day+"_sub"].value * 100))/100;

    // сумма за неделю
    subweek (thisForm, "trans_week_sub2", "trans_", "_sub2");
    totalcalc (thisForm);
}
```

Теперь займемся проверкой ошибок.

```
function checkTransInput (thisForm, day) {
    if (thisForm["trans_"+day+"_sub"].value > 0 &&
        thisForm["mitr_"+day].value == "") {
        alert ("Пожалуйста, введите километраж за "+day);
        return false;
    }
    return true;
}

function checkInputs(thisForm) {
    if (checkTransInput (thisForm, "sun") == false) {
        return false;
    }
    if (checkTransInput (thisForm, "mon") == false) {
        return false;
    }
    if (checkTransInput (thisForm, "tue") == false) {
        return false;
    }
    if (checkTransInput (thisForm, "wed") == false) {
        return false;
    }
    if (checkTransInput (thisForm, "thr") == false) {
        return false;
    }
}
```

```

    }
    if ( checkTransInput (thisForm, "fri") == false) {
        return false;
    }
    if ( checkTransInput (thisForm, "sat") == false) {
        return false;
    }
    if ( thisForm["subtotal"].value == 0) {
        alert ("Пожалуйста, введите данные.");
        return false;
    }
    if (thisForm["territory_worked"].value == "") {
        alert ("Пожалуйста, введите территорию.");
        return false;
    }
    return true;
}

```

Далее следует арифметически сложная функция `totalcalc()`, отвечающая за пересчет всей формы.

```
//
```

```
// Не существует простого способа передачи массива PHP
// в код JavaScript. Поэтому сгенерируем его вручную.
```

```

//
function totalcalc (thisForm) {
    var sectionArray = new Array ('lodging', 'meals', 'trans', 'misc');
    var subtotal = 0;
    for (var i=0; i < sectionArray.length; i++) {
        subtotal = subtotal +
            Math.round(thisForm[sectionArray[i]+
                "_week_sub"].value*10_0)/100;
    }
    subtotal = subtotal + Math.round(thisForm["mitot_tot"].value*100)/100;
    thisForm["subtotal"].value = subtotal;
    var total = subtotal - Math.round(thisForm["cash_advance"].value*100)/100;
    total = Math.round(total*100)/100;
    if (total >= 0) {
        thisForm["totaldueemployee"].value = total;
        thisForm["totalduecompany"].value = "";
    } else {
        thisForm["totaldueemployee"].value = "";
        thisForm["totalduecompany"].value =
            Math.round(total * 100/100 * (-1));
    }
}

```

Функция `recalculate()` связывается с кнопкой пересчета. Эта функция обеспечивает пересчет значений всех вычисляемых полей. Она не является примером для подражания. Гораздо удобнее использовать один и тот же код для всех дней недели, а не двойной цикл (по строкам и по столбцам), который затрудняет понимание.

```

// не слишком критикуйте эту функцию;
// цикл затрудняет понимание
function recalculate (thisForm, mileage) {

    daycalc(thisForm, 'sun', 'lodging', '0', 3);

    // Данные для остальных дней недели не приводятся
    // для ясности.

    daycalc(thisForm, 'sat', 'lodging', '0', 3);
    daycalc(thisForm, 'sat', 'lodging', '1', 3);

```

```

daycalc(thisForm, 'sat', 'lodging', '2', 3);
daycalc(thisForm, 'sun', 'meals', '0', 5);

// Данные для остальных дней недели не приводятся
// для ясности.

daycalc(thisForm, 'sat', 'meals', '0', 5);
daycalc(thisForm, 'sat', 'meals', '1', 5);
daycalc(thisForm, 'sat', 'meals', '2', 5);
daycalc(thisForm, 'sat', 'meals', '3', 5);
daycalc(thisForm, 'sat', 'meals', '4', 5);

daycalc(thisForm, 'sun', 'trans', '0', 5);

// Данные для остальных дней недели не приводятся
// для ясности.

daycalc(thisForm, 'sat', 'trans', '0', 5);
daycalc(thisForm, 'sat', 'trans', '1', 5);
daycalc(thisForm, 'sat', 'trans', '2', 5);
daycalc(thisForm, 'sat', 'trans', '3', 5);
daycalc(thisForm, 'sat', 'trans', '4', 5);

daycalc(thisForm, 'sun', 'misc', '0', 5);

// Данные для остальных дней недели не приводятся
// для ясности.

daycalc(thisForm, 'sat', 'misc', '0', 5);
daycalc(thisForm, 'sat', 'misc', '1', 5);
daycalc(thisForm, 'sat', 'misc', '2', 5);
daycalc(thisForm, 'sat', 'misc', '3', 5);
daycalc(thisForm, 'sat', 'misc', '4', 5);

micalc(thisForm, 'sun', mileage);

// Данные для остальных дней недели не приводятся
// для ясности.

micalc(thisForm, 'sat', mileage);
return (totalcalc(thisForm));
}

function reloadCalc () {
    window.document.forms [0].week_start.value =
        window.document.forms[1].week_start.value // скрытая форма
    window.document.forms[0].submit(); // скрытая форма
}

// -->
</SCRIPT>
{/literal}

```

Теперь, когда реализована функция вычисления сумм по строкам и столбцам, каждая ячейка должна реагировать на событие onkeyup вызовом функции daycalc(). Модифицируем соответствующий фрагмент в файле templates/travel-expenses.tpl.

```

<input name="{$expenses[idx].code}_sun_{$smarty.section.idx2.index}" onkeyup="return daycalc(this.form, 'sun', '{$expenses[idx].code}', '{$smarty.section.idx2.index}', {$smarty.section.idx2.total})" type="text" size="7" maxsize="17" value="{$week->getExpenseAmount(0, $p)}">

```

Аналогичное изменение внесем в фрагмент шаблона, отвечающий за вычисление километража.

```
<input name="mitr_sun" onkeyup="return micalc(this.form, 'sun',
{$travel_rate})" ...
```

Добавим кнопку пересчета.

```
<br><br>
<center>

<input type="button" value=" Пересчитать " onclick="return
recalculate(this.form, {$travelrate})">

</center>
<br><br>
Комментарии:<br><TEXTAREA NAME="comments" COLS=80 ROWS=6>{$week->comments}
</TEXTAREA>
<br><br><center>
```

Теперь при щелчке на кнопке передачи данных проверяется наличие ошибок ввода с помощью функции `checkInputs()` по событию `onclick`.

```
<input type="submit" name="submit" value=" Сохранить отчет " onclick="return
checkInputs(this.form); ">
```

Практически все готово.

Осталось несколько моментов. Например, при изменении недели форма перезагружается и значения ее полей вычисляются заново. Однако это событие не активизируется при первой загрузке формы.

Поэтому необходимо обеспечить вызов функции JavaScript `recalculate()` при загрузке формы. Это можно сделать в обработчике события `onload` в дескрипторе `<body>`. Этот дескриптор определен в файле `header.tpl`. Поэтому в файле `travel-expenses.php` модифицируем раздел отображения.

```
// отображение
if ($_REQUEST["action"] != "persist_expense") {
    $week->readWeek();

    $smarty->assign_by_ref ("user",           $user);
    $smarty->assign_by_ref ("week",           $week);
    $smarty->assign('start_weeks',           getStartWeeks());
    $smarty->assign('current_start_week',   getCurrentStartWeek());
    $smarty->assign_by_ref ('expenses',        $week->getExpensesMetaArray());
    $smarty->assign('travelrate',           $GLOBALS["expense-mileage-travelrate"]);

    $smarty->assign('formfunc',
        "recalculate(window.document.forms[1],".
        "$GLOBALS['expense-mileage-travelrate']).");
}

$smarty->display('travel-expenses.tpl');
exit();
}
```

Теперь внесем соответствующие изменения в файл `header.tpl`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta HTTP-EQUIV="content-type" CONTENT="text/html; charset=ISO-8859-1">
<title>{$title|default:"no title"}</title>
</head>
```

```
<body onload="{$formfunc|default:'"}>
```

```
<h1>Widget World</h1>
<hr><p>
```

Переменная Smarty \$formfunc содержит имя функции JavaScript, выполняемой при загрузке формы HTML. Свойство формы onload принимает значение \$formfunc или содержит пустую строку, если имя функции не определено. Это позволяет пересчитывать значение полей электронной таблицы в процессе загрузки формы. В этом случае функция daycalc() отвечает только за вычисление итоговых значений по строкам и столбцам, а не за пересчет всех полей формы.

Заключительные штрихи

Заказчики сообщили, что километраж, вероятно, некорректно сохраняется в базе данных. Дело в том, что эта информация не заносится в метамассив, а обрабатывается отдельно.

Пришло время разобраться с данной проблемой. Напишем соответствующий тест.

```
function testParsingNbrMiles() {
    $this->_session->getDatabaseHandle()->query(
        "delete FROM travel_expense_item WHERE
         emp_id = 1 and expense_date >= '1980-01-06'
         and expense_date <= '2001-09-15')";

    $response = array ('mitr_sun' => "1.1",
                      'mitr_mon' => "2.2");

    $week = new TravelExpenseWeek (
        array ('emp_id'          => "1",
              'week_start'      =>"1980-01-06",
              'territory_worked' => "Midwest",
              'comments'        => "comment",
              'cash_advance'   => "0",
              'mileage_rate'   => "0.31"),
        $this->_session->getDatabaseHandle());

    $week->parse($response);
    $this->assertEquals(true, $week->persist());

    $week = new TravelExpenseWeek (
        array ('emp_id'          => "1",
              'week_start'      => "1980-01-06",
              'territory_worked' => "Midwest",
              'comments'        => "comment",
              'cash_advance'   => "0",
              'mileage_rate'   => "0.31"),
        $this->_session->getDatabaseHandle());

    $week->readWeek();

    $this->assertEquals(1.1, (float)
        $week->getExpenseAmount(0, 'trans_miles_traveled'));
    $this->assertEquals(2.2, (float)
        $week->getExpenseAmount(1, 'trans_miles_traveled'));

}
```

Внесем соответствующие изменения в описание класса TravelExpenseWeek в файле lib/expense.phpm.

```

private function createByRequest ($description, $daynum,
                                $index, &$request) {
    if (array_key_exists($index, $request) and
        $request[$index] <> null and
        $request[$index] <> "") {
        array_push (
            $this->items,
            new TravelExpenseItem (
                array ('emp_id' => $this->emp_id,
                      'expense_date' =>
                        $this->addDays($this->week_start, $daynum),
                      'description' => $description,
                      'amount' => (float)
                        $request[$index]),
                $this->dbh));
    }
}

/**
 * Эта функция устраняет противоречия между ежедневными
 * данными из базы и еженедельным их отображением
 */
public function parse(&$request) {

    foreach ($this->getExpensesMetaArray() as $sectionlist) {

        for ($i=0; $i < count ($sectionlist['persist']); $i++) {
            $daynum = 0;
            foreach ($this->getWeekArray() as $day) {
                $index = $sectionlist['code']."_".$day."_".$i;

                $this->createByRequest ($sectionlist['persist'][$i],
                                        $daynum, $index, $request);

            }
            $daynum++;
        }
    }
    // исключение
    $daynum = 0;

    foreach ($this->getWeekArray() as $day) {
        $this->createByRequest ("trans_miles_traveled",
                               $daynum, "mitr_". $day, $request);

    }
}

```

Новая функция TravelExpenseWeek->createByRequest() позволяет избежать дублирования кода, поскольку она вызывается как для обработки обычных полей, так и для обработки полей километража.

Все тесты пройдены, заказчики довольны, форма работает. На этом завершается реализация сценария 4. Результаты его выполнения показаны на рис. 22.10.

Widget World

Отчет о командировочных расходах

Имя сотрудника:	Ed Lecky-Thompson	Отдел:	продажи
Номер:	1	Начальная неделя:	2004-07-04
Территория:	worked		

	Вс	Пн	Вт	Ср	Чт	Пт	Сб	Всего
Проживание								
Проживание & Отель	1.00	2.00	3.00					6
Другое	2.00							2
Чаевые	3.00							3
Всего	6	2	3	0	0	0	0	11
Питание								
Завтрак	4.00							4
Ланч	5.00							5
Обед	6.00							6
Чаевые	7.00							7
Развлечения	8.00							8
Всего	30	0	0	0	0	0	0	30
Транспортные расходы								
Самолет	9.00							9
Аренда авто	10.00							10
Бензин	11.00							11
Местный проезд	12.00							12
Парковка	13.00							13
Всего	55	0	0	0	0	0	0	55
Километраж	19.00							19
Стоимость/км	6.84	0	0	0	0	0	0	6.84
Итого	61.84	0	0	0	0	0	0	61.84
Разное								
Подарки	14.00							14
Телефон & Факс	15.00							15
Заказы	16.00							16
Почта	17.00							17
Другое	18.00							18
Всего	80	0	0	0	0	0	0	80
								182.84
Всего								20.00
Аванс								162.84
По сотруднику								
По компании								

Комментарии:
comments

[Выход из системы](#)

Только для приложения Widget World.

Рис. 22.10.

Полученная система не просто работает. Она хорошо оттестирована и может составить основу каркаса.

На этом примере вы ознакомились с реализацией взаимодействия между PHP, JavaScript и браузером. Каждый из этих элементов отвечает за обработку данных на своем уровне. Поэтому сложное поведение электронной таблицы может быть реализовано с помощью относительно небольших объемов кода.

Куда двигаться дальше? Продолжать реализацию сценариев? Продолжайте чтение этой главы, поскольку в ее последнем разделе описан еще один важный вопрос.

Объекты-имитаторы

Иногда в процессе тестирования приходится взаимодействовать с детерминированными объектами, к которым по разным причинам нет доступа в среде тестирования. Например:

- реальный объект является недетерминированным, например, курс доллара или погодные условия;
- реальный объект изменяется слишком медленно, поэтому его нельзя использовать при тестировании;
- необходимо протестировать исключительные ситуации, например, вторжение в систему;
- реальный объект не существует, например, к нему нет доступа или он не обладает необходимой функциональностью.

В подобных ситуациях используют объекты-имитаторы, обладающие необходимой функциональностью для тестирования системы.

Например, если вы хотите протестировать работу сессий, работоспособность системы при возникновении ошибок или при попытках взлома, вы можете импортировать файл `mock-widgetsession.php`, в котором содержатся следующие классы.

```
class MockWidgetSession extends WidgetSession {
    public function getUserObject() {
        return new WidgetUser(
            array (
                'id'          => 1,
                'username'    => 'ed',
                'md5_pw'      => "827ccb0eea8a706c4c34a16891f84e7b",
                'first_name'  => "Ed",
                'last_name'   => "Lecky-Thompson",
                'email'       => "ed@lecky-thompson.com",
                'role'        => "s",
                'department'  => "sales"));
    }
}
```

Этот класс позволяет моделировать регистрацию пользователя в системе в процессе тестирования.

Шаблон Smarty можно использовать для имитации форм HTML. Их можно не выводить на экран, а сохранять в соответствующей переменной.

Например, в сценарии 6 предполагается уведомление менеджеров по продажам. При сохранении контактов менеджерам необходимо отправить электронное сообщение. На первый взгляд, все довольно просто. Однако на деле в электронное сообщение придется включать данные HTML-формы.

Получение данных из формы не составляет проблемы. По существу, нам необходимо протестировать отображаемую в браузере информацию. До сих пор этого делать не приходилось. В следующем тесте используется вывод HTML-кода на основе регулярных выражений.

```
function testContactEmail () {
    $u = new WidgetUser(
        array ('id'          => 1,
              'username'     => "ed",
              'first_name'   => "Ed",
              'last_name'    => "Lecky-Thompson",
              'email'        => "ed@lecky-thompson.com",
              'role'         => "S",
              'department'   -> "sales"));
    $cv = new ContactVisit (
        array ('emp_id'      => "1",
              'week_start'  => "1980-01-01",
              'company_name'=> "test one",
              'contact_name'=> "Big One",
              'city'         => "Columbus",
              'state'        => "OH",
              'accomplishments'=> "phone call",
              'followup'     => "",
              'literature_request'=> ""));
    $c = new Contact (
        array ("emp_id"      => "1",
              "week_start"  => "1980-01-01",
              "shop_calls"   => 2,
              "distributor_calls"=> 3,
              "engineer_calls"=> 4,
              "mileage"      => 50,
              "territory_worked"=> "Central Ohio",
              "territory_comments"=> "Buckeyes are great." ),
    $this->_session->getDatabaseHandle());
}

list ($email, $_from, $subject, $message, $headers) =
    generateContactEmail($u, $c array ($cv), false);

// захват экрана

$this->assertEquals (1, preg_match ("/Имя сотрудника.....Ed
Lecky-Thompson/", $message), "employee name");

$this->assertEquals(1, preg_match
("/company_name_0.....test one/", $message), "company name");
$this->assertEquals(1, preg_match
("/shop_calls .....2/", $message), "shop calls");
$this->assertEquals(1, preg_match ("/To: ed@lecky-thompson.com/", $headers), "email");

// тестирование кодировки base64
list ($email_from, $subject, $message, $headers) =
    generateContactEmail($u, $c, array ($cv));

$this->assertEquals (0, preg_match ("/Имя сотрудника.....Ed
Lecky-Thompson/", $message), "employee name");

$this->assertEquals(1, preg_match ("/To: ed@lecky-thompson.com/", $headers), "email");
}
```

Функция `generateContactEmail()` возвращает информацию, необходимую PHP-функции `mail()`.

В этом примере мы тестируем выводимую в браузере информацию. При этом можно проверять не весь код HTML, а лишь удостовериться, что форма содержит необходимую информацию. Так, простое регулярное выражение

```
"Имя сотрудника.....Ed Lecky-Thompson/"
```

соответствует следующему коду HTML.

```
<tr><td><b>Имя сотрудника:</b></td><td>Ed Lecky-Thompson</td>
<td><b>Отдел:</b></td><td>sales</td></tr>
```

Поскольку кодировка Base64 неудобочитаема для человека, в заголовке электронного сообщения проверяется только наличие электронного адреса. Наличие имени Имя сотрудника не проверяется, поскольку в кодировке Base64 его нельзя прочитать обычным образом.

В файл `lib/common-functions.php` можно поместить следующие функции.

```
function mimeifyContent ($content, $mime_boundary,
                        $filename, $flagBase64=true) {
    $message = "";
    $message .= "\r\n";
    $message .= "--".$mime_boundary."\r\n";
    $message .= "Content-Type: text/html;\r\n";
    $message .= " name=\"".$filename.".html\"\r\n";
    // по умолчанию семибитовый ascii-код
    if ($flagBase64) {
        $message .= "Content-Transfer-Encoding: base64\r\n";
    }
    $message .= "Content-Disposition: attachment;\r\n";
    $message .= " filename=\"".$filename.".html\"\r\n";
    $message .= "\r\n";
    if ($flagBase64) {
        $message .= base64_encode($content);
    } else {
        $message .= $content;
    }
    $message .= "\r\n";
    return ($message);
}

function generateContactEmail (&$user, &$contact,
                               $contactVisits, $flagBase64=true) {
    global $GLOBALS;
    require_once ($GLOBALS["smarty-path"] . 'Smarty.class.php');
    $smarty = new Smarty;
    $smarty->assign_by_ref ("user", $user);
```

```

$smarty->assign_by_ref ("contact", $contact);
$smarty->assign_by_ref ("contactVisits", $contactVisits);
$smarty->assign('start_weeks', getStartWeeks());
$smarty->assign('current_start_week', $contact->week_start);
$smarty->assign("max_weekly_contacts", $GLOBALS["max-weekly-contacts"]);

$email_body = @$smarty->fetch('customer-contacts.tpl');

$headers = "";
$headers .= "From: ".$GLOBALS["email-from"]."\n";
$headers .= "To: ".$user->email."\n";
if (strlen ($GLOBALS["email-contact-cc"])) > 0)
    $headers .= "Cc: ";
    $headers .= $GLOBALS["email-contact-cc"];
$headers .= "\n";
if (strlen ($GLOBALS["email-contact-bcc"])) > 0)
    $headers .= "Bcc: ".$GLOBALS["email-contact-bcc"]."\n";

$mime_boundary = "<<<----+X[".md5(time())."]";
$headers .= "MIME-Version: 1.0\r\n";
$headers .= "Content-Type: multipart/mixed;\r\n";
$headers .= " boundary=". "$mime_boundary." "\n";

$message = "";
$message .= "Это сообщение в MIME-формате из нескольких частей.\r\n";
$message .= "\r\n";
$message .= "--".$mime_boundary."\r\n";
$message .= "Content-Type: text/plain; charset=\"iso-8859-1\"\r\n";
$message .= "Content-Transfer-Encoding: 7bit\r\n";
$message .= "\r\n";
$message .= $GLOBALS["email-contact-message"]."\n\n";
$nextEnding = "\r\n";
$message.=mimeifyContent($email_body,$mime_boundary,
                        "customer-contact", $flagBase64);

$message.=--".$mime_boundary.".$nextEnding;
$subject = $user->emp_id." ".$user->last_name." : ".$user->email_subject;
return array ($user->email, $subject, $message, $headers);
}

```

Следует отметить два момента. Вместо вызова функции `$smarty->display()` используется вызов `$smarty->fetch()`. Функция `fetch()` записывает код HTML в переменную, а не выводит на экран, как функция `display()`.

Кроме того, по техническим причинам безопаснее использовать кодировку Base64 для символов Unicode. Однако кодировка Base64 существенно затрудняет тестирование электронных сообщений, поэтому вводится параметр `$flagBase64`.

Если бы в этой функции электронное сообщение на самом деле отправлялось, а не просто создавалось в целях тестирования, пришлось бы реализовать объект-имитатор, эмулирующий API-интерфейс электронной почты.

Резюме

Итак, разработка программной системы завершена. Хотя реализованы не все описанные в начале главы сценарии, авторы не станут нагружать читателя сотнями страниц описания аналогичных процедур. Теперь у вас есть необходимые знания и средства для их самостоятельной реализации.

При описании разработки программных систем тестированию исторически уделяется слишком мало внимания. Однако на примере данной главы читатель получил опыт разработки на основе тестирования и научился отлавливать ошибки в процессе разработки системы.

Рефакторинг — это не просто “чистка кода”. Это отдельная задача, которой необходимо уделить много внимания в процессе разработки. На примере данной главы вы удостоверились, что в процессе рефакторинга можно создавать сложные каркасы.

Прозрачность и гибкость кода достигается за счет применения принципов экстремального программирования. Конечно, эта методология применима не для всех проектов, однако такие приемы как раннее тестирование и рефакторинг окажутся полезными и при использовании других процессов. В этой главе рассмотрены не все аспекты экстремального программирования. Это целая методология, которую следует изучать отдельно.

23

Обеспечение качества

Ваш проект по автоматизации торговли продвигается весьма успешно. Все идет по графику в рамках запланированного бюджета. Заказчик доволен бета-версией, работающей на вашем сервере разработки. Поскольку в процессе разработки постоянно проводилось модульное тестирование, вы абсолютно уверены в работе отдельных компонентов и в том, что при их объединении не столкнетесь с серьезными проблемами.

Однако не следует радоваться преждевременно. На самом деле вы достигли самой критической стадии всего проекта. Приложение в основном завершено, и заказчику не терпится получить его в свое распоряжение. Здесь нужно быть очень внимательным.

Термин “обеспечение качества” (quality assurance) должен быть вам знаком. Он позаимствован из традиционных производственных процессов и обозначает процедуру, выполняемую, например, перед продажей нового DVD-плеяера или дорогого семейного седана. Почему бы вам не применить эту процедуру при разработке сложного программного продукта?

В этой краткой, но важной главе вы познакомитесь с процессом обеспечения качества для больших PHP-проектов. Вы узнаете, почему качество играет столь важную роль и что означает этот термин для вас и вашего заказчика. Здесь будет рассказано о видах тестирования, которые необходимо выполнить для обеспечения качества приложения, а также о том, как исправить возможные недочеты, выявленные в процессе этого тестирования.

Основы анализа качества

Очевидно, все пользователи рассчитывают на надежность Web и Интернет. Однако бывают случаи, когда электронные сообщения не доходят до получателя или приложение электронной торговли зависает в процессе обработки номера вашей кредитной карточки. Иногда сервер провайдера не работает в течение нескольких дней, и никакие технические специалисты не могут в этом помочь.

Подобные ситуации встречаются довольно часто. Но самое ужасное, что все к этому готовы. Получатель электронной почты не удивляется тому, что сообщение ему не дошло.

У вас не перехватывает дыхания, если заказ через Интернет остается невыполненным. А клиенты провайдера пожимают плечами и идут искать ближайшее Интернет-кафе.

Такая ситуация характерна только для Интернет. Если вы купили новую машину, а она стала барахлить уже через несколько недель, вы в ярости обратитесь к своему дилеру. Если в кафе вы заказали “еспрессо”, а вам принесли “капучино”, то вы заставите официанта исправить ошибку. А если ваш DVD-плеер не работает, вы вернете его в магазин. Сбои в Интернете не удивляют по некоторым причинам. Во-первых, Интернет — слишком сложный организм, в котором могут происходить ошибки. Однако Интернет не сложнее DVD-плеяра. Действительно, инфраструктура Интернет (серверы, маршрутизаторы и другое оборудование) работает достаточно надежно. Вся проблема в плохих программах, которые становятся причиной ошибок.

Вторая и более серьезная причина связана с теми, кто уже имел опыт общения с разработчиками приложений для Интернета. Зачастую предыдущий опыт общения с подобными агентствами и специалистами не вселяет никакого оптимизма. Вряд ли в какой-либо другой области можно встретить так же много неквалифицированных специалистов, которые ринулись в область разработки Web-приложений за последние несколько лет. Это касается множества “профессиональных” специалистов по аутсорсингу, а также многих специализированных агентств. Хорошее качество работы в этой области является исключением, а не правилом. Поэтому большинство таких специалистов больше говорят, чем делают.

Отсюда можно сделать циничный вывод: поскольку ожидания заказчика достаточно невысоки, он и получает очень низкий результат. Исполнитель знает, что в случае ошибки его не ждет серьезное наказание. Однако этот подход является не только циничным, но и коммерчески обоснованным.

Почему нужно ставить высокие цели

Высокое качество продукта необходимо не только для удовлетворения желаний заказчика. В первую очередь, это стимул для самих разработчиков. Речь не идет о функциональных дефектах системы. Некачественное приложение может работать слишком медленно или содержать скрытую логическую ошибку. Систематический подход к обеспечению качества подразумевает следующее.

- Время разработки, значит, и стоимость устранения дефектов, минимизируется благодаря использованию системного подхода.
- Проблемы, возникающие в процессе развертывания приложения, не связаны с самим приложением.
- График выполнения проекта строго соблюдается.

Во многом проблемы обеспечения качества определяются отношением к работе. Сравните два подхода.

- “Ожидания клиента очень низки, поэтому можно что-то и не доделать.”
- “Ожидания клиента очень низки, поэтому если я превзойду их, то заслужу поощрение.”

Нечего и говорить, что первый ход мыслей не может привести к успеху. Возможно, предыдущий опыт заказчика не вселяет большого оптимизма, но почему бы не попытаться что-нибудь изменить?

Высокие цели означают стремление к высокому качеству, которого можно достичь, только точно зная, что понимается под качеством.

Что такое качество

Термин “качество” в ракурсе компьютерных приложений имеет несколько необычный смысл. Вернемся к аналогии с семейным седаном. Описывая высокое качество автомобиля, журналист, скорее всего, будет говорить не о надежности или производительности, а о более количественных (потребительских) факторах, таких как:

- надежность обивки;
- использование дорогих материалов, в том числе хромированных деталей;
- эстетические моменты;
- бесшумность закрытия двери.

Качество не определяется одним общим фактором — это производная от всех факторов, в которых что-то может не устраивать потребителя, но устраивает.

Такой подход выработан журналистами, оценивающими новую марку автомобиля на выставке. Для определения уровня качества товара существует набор простых тестов, предполагающих многократное выполнение одних и тех же действий (например, открытия дверей или нажатия кнопки на панели). Однако эти тесты не позволяют определить потребление топлива или надежность автомобиля. Почему же они так важны?

Истина состоит в том, что книгу нельзя оценить по ее обложке, но автомобиль можно оценить по его интерьеру. Автомобиль с мягко закрывающейся дверью, удобными хромированными переключателями почти всегда показывает себя надежным средством передвижения. Самые высококачественные бренды — BMW, Audi и Mercedes — так же хорошо зарекомендовали себя на дорогах. Это же касается и Web-приложений. Качество — это не только бесперебойная работа Web-узла или его соответствие функциональной спецификации. Это более тонкие вопросы. Если в порядке мелкие, лежащие на поверхности детали, значит, и в остальном данный проект будет соответствовать определенному уровню.

Но хватит обсуждать автомобили! Давайте обсудим, что понимается под высоким качеством Web-приложения и как его оценить.

Мера качества

Говоря об обеспечении качества приложения, необходимо затронуть множество различных областей. Степень важности каждой из них во многом определяется природой и целевой аудиторией данного приложения.

Функциональная согласованность

На самом базовом уровне приложение должно обеспечивать всю функциональность, описанную в спецификации проекта.

Речь идет не только об отсутствии ошибок и предупреждений РНР. Поведение приложения должно быть корректным и проверенным в самых разнообразных тестовых условиях. К ним относятся следующие.

- При нажатии кнопки обновления база данных должна действительно обновляться.
- Если новому пользователю должно отправляться приветственное сообщение, оно должно действительно корректно отправляться.
- Если пользователь выбирает функцию поиска, то возвращаемые данные должны точно удовлетворять заданным критериям.

Такая согласованность может показаться очевидной, но именно очевидные вещи проверяются программами проверки качества.

Работоспособность в реальных условиях

Приложение должно надежно функционировать в условиях, приближенным к реальным, а не только в среде разработки.

Речь идет не только о пользователях или данных, но и о требованиях корректной работы системы в случае некорректной работы пользователей (например, при вводе данных в неправильном формате). Корректным поведением в такой ситуации является вывод сообщения об ошибке. Чрезвычайно важно промоделировать подобные условия, поскольку пользователь время от времени будет допускать ошибки при работе с системой.

Работоспособность при сбоях

Зачастую при разработке учитывают не все сценарии, а некоторые из них могут оказаться весьма нежелательными.

Чрезвычайно важно предусмотреть любой возможный ход событий. Конечно плохо, если процесс обработки кредитной карточки пользователя завершился неудачно, но еще хуже, если система не проинформирует об этом пользователя.

Система должна вести себя корректно при любых сбоях. Например, если сервер баз данных недоступен, приложение не сможет функционировать корректно. Этую проблему может решить только системный администратор. Однако в этом случае приложение может просто не выводить результатов поиска или сообщить пользователю о некорректности введенного пароля. Это означает, что в приложении не предусмотрен отказ сервера базы данных, и оно вводит пользователя в заблуждение. Гораздо корректнее просто вывести сообщение: “Извините, в данный момент сервер недоступен.”

Восприимчивость к нагрузке

При функциональном тестировании нагрузка на приложение не слишком велика. Его одновременно использует не более десяти человек.

Однако на практике число пользователей может оказаться существенно больше. Поэтому приложение должно поддерживать сотни, а то и тысячи одновременных обращений.

Эта цифра обычно определяется в технической спецификации и принимается во внимание при разработке архитектуры приложения. Но то, что хорошо на бумаге, не всегда работает на практике. А данный вопрос является жизненно важным. Если время отклика приложения слишком велико или сервер вообще не способен справиться с большим количеством поступающих запросов, заказчик вряд ли будет считать работу выполненной.

Удобство использования

В современном мире удобству использования компьютерных систем уделяется очень большое внимание. Признанные эксперты в этой области, в частности Якоб Нильсен (Jakob Nielsen) (посмотрите на его замечательный Web-узел по адресу www.useit.com), удобство использования связывают с интуитивностью и логичностью интерфейса пользователя. Эту характеристику можно применять не только к Web, но и к традиционным программным продуктам, и даже к панели управления стиральной машины.

Удобство использования измеряется простотой и скоростью, с которой новый пользователь приложения может выполнить определенные задачи. Эти задачи формулируются как цели, для достижения которых не приводится никаких инструкций. Например, пользователям почтового Web-приложения можно попросить создать

новое сообщение и отправить его по заданному адресу, не давая при этом никаких дополнительных рекомендаций.

Эффективность выполнения этой задачи пользователем и определяет удобство использования системы.

Внешний вид

Близким к удобству использования является критерий внешнего вида приложения. Вспомним аналогию с хромированными ручками автомобиля. Для программного приложения внешний вид интерфейса пользователя является прямым эквивалентом, позволяющим судить о внутреннем качестве приложения.

Что мы понимаем под внешним видом? Вспомним простые примеры, приведенные в этой книге, не связанные с крупным проектом автоматизации торговли. Они приведены здесь только для демонстрации определенных понятий. Поскольку эти примеры призваны лишь продемонстрировать технические решения, они выглядят не слишком привлекательными. Конечный продукт ни в коем случае не должен подчиняться этому подходу.

Хромированные ручки автомобиля аналогичны кнопкам перехода на следующую или предыдущую страницу. Мягкость закрытия двери — это аналог обновления страниц приложения. Эти параметры определяют внешний вид приложения и его общее качество.

Многие разработчики программных систем слишком мало внимания уделяют эстетике интерфейса пользователя. Поэтому при реализации собственного проекта очень важно иметь опытного дизайнера интерфейса, который может придать вашему продукту внешний лоск.

На завершающих стадиях проекта необходимо выделить время и обеспечить сглаживание всех “острых углов” приложения. При этом необходимо придерживаться единого стиля оформления приложения. В противном случае заказчик может не очень обрадоваться появлению в самом неожиданном месте прямоугольной серой кнопки Отправить.

Тестирование

Единственным способом обеспечения качества проекта является его тестирование, тестирование и еще раз тестирование.

Однако тестирование может принимать множество форм. Для тестирования каждой из описанных выше метрик существуют свои способы. Рассмотрим их более подробно.

Модульное тестирование

Напомним, что этот вопрос уже обсуждался в одной из предыдущих глав.

Ранее в этой книге описывались понятия модульного программирования и каркаса тестирования, включая теорию этого вопроса и практическое применение этой методологии.

Однако кратко рассмотрим этот вопрос еще раз. Модульное программирование предполагает пакетную структуру приложения, при которой каждый компонент в ответ на заданный набор входных данных выдает нужный отклик. Для каждого компонента строится каркас тестирования, позволяющий проверить соответствие практических результатов теоретическим ожиданиям.

Такой каркас можно рассматривать как черный ящик, позволяющий проверить функциональность компонента в процессе разработки. Подобные каркасы можно соз-

давать на ранних стадиях проекта, проводя регрессионные тесты компонентов и обеспечивая достижение ожидаемых результатов. То есть, тестирование позволяет удостовериться, что внесенные изменения не нарушили полученной ранее функциональности.

Модульное тестирование позволяет выявить ошибки в коде. Оно не выявляет других проблем, поэтому является лишь частью тестов, обеспечивающих высокий уровень качества приложения. Для примера рассмотрим почтовое Web-приложение. Допустим, вы создали класс пользователя, почтового ящика и сообщения. Для каждого из них были разработаны модульные тесты.

Однако рассмотрим использование кнопки **Ответить**. При щелчке на ней должен инстанцироваться новый объект сообщения, и свойства отправителя исходного объекта сообщения должны скопироваться в объект получателя нового сообщения. Это действие не выполняется никаким конкретным методом класса, поэтому не может быть проверено в рамках модульного тестирования.

Некорректная отправка электронной почты при щелчке на кнопке **Ответить** — это функциональная ошибка, требующая выявления и устранения. Однако ее нельзя выявить с помощью каркаса модульного тестирования. К счастью, модульное тестирование не является единственным способом проверки функциональной целостности. По существу, модульный тест проверяет модель в шаблоне проектирования MVC (модель–вид–контроллер).

Функциональное тестирование

Функциональное тестирование — это еще один вид обеспечения функциональной целостности и качества проекта. Оно призвано обеспечить выполнение функциональной спецификации.

Существует множество возможных подходов для формализации функционального тестирования. Его нужно формализовать в первую очередь. Однако для больших проектов недостаточно быстро проверить базовую функциональность.

При разработке больших проектов создается команда тестировщиков, состоящая из образованных людей, эмулирующих целевую аудиторию. Каждому члену этой команды ставится задача протестировать определенную часть приложения и предоставить формальный отчет.

Отчет о функциональном тестировании

Возвращаясь к примеру с почтовым приложением, можно сформировать приведенную ниже таблицу. Столбцы **Действие** и **Ожидаемый результат** заполняются менеджером проекта. Команда тестировщиков заполняет столбцы **Реальный результат** и **Итог**.

Действие	Ожидаемый результат	Реальный результат	Итог
Откройте страницу регистрации, щелкнув на соответствующей ссылке пользователя и пароля, а также кнопку Отправить	Отображается страница регистрации, содержащая текстовые поля для ввода имени и пароля, а также кнопку Отправить	Совпадает с ожидаемым	Тест пройден
Введите регистрационное имя <code>joet1301f</code> и пароль <code>ixrsh0z1</code> , а затем щелкните на кнопке Зарегистрироваться	Заполненная форма отправляется, и перезагружается страница с сообщением об успешной регистрации. Через 5 секунд автоматически открывается начальная страница	Совпадает с ожидаемым	Тест пройден

Окончание таблицы

Действие	Ожидаемый результат	Реальный результат	Итог
Щелкните на ссылке Входящие	Вы переходите в папку Входящие и видите список сообщений. Для каждого из них отображаются тема, дата получения и отправитель. Отображается также количество страниц, содержащих список почтовых сообщений. Если список состоит из нескольких страниц, отображаются кнопки Предыдущая и Следующая	Совпадает с ожидаемым, но кнопка Следующая не отображается	Тест не пройден

Очень важно отследить все непройденные тесты. Менеджер проекта должен внимательно оценить результаты тестирования и зафиксировать их в системе управления ошибками.

Содержание отчета функционального теста определяется функциональной спецификацией. Некоторые функциональные тесты необходимо проводить в заданной последовательности. Например, важно удостовериться, что без регистрации в системе пользователь не сможет выполнить никаких действий. Если тестировщик не может выполнить указанную последовательность действий, он должен отметить это в своем отчете и двигаться дальше.

Еще о функциональном тестировании

При функциональном тестировании необходимо проверить корректность всех аспектов работы системы. Для этого можно построить соответствующее дерево решений. Например, если пользователь может зарегистрироваться с тремя различными правами доступа (как обычный пользователь, привилегированный пользователь или администратор), то необходимо предусмотреть три ветви. Если в каждом из случаев он может выбрать несколько различных вариантов, необходимо предусмотреть соответствующие ветви тестирования и для них.

Построенное дерево впоследствии можно преобразовать в отчет по тестированию.

Этот метод функционального тестирования гораздо более эффективен, чем обычная проверка работоспособности приложения командой разработчиков. Отчет по тестированию можно предоставить заказчику в качестве дополнительного доказательства добросовестности своей работы.

Тестирование нагрузки

Тестирование нагрузки призвано определить экстремальные условия работы приложения.

В первую очередь необходимо обеспечить поддержку указанного в технической спецификации количества одновременных подключений, а также необходимое время отклика.

Важно получить как можно более точные цифры. Заказчик предпочитает не просто узнать, что приложение поддерживает 500 одновременных подключений, как и требуется в спецификации. Он предпочтет услышать, что при текущей инфраструктуре приложение может поддерживать 530 одновременных подключений. Столь небольшой запас прочности может не обрадовать заказчика, но лучше узнать об этом раньше, чем позже.

Принципы тестирования нагрузки детально описаны в приложении В.

Тестирование удобства использования

Вспомним отчеты по функциональному тестированию, описанные выше в этой главе. В приведенной таблице задача тестировщика описана следующим образом: “Откройте страницу регистрации, щелкнув на соответствующей ссылке”.

Эта же задача при тестировании удобства использования была бы перефразирована так: “Зарегистрируйтесь в системе”. Фразы существенно отличаются, как отличаются и цели. При тестировании удобства использования не ставится задача проверить работоспособность элементов системы. Для этого предназначено функциональное тестирование. Тестирование удобства использования призвано определить, насколько легко и эффективно пользователь может выполнить поставленную задачу.

При этом виде тестирования важным является не ответ на вопрос, а процесс его получения.

Действия пользователя по достижению цели нельзя оценить в цифрах, однако за ними нужно пронаблюдать и проанализировать.

Например, попробуйте ответить на следующие вопросы.

- Где пользователь искал ссылку? Нашел ли он ее там, где искал? (Направление поиска можно отследить по движению указателя мыши на экране.)
- Сколько прошло времени между загрузкой окна регистрации и началом ввода данных пользователя?
- Посетил ли пользователь другие страницы, не найдя сразу нужную?

Ответы на эти вопросы позволят оценить удобство использования вашего приложения, хотя их трудно выразить в цифрах.

Тестирование удобства использования — это чрезвычайно важный вопрос, рассмотрение которого нельзя уместить в одной главе. Более подробная информация по этому вопросу содержится на замечательном Web-узле Яакоба Нильсена по адресу www.useit.com.

Отслеживание ошибок

Описанные выше тесты позволяют прийти к следующим заключениям.

- Функциональное тестирование показало, что при ответе на полученное сообщение адрес получателя отображается некорректно.
- В процессе тестирования нагрузки выяснилось, что система поддерживает 530 одновременных сессий.
- В процессе тестирования удобства использования стало ясно, что кнопки перехода на следующую и предыдущую страницы списка нужно разместить вверху, а не внизу страницы.

Эти наблюдения являются бесполезными, если непонятно, как исправить найденные недочеты. Для перечисленных ошибок можно предусмотреть следующие варианты решения проблемы.

- Убедитесь, что при ответе на сообщение имя отправителя исходного сообщения автоматически копируется в поле получателя ответа.
- Если одновременно активны 530 сессий, отображайте для новых пользователей страницу с сообщением о недоступности сервера.

- Переместите кнопки перехода на предыдущую и следующую страницы в верхнюю часть страницы.

Эти действия достаточно просты. Однако в большом проекте могут возникнуть сотни и даже тысячи простых задач, связанных с устранением ошибок. Как отслеживать их выполнение?

Отслеживание ошибок с помощью системы Mantis

Во второй части этой главы мы познакомимся с системой управления ошибками Mantis, бесплатно распространяемой по адресу <http://mantisbt.sourceforge.net>.

Задача Mantis — обеспечить эффективную регистрацию ошибок в проекте, распределение задач по их устранению между разработчиками и добавление комментариев разработчиков после решения проблемы.

Mantis — это Web-приложение, написанное на языке PHP (а на чем же еще?). Вы можете открыть доступ к этой системе для своих заказчиков. Иногда это полезно, потому что они становятся участниками процесса обеспечения качества. Система поддерживает одновременную работу нескольких пользователей, разграничение прав и уровней доступа, поэтому для менеджера проекта и разработчиков можно установить различные права.

Существуют, конечно, и другие системы управления ошибками. Мы остановились на Mantis, потому что эта система бесплатна, проста в использовании и написана на PHP. Значит, теоретически вы можете ее модифицировать и расширять.

Инсталляция Mantis

Загрузите установочный архив с Web-узла и распакуйте его.

```
# tar -xzvf mantis-0.18.2.tar.gz
```

При этом будет создан каталог `mantis-0.18.2`, содержащий все исходные и конфигурационные файлы Mantis. В конфигурационном файле сервера Apache `httpd.conf` необходимо установить виртуальный узел, указывающий на этот каталог.

```
<VirtualHost 192.168.168.2
    ServerAdmin you@example.com
    ServerName mantis.example.com
    CustomLog /home/ed/logfile_mantis common
    ErrorLog /home/ed/errorlog_mantis
    DocumentRoot /home/ed/public_html/mantis-0.18.2
</VirtualHost>
```

Если вы не используете сервер Apache, то для поддержки Mantis вам придется установить виртуальный сервер.

Настройка системы при первом использовании

Работа Mantis основана на использовании базы данных MySQL, поэтому ее тоже необходимо установить. В других главах этой книги мы ориентировались исключительно на СУБД PostgreSQL, но Mantis не оставляет другого выбора. Система Mantis написана довольно хорошо, поскольку использует уровень абстракции базы данных, описанный в файле `core/database_api.php`. Если вы являетесь опытным программистом, то можете перенастроить систему на использование базы данных PostgreSQL.

Mantis рассчитана на занятого пользователя, поэтому позволяет сформировать нужные таблицы базы данных автоматически с помощью мастера. Просто создайте базу данных с именем Mantis и введите следующие команды.

```
# /usr/local/mysql/bin/mysqladmin -uroot -p<ваш пароль root> create mantis
# /usr/local/mysql/bin/mysql -uroot -p<ваш пароль root> mantis <
sql/db_generate.sql
```

Если сервер MySQL работает не на той же машине, что и Web-сервер, придется воспользоваться директивой `-h`. Например:

```
# /usr/local/mysql/bin/mysqladmin -hимя_сервера -uroot -p<ваш пароль root>
create mantis
# /usr/local/mysql/bin/mysql -hимя_сервера -uroot -p<ваш пароль root> mantis <
sql/db_generate.sql
```

При этом будет создана пустая база данных, все модификации которой будут выполняться с помощью самой системы Mantis.

Теперь осталось сообщить системе, где искать базу данных. Скопируйте файл `config_inc.php.sample` в файл `config_inc.php` в каталоге установки, а затем откройте его в любом текстовом редакторе.

Вы увидите множество настраиваемых параметров, наиболее важными из которых являются имя и местоположение базы данных. Модифицируйте эти строки в соответствии с вашей конфигурацией.

```
# установите собственные значения
$g_hostname      = "db";
$g_port          = 3306;           # 3306 - значение по умолчанию
$g_db_username   = "root";
$g_db_password   = "myrootpassword";
$g_database_name = "mantis";
```

Заметим, что в предыдущем примере для подключения к MySQL использовалось имя пользователя `root`. Для рабочей версии системы это плохая идея. Поэтому целесообразно создать менее привилегированного пользователя, предоставив ему доступ только к базе данных Mantis. Это следует указать в конфигурации.

Теперь можно зарегистрироваться в системе Mantis.

Регистрация в качестве администратора

Запустите Web-браузер и укажите в нем адрес созданного ранее виртуального узла. Вы увидите окно регистрации, показанное на рис. 23.1.

Зарегистрируйтесь как администратор. По умолчанию для MySQL используется пароль `root`.

Теперь можно выполнить соответствующие настройки: создать новую учетную запись и присвоить ей новый пароль. Это очень важно для доступа к Mantis через Интернет.

Создание и редактирование пользователей

Всем пользователям Mantis потребуются учетные записи.

Чтобы создать учетную запись, зарегистрируйтесь в качестве администратора и щелкните на ссылке **Manage**. При ее выборе появится список пользователей системы. Для создания новой учетной записи щелкните на кнопке **Create New Account**. Не забудьте установить соответствующий уровень доступа.

Создав все необходимые учетные записи (включая как минимум одну с правами администратора), вы можете полностью удалить учетную запись `root`.

Для модификации свойств пользователя нужно щелкнуть на его имени. На рис. 23.2 показан пример сеанса редактирования данных пользователя в Mantis.



Рис. 23.1.

Добавление пользователя в проект

Страницу редактирования пользователя можно использовать не только для изменения информации о нем, но и для добавления пользователя в проект. По умолчанию пользователь не имеет доступа ни к одному проекту. Значит, имея статус репортера, пользователь не может сообщать ни о каких ошибках.

Для добавления пользователя в проект выберите проект из списка (при первом запуске Mantis этот список пуст) и щелкните на кнопке Add User. Если этот проект помечен как открытый, то данный шаг можно пропустить, поскольку доступ к такому проекту имеют все пользователи. Но этим не стоит увлекаться, особенно если доступ к системе имеют ваши заказчики.

При создании учетной записи для нового пользователя ему автоматически отправляется электронное сообщение с указанием имени пользователя и пароля, а также адреса системы Mantis. При желании вы можете отключить эту функцию в конфигурационном файле Mantis.

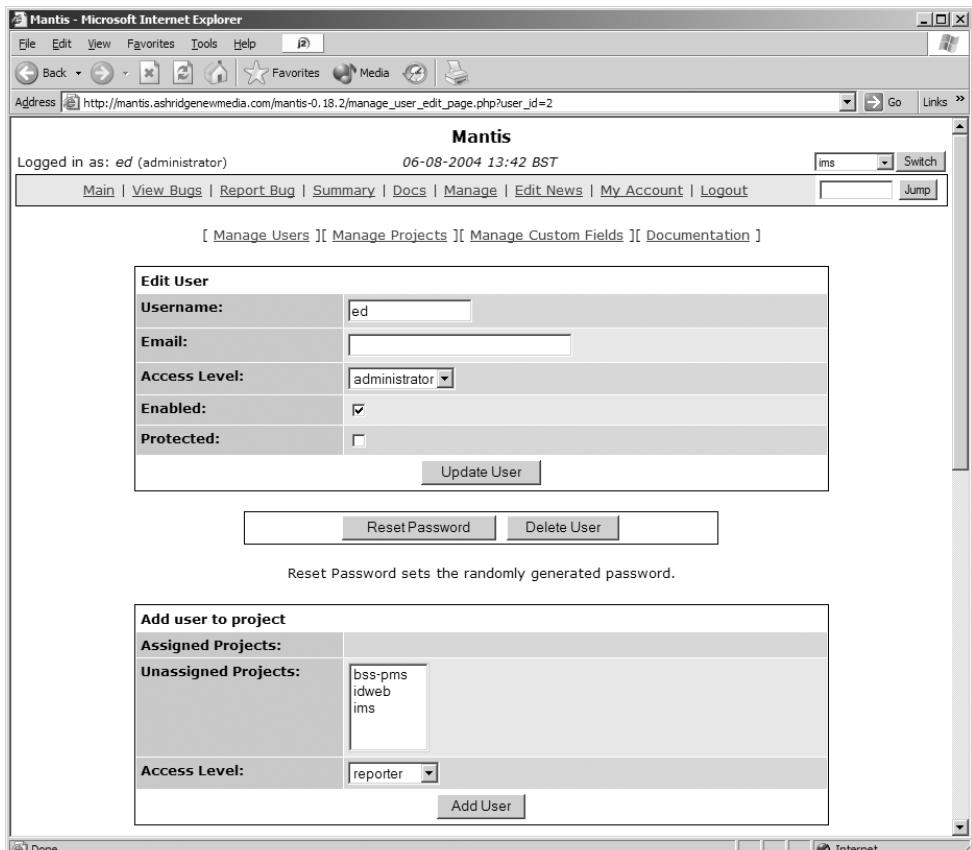


Рис. 23.2.

Создание новых проектов

Система Mantis поддерживает неограниченное количество проектов.

Чтобы создать новый проект, щелкните на ссылке **Manage Projects**. Появится список существующих проектов. Чтобы создать новый, щелкните на кнопке **Create New Project**.

Вам будет предложено указать имя проекта, его статус (произвольный флажок для вашего личного использования), статус просмотра (по указанным выше причинам следует выбирать закрытые проекты), описание и путь к загружаемому файлу. Путь к загружаемому файлу указывает место хранения двоичных файлов (копий экранов с ошибками), связанных с обнаруженными в системе ошибками. Соответствующий каталог должен быть доступен для записи Web-сервером. Возможно, требуемую настройку придется сделать вручную.

Выбор категорий ошибок

Создав новый проект, вы можете отредактировать информацию о нем, щелкнув на его имени. В первую очередь необходимо определить категории ошибок для данного проекта. Каждая ошибка относится к определенной категории, количество которых можно задавать произвольным образом. На рис. 23.3 показан типичный проект с указанием категорий ошибок.

Mantis - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Favorites Media Print Address http://mantis.ashridgenewmedia.com/mantis-0.18.2/manage_proj_edit_page.php?project_id=5 Go Links

Enabled	
View Status	private
Upload File Path	/data/mantis-ims/
Description	IMS (Insight)
<input type="button" value="Update Project"/>	
<input type="button" value="Delete Project"/>	

Categories		
Category	Assign To	Actions
copy / text		[Edit] [Delete]
design and appearance		[Edit] [Delete]
functionality: business logic		[Edit] [Delete]
functionality: desirable (ex-scope)		[Edit] [Delete]
functionality: incorrect		[Edit] [Delete]
functionality: malfunction		[Edit] [Delete]
functionality: missing (ex-scope)		[Edit] [Delete]
functionality: missing (in-scope)		[Edit] [Delete]
functionality: user interface		[Edit] [Delete]
other		[Edit] [Delete]
project: management and administration		[Edit] [Delete]
project: quality assurance		[Edit] [Delete]
schema		[Edit] [Delete]
systems / infrastructure		[Edit] [Delete]
<input type="text"/>		<input type="button" value="Add Category"/>
bss-pms	<input type="button" value="Copy Categories From"/>	<input type="button" value="Copy Categories To"/>

Versions	
<input type="text"/>	
<input type="button" value="Add Version"/>	

http://mantis.ashridgenewmedia.com/mantis-0.18.2/manage_proj_cat_delete.php?project_id=5&category_id=1 Internet

Рис. 23.3.

Можно ввести следующие категории.

- Тестовые изменения.
- Дизайн и внешний вид.
- Удобство использования.
- Функциональность: бизнес-логика.
- Желаемая функциональность.
- Функциональность: некорректности.
- Недостающая функциональность.
- Функциональность: интерфейс пользователя.
- Управление проектом и администрирование.
- Схема базы данных.
- Инфраструктура системы.

Создав набор категорий, вы можете использовать их и в других проектах, скопировав с помощью кнопок **Copy Categories From** и **Copy Categories To**.

При входе в систему необходимо выбрать соответствующий проект. Это можно сделать с помощью раскрывающегося списка в правом верхнем углу экрана.

Сообщения об ошибках

Теперь можно отслеживать ошибки в системе. Не забывайте формулировать сообщения об ошибках не просто в виде констатации факта, а в форме задачи, которую необходимо решить. Например, сообщение “Переместите кнопку с нижней части страниц вверх” выглядит гораздо лучше, чем “Кнопка расположена неправильно”.

Для добавления информации об ошибке выберите на панели навигации элемент **Report Bug**. Для начала нужно выбрать категорию ошибки из сформированного ранее списка.

Затем можно добавить и другую информацию, в том числе следующую.

- Частота возникновения ошибки. Если требуется добавить новое свойство, этот параметр задать нельзя.
- Сложность. Перечень возможных значений этого свойства является встроенным в Mantis. Большинство из них понятны по названию.
- Приоритет. Обозначает приоритет устранения ошибки командой разработчиков.
- Резюме. Краткое одностroочное описание ошибки.
- Описание. Детальное пояснение ошибки.
- Дополнительная информация. Дополнительные сведения, помогающие разработчикам решить проблему.
- Загружаемый файл. Если вы хотите связать с данной ошибкой двоичный файл, например копию экрана, на котором видна ошибка, то можете загрузить ее. Этот файл будет храниться на сервере в папке, имя которой было указано при задании параметров проекта.

После ввода информации об ошибке ее можно просмотреть с помощью главного меню навигации. Однако эта информация ничего не значит, если данную ошибку не поручено устранить конкретному разработчику.

Распределение задач по устранению ошибок

После ввода информации об ошибках задачи по их устранению необходимо распределить между отдельными разработчиками.

Выберите ошибку из списка, и рядом с кнопкой **Assign To** вы увидите раскрывающийся список с именами пользователей. Выберите пользователя, которому вы хотите поручить устранение ошибки, и щелкните на кнопке.

Счастливому избраннику по электронной почте будет отправлено сообщение с прямой ссылкой на данную ошибку.

Задания можно распределять пакетами. Это удобно при работе над небольшими проектами, когда устранение всех ошибок поручается одному разработчику. Установите флажки рядом со всеми выбранными ошибками, выберите пользователя, которому поручается их устранение, и отправьте ему список ошибок.

Комментирование ошибок

Возможно, разработчики захотят добавить свои вопросы или комментарии, связанные с некоторой ошибкой.

Для этого им придется ввести текст в соответствующее поле и щелкнуть на кнопке **Add Bug Note**. Комментарий будет добавлен рядом с описанием ошибки, а его копия будет отправлена человеку, выявившему данную ошибку. Тестировщик, выявивший ошибку, возможно, захочет добавить свои комментарии, которые тоже будут отправлены разработчику.

Разработчик может указать состояние решения проблемы, установив флажок **not a bug** (не является ошибкой), **won't fix** (не хочу исправлять), **fixed** (исправлена) и т.д. Информация о любых изменениях статуса тоже отправляется по электронной почте тестировщику, зафиксировавшему ошибку.

Устранение ошибок

После устранения ошибки разработчик щелкает на кнопке **Resolved**. При этом изменяется состояние обработки ошибки и ее общее состояние, а соответствующее письмо отправляется тестировщику.

Если менеджер проекта согласен с предлагаемым решением, он “закрывает” вопрос, а если не согласен, то изменяет статус устранения ошибки на **Reopened**. Сообщение об этом отправляется разработчику.

Флажки состояния используются на странице просмотра ошибок для выделения ошибок с фиксированным состоянием. Статус ошибки выделяется с помощью цвета.

Несколько заключительных слов о системе Mantis

Система Mantis является понятным и мощным пакетом, заслуживающим внимательного изучения.

Ввиду ограниченного объема книги, мы не будем рассматривать все ее преимущества, а направим читателя на [Web-узел `http://mantisbt.sourceforge.net`](http://mantisbt.sourceforge.net), где можно найти детальную информацию об этой системе.

Резюме

В этой главе вы познакомились с вопросами обеспечения качества приложения и узнали, что понимается под этим термином.

Было рассказано, какие аспекты необходимо протестировать, чтобы оценить и измерить качество данного проекта.

И наконец, читатель познакомился с системой Mantis — мощным средством управления ошибками. Было рассказано о том, как установить и настроить систему, а также как использовать ее в текущих проектах.

Итак, вы разработали качественный продукт, не содержащий ошибок и недочетов. В следующей главе будет описана эффективная стратегия развертывания, предполагающая перенос программной системы с сервера разработки в реальную среду.

24

Развертывание

Итак, приложение автоматизации торговли полностью разработано и протестировано благодаря методологии обеспечения качества, описанной в предыдущей главе.

Процесс загрузки приложения на рабочий сервер новичкам может показаться тривиальным, но профессионалы знают, что это совсем не так. В процессе развертывания необходимо придерживаться некоторых базовых принципов, важнейшими из которых являются контроль версий и обеспечение качества.

В этой краткой главе будет рассказано о структурировании серверной среды и управлении переносом кода на новые серверы. Вы узнаете, как и когда применять каждый метод переноса кода, чтобы наилучшим образом удовлетворять потребности команды разработчиков и заказчика.

Организация среды разработки

При разработке крупномасштабных PHP-приложений в процессе развития системы придется использовать несколько серверов или сред разработки.

Естественно, любое приложение, будь-то открытый Web-узел или закрытая корпоративная система, будет размещаться на сервере, расположенном в вычислительном центре или специальной комнате компании-заказчика.

Этот сервер называется рабочим, или производственным. Его можно приобрести по завершении проекта, поэтому в процессе развертывания приложения на сервере придется выполнить его настройку.

Сервер разработки

В процессе создания приложения необходимо использовать локальную среду разработки.

Допустим, создаваемое приложение называется Widgets. Локальный сервер разработки может располагаться во внутренней сети компании, иметь IP-адрес вида 192.168.1.1 и быть недоступен извне. Можно также использовать внутреннее доменное имя, например mydevelopmentco.local, доступное только для внутренних серверов DNS.

На Web-сервере можно установить виртуальный сервер.

`http://dev.widgets.mydevelopmentco.local`

Соответствующий элемент конфигурационного файла сервера Apache `httpd.conf` будет иметь такой вид.

```
<VirtualHost 192.168.1.1:80>
    ServerName dev.widgets.mydevelopmentco.local
    ServerAdmin ed@example.com
    CustomLog /home/widgets/logfile_wid common
    ErrorLog /home/widgets/errlog_wid
    DocumentRoot /home/widgets/public_html/dev/php
</VirtualHost>
```

Здесь придется указать IP-адрес, имя сервера, адрес электронной почты, соответствующие пути. Очевидно, что придется создать каталог для пользовательских элементов интерфейса, создаваемых командой разработчиков, чтобы они могли модифицировать код. Код можно модифицировать на рабочих станциях Windows, а для обеспечения сетевого доступа к исходному коду использовать пакет Samba (www.samba.org).

Внесенные модификации будут видимы только по URL-адресу `http://dev.widgets.mydevelopmentco.local`, а не на реальном узле.

Если вы используете методологию управления версиями, описанную в приложении А, вам придется установить экземпляр виртуального сервера для каждого разработчика, например `http://johndoe.dev.widgets.mydevelopmentco.local`. Эта технология более подробно описана в приложении А. В этом случае главный сервер разработки тоже используется, но не для текущей версии системы, а для хранения системы контроля версий.

В серьезных средах разработки создание элементов DNS для каждого проекта, а в случае контроля версий и для каждого разработчика, является ресурсоемкой процедурой. Если вы разрабатываете все проекты на одном физическом сервере, то можете упростить этот процесс с использованием WildCard DNS. Это означает, что любое имя `*.mydevelopmentco.local` будет автоматически связываться с IP-адресом 192.168.1.1. Более подробную информацию можно получить в Интернете, набрав в любой системе поиска ключевую фразу WildCard DNS.

Поэтапная разработка

Команда разработчиков зачастую устанавливает рабочий сервер у себя и тестирует на нем готовое приложение перед передачей заказчику.

При использовании контроля версий вы будете иметь последнюю версию каждого файла, одобренную для передачи заказчику. Такие файлы можно пометить дескриптором `release`. Более подробно об этом рассказывается в приложении А. Контроль подобных выпусков осуществляется менеджером проекта, который должен отслеживать ход его выполнения и обеспечивать качество проекта.

В предыдущем примере среду поэтапной разработки можно назвать

`http://studiotesting.widgets.mydevelopmentco.local`

Соответствующий элемент настройки сервера Apache будет иметь следующий вид.

```
<VirtualHost 192.168.1.1:80>
  ServerName dev.widgets.mydevelopmentco.local
  ServerAdmin ed@example.com
  CustomLog /home/widgets/logfile_wid-staging common
  ErrorLog /home/widgets/errlog_wid-staging
  DocumentRoot /home/widgets/public_html/staging/php
</VirtualHost>
```

Среда поэтапного развертывания

Среда поэтапного развертывания обеспечивает возможность демонстрации заказчику новой функциональности перед ее развертыванием в рабочей среде.

Эта среда обычно настраивается на производственном сервере, но ее адрес URL несколько отличается. Например, если рабочее приложение будет располагаться по адресу `http://www.widgets.com`, то для поэтапного развертывания можно использовать адрес `http://staging.widgets.com`.

Соответствующий элемент конфигурации сервера Apache будет иметь такой вид.

```
<VirtualHost 192.168.2.1:80>
  ServerName staging.widgets.com
  ServerAdmin ed@example.com
  CustomLog /home/widgets/logfile_wid-staging common
  ErrorLog /home/widgets/errlog_wid-staging
  DocumentRoot /home/widgets/public_html/staging/php
</VirtualHost>
```

Обратите внимание на различие подсетей в IP-адресах. Для каждого кластера серверов целесообразно использовать свой диапазон закрытых адресов. Поскольку среда поэтапного развертывания будет находиться не в вашей локальной сети, а в центре данных, вместо `192.168.1.x` была добавлена вторая подсеть `192.168.2.x`. Наличие различных подсетей позволяет настроить виртуальную частную сеть VPN для доступа к удаленному центру данных.

Если ваше приложение будет располагаться на нескольких серверах с балансированной нагрузкой, то среду поэтапного развертывания можно разместить на одном сервере, поскольку при тестировании приложения большой нагрузки не предвидится.

Среду поэтапного развертывания целесообразно защитить именем пользователя и паролем. При этом не следует пользоваться файлом `.htaccess`, потому что этот файл может быть перенесен на рабочий сервер, и готовое приложение будет связано с тем же паролем.

Данный уровень безопасности целесообразно добавить непосредственно в конфигурационный файл Apache.

```
<VirtualHost 192.168.2.1:80>
  ServerName staging.widgets.com
  ServerAdmin ed@example.com
  CustomLog /home/widgets/logfile_wid-staging common
  ErrorLog /home/widgets/errlog_wid-staging
  DocumentRoot /home/widgets/public_html/staging/php
  <Directory /home/widgets/public_html/staging/php>
    AuthType Basic
    AuthName Staging
    AuthUserFile /home/widgets/.htpasswd
    Satisfy All
  </Directory>
```

```
Require valid-user
</Directory>
</VirtualHost>
```

Файл .htpasswd создается обычным образом, т.е. по адресу /usr/local/apache/bin/.htpasswd.

Рабочая среда

Рабочая среда — это место размещения приложений с обеспечением доступа пользователей. Она может не быть открытой в классическом смысле, но это место, куда помещается полностью протестированный и апробированный код.

Конфигурация сервера Apache для рабочей среды может иметь следующий вид.

```
<VirtualHost 192.168.2.1:80>
  ServerName www.widgets.com
  ServerAdmin ed@example.com
  CustomLog /home/widgets/logfile_wid-live common
  ErrorLog /home/widgets/errlog_wid-live
  DocumentRoot /home/widgets/public_html/live/php
</VirtualHost>
```

Разработка баз данных

Если в приложении используется база данных, то вам придется решить, какой экземпляр базы данных (включая физический сервер и соответствующее программное обеспечение) будет использоваться в каждой из сред.

Целесообразно предположить, что модификация кода приложения может потребовать изменения схемы базы данных, поэтому использование устаревшей базы с обновленным кодом может привести к возникновению ошибок.

Поэтому рассмотрим следующую стратегию.

- Для каждой среды разработки следует использовать отдельную локальную базу данных. При этом отдельная база данных должна создаваться для каждого разработчика.
- В главной среде разработки должна размещаться база данных, не связанная с отдельным разработчиком, но отражающая последние изменения схемы базы данных и обеспечивающая поддержку последней версии системы.
- В среде поэтапной разработки должна размещаться локальная база данных, не связанная с конкретным разработчиком, но отражающая последние изменения схемы базы данных. Эта база данных должна соответствовать последнему выпуску системы (помеченному дескриптором release).
- В рабочей среде, естественно, должен использоваться рабочий сервер базы данных и рабочая база данных.
- В среде поэтапного развертывания тоже должна использоваться рабочая база данных.

Если в использовании одновременно находится как минимум четыре базы данных, то отследить необходимые изменения схемы довольно проблематично.

Здесь не поможет даже средство контроля версий. Конечно, в хранилище можно хранить дамп структуры базы данных, но загрузка этого дампа не позволит физически

обновить схему базы данных. Ее придется создать заново, но при этом будут утеряны тестовые данные. При работе над большими проектами подобный файл дампа сложно использовать для выявления различий в схемах базы данных.

Проблему можно решить с помощью контроля версий. Создайте в хранилище папку под названием `db-changes` и потребуйте от разработчиков добавлять в нее текстовые файлы, содержащие последовательность операторов `ALTER` при изменении схемы базы данных.

Такие файлы тоже можно пометить дескриптором `release`, чтобы ответственный за поддержку сред разработки мог удостовериться в соответствии баз данных.

Выбор базы данных в коде приложения может привести к появлению многочисленных операторов `switch` или необходимости изменения параметров вручную. Однако существует и другой способ. В системе используется файл `constants.php`, в котором указывается адрес базы данных, ее имя, имя пользователя, пароль и т.д. Если его связать со свойством `$_SERVER["HTTP_HOST"]`, то IP-адрес базы данных, ее имя, имя пользователя и пароль можно определять динамически и обеспечивать автоматический выбор нужной базы данных в зависимости от используемого виртуального сервера. При этом каждый сможет пользоваться одним и тем же файлом констант.

Процесс развертывания

Стратегия развертывания не ограничивается настройкой сред разработки. Очень важно определить процесс установки исходного кода.

Этот процесс проиллюстрирован на рис. 24.1.

Суть процесса сводится к следующему.

- ❑ Отдельные разработчики работают в своей собственной среде разработки с собственными базами данных.
- ❑ Их код разворачивается в главной среде разработки. Это может происходить автоматически или по графику, определенному менеджером проекта. В этот момент вносятся все необходимые изменения в схему базы данных главной среды.
- ❑ Полученный код тестируется проверяется и утверждается менеджером проекта или главным архитектором. Некоторые файлы могут быть доведены до полной степени готовности и помечены дескриптором `release`.
- ❑ Этот код полностью или по частям передается в среду поэтапной разработки, где еще раз тестируется перед передачей заказчику.
- ❑ Затем код развертывается в среде поэтапного развертывания с использованием рабочей базы данных. В этот момент заказчик принимает либо отвергает внесенные изменения.
- ❑ И наконец, код переносится из среды поэтапного развертывания в рабочую среду. На этом этапе база данных не изменяется.

Теперь рассмотрим технологии обеспечения этих переходов.

Автоматическое извлечение данных из хранилища контроля версий

В больших проектах используются средства *контроля версий*, обеспечивающие согласованность кода различных разработчиков на всех стадиях проекта. Эти средства позволяют отслеживать вносимые изменения. Более подробная информация о контроле версий содержится в приложении А.

При использовании средств контроля версий время от времени приходится извлекать последнюю версию системы из хранилища.

Использование CVS

При использовании системы CVS эту операцию в системе UNIX можно выполнить с помощью следующей команды (в версии Windows нужно просто щелкнуть на соответствующей кнопке).

```
cvs -d :pserver:имя-пользователя@cvs-сервер:/путь checkout -r дескриптор модуль
```

Здесь *модуль* — это имя проекта, *имя-пользователя* — это имя пользователя системы, а *cvs-сервер* — это имя сервера. *Дескриптор* используется менеджером проекта или главным архитектором для обозначения различных версий каждого файла проекта. Например, последняя устойчивая версия приложения может содержать версию 1.3 одного файла, 1.5 другого, 1.4 третьего и т.д. Соответствующей версии можно присвоить дескриптор *lateststable* (новейший, устойчивый), тогда с помощью таких дескрипторов можно легко собрать последний выпуск приложения.

Эту команду можно запускать по графику, который может иметь следующий вид.

- Ежедневно в полночь содержимое хранилища независимо от дескриптора извлекается и устанавливается в главную среду разработки.
- Ежедневно в три часа ночи файлы хранилища с дескриптором *release* извлекаются и переносятся в среду поэтапной разработки.

Использование системы Visual SourceSafe

В приложении А система Visual SourceSafe рекламируется как более удобная и популярная альтернатива CVS. Однако с описанной выше задачей автоматического извлечения данных эта система справляется гораздо хуже. Система Visual SourceSafe (VSS) представляет собой выполняемый файл для Win32. Поэтому для автоматической настройки сред понадобится машина с операционной системой Win32, даже если сами среды разработки настроены в UNIX.

С помощью Samba каталоги с исходным кодом можно подключить как сетевые диски. Тогда на машине Win32 можно выполнять команды по графику и автоматически формировать последние версии системы аналогично использованию CVS. Синтаксис запуска VSS из командной строки достаточно сложен, но чаще всего можно ограничиться следующими командами.

```
SET SSDIR=\\sourcesafeserver\sourcesafe
SET SSUSER=ed
ss Workfold /$Widgets z:\public_html\live
ss Get /$Widgets -R
```

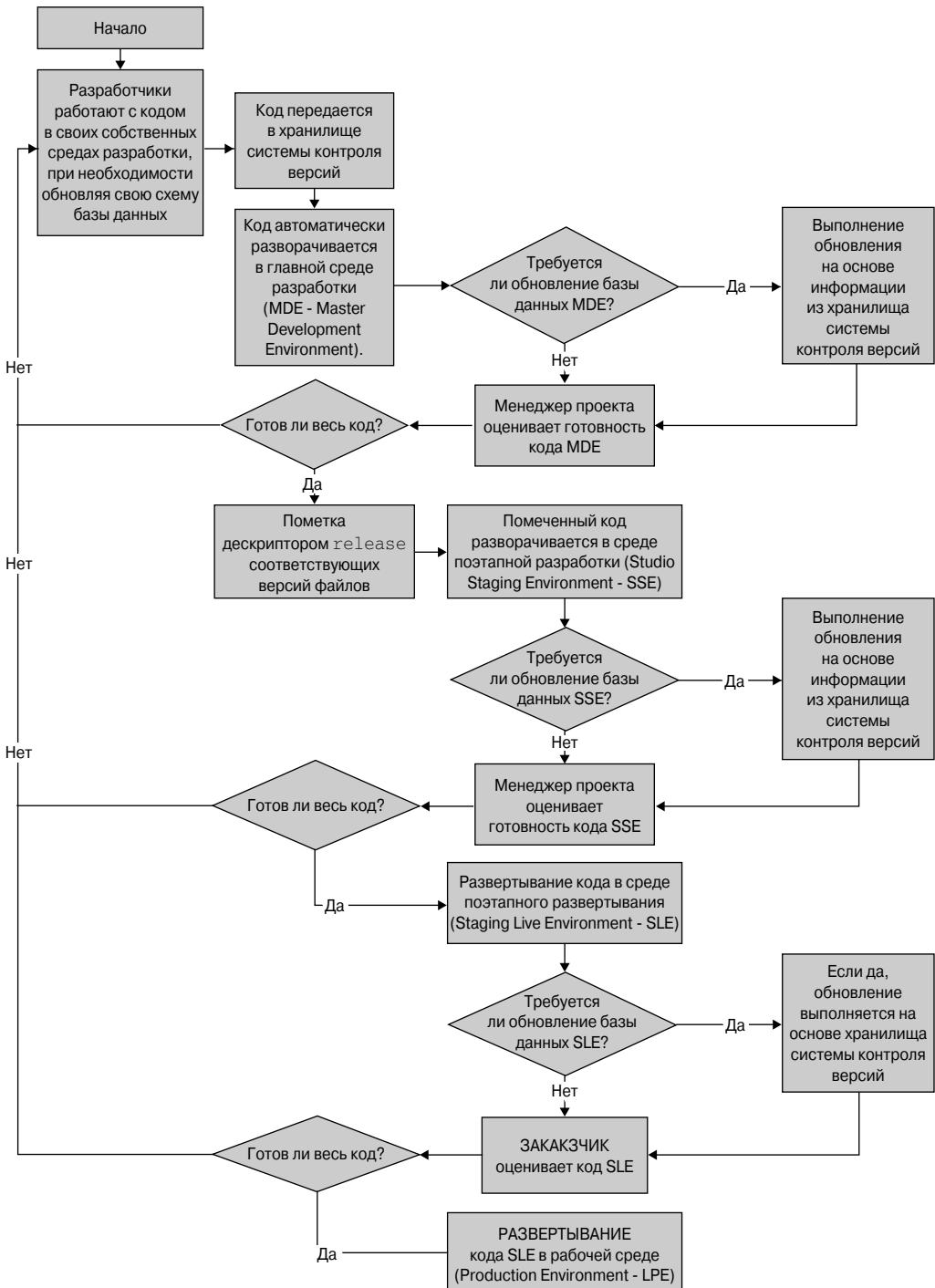


Рис. 24.1.

В этом примере в качестве рабочей папки проекта Widgets мы использовали каталог z:\public_html\live, доступ к которому с сервера разработки UNIX осуществляется через сетевой диск. Переменные окружения используются для определения местоположения хранилища и имени пользователя.

Команда Get является эквивалентом команды Get Latest Version в оконном приложении SourceSafe. Параметр -R обеспечивает рекурсивное извлечение информации из хранилища.

Более подробная информация об использовании системы SourceSafe из командной строки содержится по адресу:

http://msdn.microsoft.com/library/en-us/guides/html/vstskuse_command_line_syntax.asp

Утилита rsync

Как видно из приведенных примеров, самой популярной задачей на этапе развертывания является перенос содержимого папки A на сервере B в папку C на сервере D.

Сама по себе эта операция достаточно проста, но, учитывая размер современных проектов, эта задача сводится к передаче сотен мегабайтов через Интернет.

Ее можно выполнить с помощью утилиты rsync, которая работает только в среде UNIX, хотя в настоящее время предпринимаются многочисленные попытки перенести ее и в систему Windows. Она позволяет копировать содержимое папки на локальном сервере в заданную папку на удаленном сервере, перенося только измененные файлы. Другими словами, она синхронизирует содержимое двух папок.

Однако синхронизация выполняется только в одном направлении: от А к С. Измененные в папке А файлы копируются в папку С, а внесенные в С изменения обратно не переносятся. Однако это не проблема, если следовать сценарию развертывания, описанному выше в этой главе.

Утилиту rsync удобно применять в следующих ситуациях.

- ❑ Для развертывания данных среды поэтапной разработки в среде поэтапного развертывания.
- ❑ Для переноса среды поэтапного развертывания (после утверждения заказчиком) в рабочую среду.

Утилиту rsync можно использовать для синхронизации двух папок на одном и том же сервере. При копировании данных с сервера на сервер она использует протокол SSH. Для этого должен быть открыт порт 22. Существует и специальный протокол передачи данных для программы rsync, однако его нужно использовать только в том случае, если по каким-то причинам нельзя воспользоваться протоколом SSH.

Использование rsync

Утилита rsync используется в сценариях оболочки. Например, можно создать два сценария: deploy-studio-staging-to-live-staging и deploy-live-staging-to-live-production. Они будут работать одинаково, а утилита rsync будет вызываться следующим образом.

```
rsync -avrz -e ssh /локальный-каталог/* имя-пользователя@удаленный-узел:  
/удаленный-каталог
```

В этом вызове использованы следующие параметры.

- Флаг `-a` сохраняет разрешения на использование каталогов и символьные ссылки.
- Флаг `-v` обеспечивает вывод детального списка копируемых файлов.
- Флаг `-r` отвечает за рекурсивное копирование.
- Флаг `-z` обеспечивает сжатие данных перед передачей, а при копировании в рамках одного сервера игнорируется.
- Флаг `-e` отвечает за использование протокола SSH.
- Исходный каталог задается в виде `/локальный-каталог/*` (* означает копирование всех файлов каталога).
- Удаленный каталог задается в виде `имя-пользователя@удаленный-узел:/удаленный-каталог`. При копировании файлов в рамках одного сервера часть адреса слева от символа : не указывается.

При выполнении команды `rsync` необходимо ввести пароль, связанный с именем пользователя на указанном удаленном сервере.

Рассмотрим некоторые примеры использования команды `rsync`.

Примеры использования команды `rsync`

Сначала рассмотрим пример развертывания данных среды поэтапной разработки в среде поэтапного развертывания на сервере заказчика. При этом используется следующий синтаксис.

```
rsync -avrz -e ssh /home/widgets/staging/*
widgets@staging.widgets.com:/home/widgets/staging
```

Если заказчик принимает эти изменения, они переносятся в рабочую среду. При этом данные необходимо просто скопировать в папку на том же сервере с помощью следующей команды.

```
rsync -avr /home/widgets/staging/* /home/widgets/live
```

Обратите внимание, что здесь не задается параметр протокола, поскольку это локальная синхронизация. Кроме того, данные не сжимаются, поскольку это практически не обеспечит выигрыша в производительности.

Если рабочий сервер расположен на другой машине, придется использовать следующий синтаксис.

```
rsync -avrz -e ssh /home/widgets/staging/*
widgets@live:/home/widgets/live
```

Если заказчик реализует балансировку нагрузки между тремя серверами, копирование придется выполнить трижды.

```
rsync -avrz -e ssh /home/widgets/staging/*
widgets@web01:/home/widgets/live
rsync -avrz -e ssh /home/widgets/staging/*
widgets@web02:/home/widgets/live
rsync -avrz -e ssh /home/widgets/staging/*
widgets@web03:/home/widgets/live
```

Синхронизация серверов с помощью утилиты `rsync`

Утилита `rsync` играет важную роль и при использовании нескольких рабочих Web-серверов.

На многих узлах используются папки для записи данных сервером, выступающие в качестве контейнеров для больших выполняемых файлов и других данных, которые неудобно хранить в базе данных PostgreSQL или MySQL. В качестве таких файлов могут выступать, например, фотографии. При использовании нескольких рабочих Web-серверов возникает проблема: как обеспечить доступ к вновь загруженным данным для всех Web-серверов кластера.

Одним из возможных решений является использование централизованного файлового сервера и экспортование данных через систему NFS. Однако этот способ очень медленный.

Альтернативой является использование утилиты `rsync` для регулярной синхронизации данных на серверах. При обновлении данных на одном из серверов они в течение нескольких минут будут скопированы и на остальные.

Эту задачу можно эффективно решить с помощью утилиты `rsync`, однако ее автоматизации мешает использование пароля.

Для решения этой проблемы можно использовать беспарольный протокол SSH, тогда и для утилиты `rsync` пароль не понадобится. Это существенно упрощает задачу. Настройка протокола SSH выходит за рамки тематики данной книги, поэтому мы отправляем читателя к документации, расположенной по адресу:

<http://www.massey.ac.nz/~jriden/passwordless-ssh.html>

Резюме

В этой короткой главе вы познакомились с принципами развертывания корпоративных приложений, получили представление об эффективном структурировании сред разработки и синхронизации исходного кода с помощью программы `rsync`.

В следующей главе речь пойдет о создании надежной системы генерации отчетов для приложения автоматизации торговли.

25

Разработка надежной системы генерации отчетов

Передав систему заказчику, можно заняться и другими проблемами.

Наверняка у заказчика возникнет множество пожеланий по модификации системы. Чтобы проверить надежность ее работы, в течение нескольких недель придется скрупулезно собирать данные. На как справиться с такими большими объемами? В этой главе речь пойдет о том, как создать систему генерации отчетов, включающую типичные запросы и их обработку. Затем мы познакомимся с интеллектуальным подходом к построению системной архитектуры для генерации отчетов и определим наилучший способ доставки их заказчику.

Рабочие данные

Словосочетание “рабочие данные” означает информацию, собранную в процессе повседневной работы приложения. Приведем несколько примеров.

- ❑ В электронном магазине рабочими данными являются сведения о заказах.
- ❑ В приложении автоматизации торговли рабочими данными являются сведения, собранные в процессе общения продавца и покупателя.
- ❑ В корпоративной системе учета рабочего времени рабочими данными являются сведения о загруженности сотрудников.

Не все записи базы данных можно считать рабочими данными. Например, к ним не относятся данные о регистрации пользователей, группах доступа, сессиях, поскольку эта информация относится только к администрированию системы.

Понимание потребностей заказчика

Вряд ли вы сможете определить требования к системе отчетности на стадии разработки спецификации (см. главу 19). На этом этапе за деревьями не видно леса. Заказчик вместе с разработчиком пытаются определить задачи приложения, поэтому им сложно сделать шаг назад и посмотреть на систему под разными углами зрения.

Кроме того, в серьезных коммерческих приложениях отчеты используются только при необходимости. Вспомним докомпьютерную эру. Если руководитель обращался к своему секретарю с просьбой подготовить отчет, у того не было готового материала и требовалось составлять его с нуля. Это правило действует и сегодня. Требования к отчетности неразрывно связаны с коммерческими целями и условиями рынка, которые непрерывно изменяются.

Поэтому в функциональной спецификации целесообразно записать, что данное приложение должно обеспечивать генерацию отчетов, конкретный вид которых должен быть определен на этапе опытной эксплуатации приложения.

Управление запросами заказчика

При реализации системы отчетности главная задача состоит в эффективном управлении запросами заказчика.

На практике это означает, что вы должны читать между строк и на один шаг опережать видение заказчика. Формулируя свои желания, заказчик может не осознавать своих реальных потребностей, поэтому вы существенно упростите себе жизнь, если сможете предложить собственное решение, обеспечивающее достижение целей заказчика. Ниже эта мысль будет разъяснена более подробно.

Заказчик интуитивно пытается формулировать требования к системе отчетности. Однако его ограниченный опыт в решении подобных задач может оказаться бесполезным. Его собственный подход к проблеме (что я хочу) может неточно или неполно решать поставленную задачу (что ему нужно). Даже в случае формулировки разумных требований спроектированная заказчиком система отчетности не будет обладать возможностью повторного использования.

Рассмотрим следующий диалог между заказчиком и разработчиком.

Заказчик. Я хочу видеть данные продаж за июнь и сравнить их с данными за май.

Разработчик. Это не проблема.

Заказчик. Особенно меня интересует продажа книг издательства “Диалектика”. Я хочу сравнивать объемы продаж за май и за июнь в процентном соотношении.

Разработчик. Конечно. Через несколько часов мы предоставим вам нужный отчет.

При таком подходе заказчик будет обращаться к вам постоянно с просьбой сформировать отчет с несколько иными требованиями.

Чтобы этого избежать, вам необходимо взглянуть на процесс проектирования системы отчетности. Пусть заказчик определит свои цели, а вы спроектируете систему.

Решение этой проблемы зависит от ваших отношений с заказчиком. Некоторые из них вам доверяют и позволят взять бразды правления в свои руки. С другими все окажется гораздо сложнее.

Нужно постараться объединить цели заказчика со здравым смыслом и предложить привлекательное решение проблемы.

Заказчик. Я хочу сравнить объемы продаж за май и за июнь.

Разработчик. Я предлагаю отображать объемы продаж в долларах за выбранный период времени. Вы получите возможность оценивать объемы продаж для конкретного автора или группы авторов либо для конкретного издательства или группы издательств.

Заказчик. Хорошо. А можно ли увидеть процентное соотношение?

Разработчик. Замечательная идея. Дайте нам день или два, и все будет готово.

В этом диалоге разработчик перехватывает инициативу и предлагает заказчику свое решение.

Используя подобную тактику, нужно дать возможность заказчику вносить свои предложения (как в случае с процентным соотношением). Вы будете знать, что процесс разработки находится в ваших руках, но заказчик будет чувствовать себя причастным к нему.

Данные отчета

С заказчиком необходимо обсудить представление данных. Существует множество возможных форматов, для каждого из которых понадобится свой сценарий.

Следует избегать обсуждения требований на словах. В предыдущем диалоге возникла срочная необходимость подготовки отчета, поэтому он мог состояться и по телефону. Однако обсуждения вопросов отчетности необходимо проводить, сидя перед приложением. Несколько позже мы расскажем, как это обеспечить.

Для электронных отчетов возможны следующие форматы.

- HTML. Существенным преимуществом этого формата является его простота и возможность отображения в браузере заказчика. Недостатком этого формата является сложность красивой печати. Он подходит для небольших отчетов.
- PDF. С помощью класса PDF пакета R&OS (www.ros.co.nz/pdf) довольно просто генерировать на PHP документы в формате PDF. Однако при этом придется затратить много времени на создание таблиц. Зато полученный в результате PDF-файл будет хорошо выглядеть при печати.
- Microsoft Excel. Некоторые заказчики предпочитают отчеты в виде электронных таблиц Excel. Очевидно, таблицу Excel можно преобразовать в формат PDF, но на практике это делается очень редко. Сформировать таблицу Excel с помощью PHP можно с помощью пакета PEAR (http://pear.php.net/package/Spreadsheet_Excel_Writer).
- XML. Отчеты можно генерировать на языке XML, а затем преобразовывать их в один из трех перечисленных выше (или других) форматов с помощью XSLT или любого механизма преобразования (например, Apache FOP, <http://xml.apache.org/fop/>). Этот подход потребует больше времени для реализации, но позволяет разделить уровни извлечения данных и представления, что впоследствии позволит сэкономить усилия. Именно этого подхода мы будем придерживаться в этой главе.

Разработка отчета

Разработка отчета состоит из трех шагов: проектирование интерфейса пользователя, определение необходимого результата и определение способа извлечения данных из базы.

Проектирование интерфейса

При использовании предлагаемой архитектуры разработчику не понадобится заботиться о внешнем виде интерфейса пользователя. Он сможет сконцентрироваться на параметрах отчета.

Для каждого отчета необходимо определить перечень входных параметров. А для каждого из критериев нужно определить следующие показатели.

- Если это число, необходимо задать границы диапазона или указать точное число.
- Если это строка, нужно определить ее шаблон (при его наличии).
- Если это строка, нужно определиться, учитывается ли регистр.
- Требуется решить, включать ли в диапазон его границы.
- Нужно определить, является ли данный показатель обязательным.
- Возможно, в интерфейсе необходимо отображать некоторые значения по умолчанию.

Для предыдущего примера отчета о продаже книг можно сформировать следующий список параметров.

- Для первого временного диапазона: день, месяц и год начала (параметр обязательный, задается точно).
- Для первого временного диапазона: день, месяц и год завершения (параметр обязательный, задается точно).
- Для второго временного диапазона: день, месяц и год начала (параметр обязательный, задается точно).
- Для второго временного диапазона: день, месяц и год завершения (параметр обязательный, задается точно).
- Издательства (параметр необязательный, по умолчанию учитываются все).
- Авторы (параметр необязательный, по умолчанию учитываются все).

Это исчерпывающий список входных параметров отчета. Теперь необходимо определить, какой должна быть выходная информация.

Выходная информация

Для данного отчета не будем слишком много внимания уделять его внешнему виду. Предположим, что отчет генерируется в формате XML, а затем может быть преобразован в любой удобный формат.

В отчете будут отображаться следующие данные.

- Краткая информация о входных параметрах, используемых для генерации отчета.
- Объемы продаж в долларах за первый период времени.
- Объемы продаж в долларах за второй период времени.
- Разность объемов продаж для двух диапазонов в долларах.
- Различие в процентах.

Теперь необходимо решить, как получить эту выходную информацию на основе входных параметров.

Извлечение данных

Для данного отчета процесс извлечения данных довольно прост. Обычно он реализуется с помощью SQL-запросов к базе данных.

Для минимизации использования SQL-запросов в этой книге используется класс `GenericObject`. Такой подход улучшает переносимость приложения и ясность кода.

Однако здесь мы откажемся от этого подхода.

Для генерации отчета потребуется выполнить несколько отдельных запросов, поэтому нет необходимости тянуть за собой всю иерархию классов. Это только усложнит поддержку системы отчетности.

Кроме того, вполне возможно, что в будущем приложение будет дополнено модулями, написанными не на языке PHP, которые будут обеспечивать взаимодействие с базой данных.

Поэтому данные лучше извлекать напрямую с помощью оптимизированных SQL-запросов.

Каждый из запросов должен динамически зависеть от входных параметров.

Например, рассмотрим сценарий продажи книг. Если пользователь системы просто хочет знать объемы продаж за два периода, можно использовать следующие запросы.

```
SELECT COUNT(id) FROM order WHERE date_placed >= '2004-05-01' AND date_placed
<= '2004-05-31';
SELECT COUNT(id) FROM order WHERE date_placed >= '2004-03-01' AND date_placed
<= '2004-03-31';
```

Однако при модификации критерия придется изменить и вид запроса. Например, запрос для конкретного автора будет иметь такой вид.

```
SELECT COUNT(id) FROM order WHERE date_placed >= '2004-05-01' AND date_placed
<= '2004-05-31' AND book_id IN (SELECT id FROM book WHERE author_id=1295);

SELECT COUNT(id) FROM order WHERE date_placed >= '2004-03-01' AND date_placed
<= '2004-03-31' AND book_id IN (SELECT id FROM book WHERE author_id=1295);
```

В этих запросах для ясности используется упрощенная схема.

Каждый запрос можно сформулировать в различных форматах. Для комбинации критериев можно использовать оператор `switch`. Однако это нецелесообразно, поскольку для сложного набора критериев мы получим сотню возможных вариантов для этого оператора.

Поэтому для построения строки SQL-запроса лучше использовать стандартную условную логику. Рассмотрим следующий фрагмент кода.

```
$strSQL = "SELECT COUNT(id) FROM order WHERE date_placed >= '$from_date' AND
date_placed <= '$until_date'";
if ($author_id) {
    $strSQL .= "AND book_id IN (SELECT id FROM book WHERE author_id=$author_id)";
};
if ($publish_id) {
    $strSQL .= "AND book_id IN (SELECT id FROM book WHERE publisher_id = $publish_id)";
};
```

Здесь SQL-запрос генерируется динамически на основе определенных условий.

Этот подход можно применять для проектирования любых отчетов. Конкретные запросы следует использовать для жестко заданных входных параметров и для их возможных вариаций применять условную логику.

Архитектура генерации отчетов

В предыдущих разделах вы познакомились с принципами генерации отчетов на языке PHP. Но как реализовать эту теорию на практике?

Для этого можно предложить следующий подход.

- ❑ Страница `salesreport.php` обрабатывает форму запроса с входными параметрами.
- ❑ Эта страница выполняет HTTP-запрос по методу POST к странице `salesreport-results.php`, которая непосредственно реализует SQL-запросы для получения данных отчета и отображает эти данные на экране.

Такой подход имеет множество недостатков.

- ❑ Он не соответствует шаблону проектирования MVC.
- ❑ Данный код не подлежит повторному использованию.
- ❑ Если пользователь снова захочет увидеть данные отчета, он будет генерироваться заново. При этом не только будет потрачено время, но, возможно, будет получен другой результат. Возможно, за это время была изменена база данных.
- ❑ Отчет может генерироваться довольно долго. Для сложных отчетов этот процесс может занять одну или несколько минут. Пользователь может не дождаться результатов и щелкнуть на кнопке **Остановить** в Web-браузере. Web-страницы должны генерироваться довольно быстро. Это относится и к отчетам.

Последний из указанных недостатков является, пожалуй, самым серьезным. Рассмотренный пример отчета будет сгенерирован за несколько секунд даже при использовании самых медленных серверов баз данных. Однако в более серьезных случаях для генерации отчета может потребоваться сотня или даже тысяча запросов.

Учитывая то, что протокол HTTP не поддерживает соединения, обработка страниц, зависящая от внешних и потенциально медленных процессов, должна выполняться в фоновом режиме.

Вспомните работу Web-узлов электронной коммерции, предлагающих пользователям ввести номер кредитной карточки. Это всего лишь номер, но многие Web-узлы выводят предупреждение о том, что на кнопке **Передать** необходимо щелкнуть всего один раз, поскольку для авторизации карточки требуется порядка сорока секунд. Как узнать, что запрос все еще находится в процессе выполнения?

Интерактивный магазин Amazon.com использует принципиально другой подход, который является гораздо более предпочтительным. Почему авторизацию необходимо выполнять прямо сейчас? Почему бы ее не выполнить в фоновом режиме, сообщив о результатах клиенту?

Поскольку магазин Amazon.com обеспечивает физическую доставку своих товаров, а не электронную, то он может позволить себе уведомить пользователя об успешной авторизации карточки через несколько минут или даже часов. Однако этот же принцип можно применять и в сценариях реального времени.

Рассмотрим традиционный подход к авторизации кредитной карточки.

- ❑ На первой странице запрашивается номер кредитной карточки, который необходимо ввести в форму HTML. Данные передаются по протоколу HTTPS по методу POST.

- ❑ Вторая страница получает данные первой страницы, в реальном времени выполняет авторизацию карточки и через 30 секунд возвращает результаты пользователю.

Сравним этот метод с более интеллигентным подходом, “навеянным” компанией Amazon и предполагающим обработку запроса в псевдореальном времени.

- ❑ На первой странице запрашивается номер кредитной карточки, который необходимо ввести в форму HTML. Данные передаются по протоколу HTTPS по методу POST.
- ❑ Вторая страница получает данные первой страницы, помещает их в базу данных, подлежащих обработке, и возвращает пользователю страницу с информацией о ходе процесса обработки.
- ❑ Эта страница на стороне клиента обновляется каждые несколько секунд до тех пор, пока процесс авторизации не будет завершен.
- ❑ Совершенно отдельный процесс на сервере периодически опрашивает базу данных на предмет появления новых записей, и при их обнаружении обрабатывает их.

Такой подход имеет очевидные преимущества.

- ❑ Пользователь не станет щелкать на кнопке Остановить, поскольку практически без задержек получает ответ от сервера.
- ❑ Если пользователь щелкнет на кнопке Передать несколько раз, это не составит проблемы, поскольку уникальный ключ в базе данных предотвратит дублирование записей.
- ❑ Если пользователь в процессе ожидания щелкнет на кнопке Обновить, это не приведет к сбою процесса авторизации.

Эту же логику можно применить и для генерации отчетов.

Генерация отчетов в фоновом режиме

Генерация отчетов в фоновом режиме имеет огромные преимущества как для пользователя, так и для приложения. При этом отчет не будет генерироваться быстрее, поскольку для его формирования выполняются те же запросы к базе данных. Однако такой подход позволит избежать множества описанных выше проблем.

Основные принципы предлагаемого подхода сводятся к следующему.

- ❑ В таблицу report базы данных вносится шаблонная информация для каждого отдельного отчета в системе.
- ❑ Все возможные отчеты перечислены на странице reporting.php.
- ❑ Пользователь может выбрать конкретный отчет из таблицы. При этом выполняется переход к другой странице, которой в качестве параметра GET передается идентификатор отчета (например, newreport.php?report_id=14). Эта страница для формирования отчета использует соответствующий шаблон XML, который используется для задания входных параметров отчета.
- ❑ После ввода входных параметров отчет заносится в таблицу report_instance и помечается флагом PENDING.

- Входные параметры заносятся в таблицу `report_instance_criteria`, а пользователь перенаправляется на общую страницу отчетов **Мои отчеты**.
- На странице **Мои отчеты** отображается список всех отчетов для данного пользователя с указанием статуса (PENDING, PROCESSING или COMPLETED). Отчеты с пометкой COMPLETED можно просмотреть в одном из выбранных форматов (HTML, PDF или Excel).
- Отдельный сценарий PHP `reportprocessor.php` (расширение .php указывает на выполнение этого сценария в командной строке) постоянно работает на машине, связанной с сервером баз данных приложения, и непрерывно сканирует таблицу `report_instance` в поисках записей с флагом PENDING. Обнаружив такую запись, он присваивает ей статус PROCESSING, а затем порождает новый процесс формирования отчета.
- Этот процесс проверяет критерии данного отчета в таблице `report_instance_criteria` и строит SQL-запросы, необходимые для извлечения соответствующих данных. Полученные данные записываются в файл XML, а экземпляр отчета помечается как COMPLETED.
- Статус этого отчета поменяется и на странице **Мои отчеты**. Теперь пользователь может просмотреть отчет.
- Сценарий трансляции для данного отчета получает XML-файл и преобразует его в запрашиваемый формат.

Рассмотрим этот подход более детально. Мы не будем здесь приводить исходный код генератора отчетов, поскольку он занимает сотни страниц. В этом разделе мы лишь рассмотрим архитектурные особенности системы генерации отчетов. Полный исходный код, реализующий описанную методологию, можно найти на Web-узле www.wrox.com.

Ниже в этой главе будет приведен реальный пример применения этой методологии.

Страница Reports

Reports — это начальная страница компонента генерации отчетов. Если в приложении имеется кнопка генерации отчета, то при щелчке на ней переход будет осуществляться именно на эту страницу. Эта страница должна строиться на основе двух файлов: `reporting.php` и `reporting.tpl`, реализующих шаблон Smarty.

Задачи этой страницы следующие.

- Обеспечить вход пользователя в компонент генерации отчетов.
- Обеспечить ссылку на страницу **Мои отчеты**.
- Обеспечить перечень типов отчетов в системе и дать возможность пользователю создать новый экземпляр отчета.

Слово “экземпляр” в данном случае не имеет ничего общего с ООП. Это просто конкретный отчет. Например, общий отчет предназначен для вывода данных о продажах. На его основе можно сформировать экземпляр, сравнивающий объемы продаж за апрель и июнь. Общий отчет не связан ни с какими входными данными, в то время как конкретный экземпляр строится на основе заданных критериев.

В отдельной таблице содержится список типов возможных отчетов. Можно обеспечить некоторые административные средства для добавления в эту таблицу новых типов отчетов. Однако для каждого нового отчета потребуется создать специальные файлы

с его описанием. Поэтому создание новых шаблонов отчетов лучше не перекладывать на плечи заказчика, а оставить за собой. В следующей таблице приведена структура таблицы базы данных `report`, содержащей перечень возможных типов отчетов.

Столбец	Тип данных	Описание
<code>id</code>	SERIAL	Идентификатор, или первичный ключ таблицы
<code>report_code</code>	character varying(8) NOT NULL	Еще один уникальный идентификатор, предназначенный для чтения человеком, например <code>salesrep</code>
<code>report_name</code>	character varying(64) NOT NULL	Название отчета, например <code>Объемы продаж</code>
<code>report_desc</code>	character varying(256)	Текстовое описание отчета

Эта структура таблицы ориентирована на использование базы данных PostgreSQL, но может быть легко адаптирована к базе данных MySQL или любой другой платформе.

В таблице `report` хранится немного данных. Вся остальная информация будет храниться на диске.

В этой таблице содержится перечень возможных отчетов. Имя отчета можно взять из таблицы и сформировать для него новый экземпляр данного типа.

Идентификатор отчета передается как параметр метода GET странице `newreport.php`. Входная информация должна быть представлена с использованием шаблона Smarty (см. главу 13), например, в виде файла `reporting.tpl`.

Страница `newreport`

Эта страница вызывается на главной странице `Reports`, описанной в предыдущем разделе. Она тоже состоит из двух файлов (в файле `a.php` содержится модель, а в шаблоне Smarty – компоненты вида и контроллера). Эта страница выполняет обработку данных формы для выбранного отчета.

Шаблон Smarty можно использовать для каждого отчета, однако если приходится строить десятки или сотни отчетов, то перспектива создания вручную десятков или сотен HTML-форм не покажется вам слишком оптимистичной. В конце концов вы специалист по PHP, поэтому следует использовать свои навыки.

Для каждого конкретного отчета будем использовать шаблон XML.

Шаблоны XML

С каждым отчетом необходимо связать свой шаблон XML, определяющий входные данные. Возвращаясь к предыдущему примеру, в этом шаблоне необходимо предусмотреть обязательное задание двух начальных дат, двух завершающих дат и, дополнительно, автора или издательство.

В каждом документе XML должна быть предусмотрена связь с базой данных. Не используйте в качестве имени элемента XML столбец `id` таблицы `report`, поскольку он может изменяться при переходе на другой сервер или другую базу данных. Этот идентификатор предназначен только для внутреннего использования. Шаблоны XML следует именовать с использованием второго столбца таблицы `report`, например `/templates/salesrep.xml`.

Выбор формата XML остается за разработчиком. Однако в качестве примера можно привести следующий формат, который можно сохранить в файле /templates/salesrep.xml.

```
<report id="salesrep">
  <criterion type="date" name="datefrom1" mandatory="true" caption="Начальная
дата (1)"/>
  <criterion type="date" name="datetol" mandatory="true" caption="Конечная дата
(1)"/>
  <criterion type="date" name="datefrom2" mandatory="true" caption="Начальная
дата (2)"/>
  <criterion type="date" name="dateto2" mandatory="true" caption="Конечная дата
(2)"/>
  <criterion type="fkmultiple:author/id/author_name" name="authors"
caption="Выберите автора(ов)"/>
  <criterion type="fkmultiple:publisher/id/publisher_name" name="publishers"
caption="Выберите издаельство(а)"/>
</report>
```

В этом шаблоне каждый критерий определяется дескриптором `<criterion>`, который характеризуется именем, заголовком (отображаемым на экране) и типом.

Тип играет важную роль в описании критерия, поскольку определяет способ его отображения в Web-браузере. В следующей таблице приводятся некоторые стандартные типы.

Тип	Описание
freetext	Текстовое поле, данные которого хранятся как при вводе
date	Дата, представленная в виде трех компонентов (день, месяц, год), которая хранится в формате ISO (YYYY-MM-DD)
time	Время, представленное в виде трех компонентов (часы, минуты, секунды), которое хранится в формате ISO (HH:MM:SS)
datetime	Дата и время, представленные в виде шести компонентов (день, месяц, год, часы, минуты, секунды), которые хранятся в формате ISO (YYYY-MM-DD HH:MM:SS)
fkmultiple	Список, отображающий элементы другой таблицы по внешнему ключу. Этот тип данных имеет следующий формат: fkmultiple:имя-таблицы/хранимый-столбец/отображаемый-столбец. Здесь имя-таблицы определяет таблицу внешних сущностей, хранимый-столбец — это столбец с сохраняемыми значениями, а отображаемый-столбец — столбец, данные которого отображаются в списке. В рассмотренном выше примере этот формат используется для выбора из списка издательства или автора (из таблицы publisher и author, соответственно)
fksingle	Аналогичен предыдущему, но позволяет выбрать из внешней таблицы только одну сущность

Среди этих типов дополнительного объяснения требует только `fkmultiple` и `fksingle`.

Эти типы предназначены для формирования раскрывающихся списков на основе данных некоторой таблицы. В рассмотренном выше примере они применяются для выбора издательства или автора.

В этом состоит еще одно преимущество использования XML. Разработчику не приходится писать код для генерации подобных списков, а в каждом отчете можно использовать один и тот же тип данных.

Начальная дата (1)	<input type="text" value="01"/>	<input type="text" value="01"/>	<input type="text" value="2004"/>
Конечная дата (1)	<input type="text" value="01"/>	<input type="text" value="01"/>	<input type="text" value="2004"/>
Начальная дата (2)	<input type="text" value="01"/>	<input type="text" value="01"/>	<input type="text" value="2004"/>
Конечная дата (2)	<input type="text" value="01"/>	<input type="text" value="01"/>	<input type="text" value="2004"/>
Выберите автора(ов)	<input type="checkbox"/> Ed Lecky-Thompson <input type="checkbox"/> Heow Eide-Goodman <input type="checkbox"/> Steven D. Nowicki <input type="checkbox"/> Alec Cove		
Выберите издательство(а)	<input type="checkbox"/> Wiley <input type="checkbox"/> Roadrunner <input type="checkbox"/> Coyote Publishing <input type="checkbox"/> Compuglobalhypermegane		

Рис. 25.1.

Форма HTML, сформированная на основе этого шаблона, будет выглядеть примерно так, как показан на рис. 25.1.

В конкретном приложении эту форму можно несколько усовершенствовать, улучшив ее стиль оформления.

Данная форма генерируется на основе соответствующего шаблона XML.

Имя каждого элемента формы позволяет легко декодировать название критерия. Например, для критерия `datefrom1` можно использовать имя элемента формы `CRITERION_datafrom1`. Использование префикса `CRITERION_` позволяет отличить передаваемые параметры формы от остальных.

Эту архитектуру можно несколько модифицировать, добавив значения элементов формы по умолчанию.

Данные этой формы передаются сценарию `newreport.php`. Этот сценарий должен отличать запросы, направляемые по методу GET и POST, поскольку запросы GET используются для заполнения стандартных параметров шаблона.

Хранение запросов

Сценарий `newreport.php` должен корректно обрабатывать данные запроса POST. При этом необходимо создать новый экземпляр отчета и начать его обработку.

Каждый экземпляр генерируется только один раз и хранится в таком состоянии, чтобы при необходимости пользователь смог вернуться к старому отчету. Однако пользователь может также сгенерировать абсолютно новый экземпляр отчета, отражающий современное состояние дел.

Экземпляры отчетов должны храниться в таблице базы данных `report_instance` со следующей структурой.

Столбец	Тип данных	Описание
<code>id</code>	SERIAL	Идентификатор, или первичный ключ таблицы
<code>report_id</code>	int4	Запрашиваемый отчет, внешний ключ из таблицы <code>report</code>
<code>submitted</code>	datetime	Дата и время построения отчета
<code>submitting_user_id</code>	int4	Идентификатор пользователя, сгенерировавшего отчет, используемый для формирования страницы Мои отчеты
<code>status_flag</code>	character (1)	Статус отчета: P — ожидание, I — обработка, C — готовность

Вам также понадобится таблица для хранения критериев, связанных с каждым экземпляром отчета. При этом необходимо придерживаться правил нормализации базы данных.

Назовем эту таблицу `report_instance_criterion`. Она будет иметь следующую структуру.

Столбец	Тип данных	Описание
<code>id</code>	<code>SERIAL</code>	Идентификатор, или первичный ключ таблицы
<code>instance_id</code>	<code>int4</code>	Экземпляр отчета, внешний ключ из таблицы <code>report_instance</code>
<code>criterion_name</code>	<code>character varying(128)</code>	Имя критерия, например <code>datefrom1</code>
<code>criterion_value</code>	<code>character varying(128)</code>	Значение критерия, например <code>2006-01-01</code>

Такая структура таблицы обеспечивает поддержку нескольких критериев для данного отчета.

Сценарий считывает входные параметры из запроса POST и заносит их в базу данных. При этом для данного отчета используется статус P (`PENDING`), сигнализирующий сценарию-обработчику о необходимости генерации данного отчета. Из шаблона XML необходимо получить информацию об обязательных и необязательных полях. Если пользователь не заполнил обязательные поля, его необходимо вернуть на страницу с формой.

При хранении значений критерия можно столкнуться с проблемой плохой нормализации базы данных. В предыдущем примере пользователь может выбрать из списка несколько издательств или авторов. Если использовать правила нормализации Бойса-Кодда (третья нормальная форма), то для хранения этих данных понадобится не одна, а две таблицы. Но с целью упрощения множественные значения критериев можно хранить в одном и том же поле, разделяя их запятыми.

Однако если критерий может принимать сотни различных значений, то необходимо увеличить размер соответствующего поля, например, до 1024 символов. В PostgreSQL это не проблема, поскольку эта база данных допускает строковые поля длиной до 8192 символов. Когда соответствующая информация записана в таблице `report_instance` и `report_instance_criterion`, в действие вступает фоновый сценарий обработки запросов. При этом пользователя необходимо перенаправить на страницу `Мои отчеты`.

```
header("Location: /myreports.php");
exit(0);
```

На этой странице пользователь сможет отследить состояние всех запрошенных отчетов.

Сценарий обработки отчета

Сценарий обработки отчета — это пример сценария PHP, выполняемого в фоновом режиме. Его выполнение не инициируется Web-браузером, а запускается сервером приложения независимо от Apache.

Этот сценарий должен работать непрерывно, постоянно проверяя наличие в базе данных отчетов с флагом `PENDING`. Он переводит их в состояние обработки и вызывает соответствующий сценарий (обработчик) для реальной генерации результатов отчета.

Здесь можно использовать множество подходов. Сценарий можно запустить в бесконечном цикле либо запускать его по графику каждые несколько секунд.

Возможно, первый подход является более предпочтительным. Запуск сценария по графику удобен в том случае, если известно, сколько времени понадобится на его выполнение. В данном случае это неизвестно, поэтому не имеет смысла запускать множество экземпляров обработчика с непредсказуемым результатом. Лучше запустить сценарий в момент загрузки сервера, а процесс cron использовать для периодической проверки его работоспособности и перезапуска при случайном отказе.

Обработка

Псевдокод этого сценария будет иметь следующий вид.

- Начать цикл.
- Проверить наличие отчетов со статусом P (PENDING).
- Перевести каждый из них в состояние I (IN PROGRESS), определить код отчета, например salesrep, и найти обработчик с соответствующим именем, например /scripts/handlers/salesrep.phpx.
- Запустить обработчик в фоновом режиме.
- По завершении работы обработчика пометить соответствующий отчет флагом C (COMPLETED).
- Подождать пять секунд.
- Перейти к следующей итерации цикла.

Изменение флага состояния не только позволяет проинформировать пользователя о начале процесса обработки, но и предотвращает повторную обработку одного и того же запроса.

Пятисекундная задержка обеспечивает время для выполнения SQL-запроса. Длительность этой задержки можно задать по своему усмотрению.

Соответствующий сценарий-обработчик необходимо запускать в фоновом режиме, а не с использованием директив require или include. Он запускается как любой другой выполняемый файл. Тогда выполнению сценария обработки запроса не сможет помешать никакой сценарий-обработчик.

Для реализации этой процедуры можно использовать функцию proc_open(), подробная информация о которой содержится по адресу <http://www.php.net/manual/en/function.proc-open.php>. Эта функция позволяет запустить процесс и обеспечить некоторое управление им из кода приложения. Это позволяет создавать обработчики с правами чтения базы данных. Обработчику в качестве параметра командной строки передается имя выходного файла, а в качестве стандартных параметров функции — критерии отчета.

Выходной файл должен быть связан с экземпляром отчета. Например, для экземпляра отчета ID19039 выходной файл может иметь имя /generated/output19039.xml.

Процесс обработки отчета необходимо отслеживать, и при его завершении для данного экземпляра отчета устанавливать флаг C.

Основная идея состоит в отделении обработчиков от системы генерации отчетов. При этом минимизируется дублирование кода и повышается эффективность обработки.

Сценарии обработчиков

В предыдущем разделе отмечалось, что каждому типу отчета в предлагаемой архитектуре соответствует свой собственный сценарий обработчика. Например, отчету salesrep соответствует обработчик /scripts/handlers/salesrep.phpx. Этот сценарий запускается сценарием обработки отчета. При этом в качестве параметра командной строки используется имя выходного документа.

Этому сценарию необходимо также передать значение критериев в виде пар “ключ–значение”.

Передавать эти параметры в командной строке не удобно, поэтому сценарий обработки отчета передает их через стандартный поток ввода, как будто они вводятся с клавиатуры. Сигналом к запуску сценария и свидетельством завершения списка параметров является пустая строка.

Если сценарий обработчика запустить из командной строки, то команда его запуска будет выглядеть следующим образом.

```
# /scripts/handlers/salesrep.php /generated/output19039.xml
datefrom1=2004-03-01
dateto1=2004-03-31
datefrom2=2004-05-01
dateto2=2004-05-31
authors=31,14,12,11,15
publishers=
[пустая строка]
[здесь выполняется сценарий]
```

Все эти данные обеспечиваются сценарием обработки отчета. При запуске обработчиков выполняются следующие задачи.

- С помощью массива argv определяется имя выходного файла и создается сам файл.
- Через стандартный поток ввода stdin сценарий получает пары “ключ–значение” и помещает их в ассоциативный массив.
- Получив символ новой строки, обработчик игнорирует всю остальную информацию и выполняет SQL-запросы, необходимые для получения результата.
- Полученный результат преобразуется в форму XML-документа и сохраняется в заданном на первом шаге выходном файле.

Все очень просто. Обработчику даже не приходится обновлять базу данных. Он выполняет только считывание. Завершение работы обработчика является сигналом для сценария обработки о завершении формирования отчета.

Формат файла XML большой роли не играет. Для каждого отчета понадобится один сценарий его преобразования. Для рассмотренного выше примера структура XML может иметь следующий вид.

```
<results>
<rangeonesales>31319</rangeonesales>
<rangetwosales>33153</rangetwosales>
<percentuplift>5.856</percentuplift>
</results>
```

Полученный файл XML короток и ясен. Хотя входные критерии представляли собой достаточно сложную структуру данных, включающую два временных диапазона и список из пяти авторов, результаты выполнения отчета довольно просты: объем продаж за

первый период, за второй и разница между ними в процентах. Формат выходных данных не зависит от входных параметров. Это очень удобно, поскольку позволяет использовать универсальный сценарий преобразования, не зависящий от входных критериев.

Этот сценарий преобразования файла XML в более удобную форму будет рассмотрен в следующем разделе.

Страница Мои отчеты

Вернемся к странице, отображающей список всех отчетов с указанием их статуса.

На ней должны содержаться элементы управления для просмотра генерированных отчетов. При этом пользователь может выбрать формат просмотра отчета (HTML или PDF) либо просмотреть отчет в исходном XML-формате.

Необходимо также обеспечить возможность удаления отчета или его отправки по указанному электронному адресу.

Перечень возможных форматов вывода определяется доступными сценариями преобразования, речь о которых пойдет в следующем разделе.

Сценарии преобразования отчетов

Сценарии преобразования отчетов из формата XML в более удобный для пользователя формат являются интерактивными, т.е. выполняются в реальном времени, а их результаты можно просмотреть в Web-браузере. Они связаны со страницей **Мои отчеты**, а не вызываются напрямую пользователем.

Размещение этих сценариев должно обеспечивать удобство нахождения соответствующих трансляторов для каждого отчета. Например:

```
/translators/html/salesrep.php  
/translators/pdf/salesrep.php  
/translators/excel/salesrep.php
```

Входными данными для каждого сценария является соответствующий файл XML, а выходными — данные отчета в удобном формате. Например, для преобразования результатов отчета в формат PDF можно использовать следующую команду.

```
/translators/pdf/salesrep.php?filename=output19039.xml
```

Этот адрес URL генерируется непосредственно страницей **Мои отчеты**, которая определяет перечень возможных трансляторов.

Процесс преобразования форматов во многом зависит от конкретного отчета и объема обрабатываемых данных, а также от используемой схемы XML.

Приведем несколько полезных советов.

Создание отчетов в формате HTML

Формат HTML можно рассматривать как упрощенную форму формата XML, для которого все дескрипторы имеют предопределенное значение, поэтому для конвертирования XML-файла в формат HTML можно использовать листы стилей XSL. В PHP4 для этой цели использовался пакет Sablotron. Однако его использование вызывало массу неудобств. PHP 5 поддерживает модель DOM (Document Object Model), поэтому преобразование форматов выполняется с помощью функции взаимодействия между двумя объектами DOM.

```
$objDomXML = new DomDocument;  
$objDomXML->loadXML($strXML);
```

```
$objDomXSL = new DomDocument;
$objDomXSL->loadXML($strXSL);

$proc = new XSLTProcessor;
$proc->importStyleSheet ($objDomXSL) ;
$strHTML = $proc->transformToXML ($objDomXML) ;
```

Здесь создаются два объекта класса `DomDocument`, в которые загружаются соответствующие файлы XML и XSL (это возможно, поскольку XSL является еще одной формой XML).

Затем создается экземпляр утилиты класса `XSLTProcessor`. Этот класс выполняет XSL-преобразование, но не может использовать XSL в текстовом виде. Поэтому лист стилей XSL необходимо преобразовать в форму объекта модели DOM.

После этого можно использовать метод `transformToXML()`, в результате выполнения которого создается документ HTML.

Формирование отчетов в формате PDF

Файлы PDF можно сформировать разными способами. Если делать ставку на использование PHP, то для преобразования конкретной схемы DTD можно использовать класс `PDF` пакета R&OS (<http://www.ros.co.nz/pdf/>). Эта технология не зависит ни от каких внешних библиотек и не связана ни с какими лицензионными ограничениями.

Однако использовать этот подход не так просто, поскольку таблицы выходного файла приходится рисовать вручную.

Более простой способ состоит в использовании средств Apache FOP (<http://xml.apache.org/fop/>) (обеспечивающих стандартную схему DTD для представления PDF-документов).

Недостатком этой технологии является то, что она основана на Java. Поэтому для ее использования необходимо установить набор средств Java SDK.

Тем не менее, в этом нет большой проблемы — предлагаемая архитектура генератора отчетов позволяет запустить отдельный сервер приложений Java (например, Jakarta) через другой порт и таким образом обеспечить работу транслятора, позволяющего преобразовать отчет в PDF-формат.

Формирование электронных таблиц Excel

Электронные таблицы Microsoft Excel можно генерировать с помощью компонента `Spreadsheet_Excel_Writer` пакета PEAR (http://pear.php.net/package/Spreadsheet_Excel_Writer). Это естественный механизм для создания документов Excel, не требующий использования внешних библиотек или СОМ-объектов.

Пример использования генератора отчетов

Проиллюстрировать работу описанной системы достаточно сложно. Тем не менее рассмотрим от начала до конца пример ее использования, отслеживая процессы, выполняемые в фоновом режиме.

Пользователь заходит на начальную страницу со списком возможных отчетов и выбирает тип отчета, который нужно сгенерировать.

Сценарий `newreport.php` загружает соответствующий файл шаблона XML и обрабатывает поля ввода. Пользователь заполняет эти поля и отправляет форму. Этот же сценарий сохраняет введенную пользователем информацию в базе данных и создает

новый экземпляр отчета с флагом P (PENDING). Для пользователя отображается страница **Мои отчеты** со списком всех отчетов, включая текущий.

Работающий в фоновом режиме сценарий обработки подхватывает новую запись базы данных и присваивает ей статус I. Затем он находит соответствующий обработчик и запускает его как новый процесс. Обработчик анализирует входные параметры, формирует запросы к базе данных и генерирует результирующий файл в формате XML. По завершении этого процесса сценарий обработки присваивает этому отчету статус C.

После этого пользователь может просмотреть отчеты в одном из удобных форматов. Выбор формата просмотра, по существу, означает запуск сценария трансляции, входными данными которого является файл XML, а результатом — файл в нужном формате.

Визуализация

Зачастую отчеты удобно представлять в графическом виде. Например, длинный список пар “ключ–значение” удобнее представить в виде графика.

Если отчет генерируется в формате Excel, пользователь может сам построить нужные диаграммы. Однако для отчета в формате HTML это сделать гораздо сложнее.

К счастью, PHP позволяет довольно легко строить графики и диаграммы.

С помощью компонента `Image_Graph` пакета PEAR (http://pear.php.net/package/Image_Graph/) можно легко конвертировать строковые данные в диаграммы. Для этой цели подойдет и множество других пакетов.

Сценарий трансляции генерирует выходной файл в формате PNG. Этот файл сохраняется на диске и включается в результирующий HTML-документ.

Резюме

В этой главе вы узнали, как построить серьезную систему генерации отчетов для вашего приложения. Сначала было рассказано о том, как разрозненные запросы заказчика можно преобразовать в четкий набор целей и спроектировать систему, которая позволяет их достичь и сгенерировать соответствующий отчет.

Затем речь шла о разработке рабочей, надежной и расширяемой архитектуры для интеграции системы отчетов с приложением. Эта система позволяет строить множество разнообразных отчетов и выводить данные в различных форматах.

В следующей, заключительной, главе мы заглянем в будущее: в ней речь пойдет о том, как развивать свою карьеру PHP-профессионала.

26

Что дальше

Если вы до сих пор читали эту книгу, не обращаясь к компьютеру, пожалуйста, вернитесь хотя бы к некоторым из предыдущих 25 глав. Человеку свойственно обучаться. Вернитесь к одной из предыдущих глав, вызвавшей у вас наибольший интерес, и поэкспериментируйте с кодом. В данном случае как нельзя к месту поговорка: “Что посеешь, то и пожнешь”.

Это верно и в том случае, если вы скрупулезно прочли каждый абзац и запустили все приложения. Задайте себе вопрос: а что будет, если ... Вы не знаете ответа? Не можете найти его в документации? Тогда экспериментируйте. Теперь у вас есть соответствующие навыки по использованию языка PHP. Вы можете самостоятельно решить любую проблему.

Это очень важное замечание. Наличие технических навыков, умение протестировать свое приложение и знание многих аспектов программирования означает, что можно продолжать образование и самосовершенствование.

Мотивация

Не важно, как эта книга оказалась у вас в руках. Описанный в ней материал нетривиален, и вам удалось дойти до конца. Это означает, что вы “оторвали” время от общения со своей семьей, компьютерных игр, а может быть, даже и сна. Надеемся, что вы не пожалеете и сможете применить полученные знания на практике. Установите модуль PHPUnit, загрузите архивы PEAR, инсталлируйте и запустите необходимые приложения и проанализируйте их функциональность.

Человек никогда не перестает учиться. Каждая глава этой книги содержит интересный материал, и ни одна из них не является чисто академической. Эта книга написана профессионалами, и все изложенные в ней идеи используются в профессиональной среде.

Однако ничто не стоит на месте. Не останутся вечными и идеи, изложенные в этой книге. Поэтому продолжайте учиться и знакомиться с новыми ресурсами.

Ваша карьера как разработчика

Решив стать разработчиком программного обеспечения, вы обрекаете себя на вечное самосовершенствование. Техника развивается очень быстро, поэтому необходимо прилагать усилия по отслеживанию этих тенденций.

Однако как оценить свой уровень знаний? Нельзя знать все обо всем.

За пределами Web-приложений

Программирование на языке PHP — это не только управление сессиями, загрузка файлов и настройка конфигурационных переменных. PHP — это язык, который быстро развивается, удовлетворяя растущие потребности разработчиков.

Хотя PHP разрабатывался как язык создания приложений, он не только обеспечивает взаимодействие с браузером. Его можно использовать для написания приложений под Windows и Linux, взаимодействия с файловыми системами и практически любыми базами данных. Какие еще возможности предоставляет PHP? Перечислим некоторые из них.

- Преобразование баз данных.
- Обработка XML-файлов.
- Тестирование форм и Web-узлов, созданных с помощью других технологий.
- Обработка текстов, аналогичная возможностям Perl.
- Запуск стандартных утилит.

Даже если вам не приходится ежедневно создавать Web-узлы, некоторые навыки по работе с PHP будут вам полезны.

Жизненный опыт

Эти навыки очень важны при разработке любых приложений. Поэтому если уровень ваших знаний позволяет решить любую возникающую проблему, вы на вершине успеха. Речь идет не только о технических навыках. В процессе разработки приходится много общаться с заказчиками и сотрудниками, и даже в случае неудачного проекта вы получаете необходимый жизненный опыт.

Этот опыт чрезвычайно ценен. Вы можете долго изучать методологию разработки, в том числе экстремальное программирование, а в результате понять, как не нужно писать программы. Жизненный опыт помогает быстрее разобраться в новой ситуации.

Академические навыки

Разработка программного обеспечения — это не волшебство, а скорее искусство, “переходящее” в науку.

Хорошо написанный код соответствует известным шаблонам проектирования. Проблемам грамотной разработки программных систем посвящено множество книг, которые желательно изучить.

ЖИЗНЬ В СОЦИУМЕ

Своим существованием язык PHP обязан целому сообществу разработчиков, принимающих участие в его развитии и тестировании.

Разработчики делятся своими проблемами и предлагают друг другу их решение. Это определенная политика, помогающая развитию каждого отдельного проекта.

Участие в общих проектах, таких как PEAR, повысит ваши профессиональные навыки и позволит сэкономить время других разработчиков. Участие в коллективных проектах позволяет быстрее освоить новую технологию, поскольку не всегда существует необходимая документация по интересующим вас вопросам.

Резюме

Представленные в этой книге технические концепции достаточно апробированы на практике и полезны каждому разработчику. Опытным разработчикам наверняка известны хорошие и плохие реализации шаблона MVC, рефакторинга, моделей на языке UML и реализации коллекций.

Новые знания позволят вам улучшить свои приложения. Попытайтесь применить их в других областях и проектах.

Все мы живем на одной планете, поэтому давайте стараться минимизировать количество плохих программ.

И наконец, помните, что будущее — в движении.

Часть V

Приложения

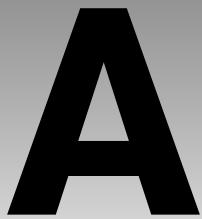
В ЭТОЙ ЧАСТИ...

Приложение А. Зачем использовать контроль версий

**Приложение Б. Интегрированные среды разработки
для языка PHP**

Приложение В. Практические советы по установке PHP

Приложение Г. Практические советы по установке PHP



Зачем использовать контроль версий

Зачастую разработчики боятся участия в больших проектах по множеству причин. Однако одна отличительная особенность таких проектов иногда может оказаться гораздо менее привлекательной, чем все остальные, — организация кода. Если шесть или семь разработчиков, два архитектора программного обеспечения и разработчики клиентской части приложения работают над одним и тем же проектом, то это может привести к возникновению серьезных проблем.

К счастью, в последние годы при выполнении большого проекта организовать программный код стало гораздо проще благодаря появлению простого в использовании программного обеспечения управления версиями.

В данном приложении вы познакомитесь с основными принципами контроля версий и увидите, как организовать стратегию контроля версий при выполнении проекта. После этого будет показано, как эти принципы применяются в самом популярном программном обеспечении контроля версий, а также как выбрать тот программный продукт, который подходит для конкретного проекта больше всего.

Принципы контроля версий

Контроль версий преследует две цели. Во-первых, при разработке необходимо избежать конфликта версий, который может возникнуть в том случае, когда несколько разработчиков работают над одним и тем набором файлов. Во-вторых, нужно обеспечить регистрацию изменений в наиболее важных из них.

Если разработчики будут обязаны как-то сообщать о работе с файлами из центрального хранилища, система контроля версий сможет сохранять записи о том, кто, какие файлы и в какое время использовал. В зависимости от используемого подхода, можно либо вообще запретить другим разработчикам использовать “занятые” в данный момент файлы, либо все же разрешить к ним доступ. Тогда придется обеспечить слияние всех сделанных изменений, когда все разработчики, которые работали с одним файлом, “вернут” его обратно в хранилище.

Ведение журнала файлов проекта предполагает необходимость сохранения предыдущих версий этих файлов. Каждый раз, когда файл возвращается в хранилище, система контроля версий помечает его как текущую версию. Так что любой, кто позже извлечет этот файл из хранилища, по умолчанию получит в свое распоряжение самую последнюю версию. Однако при этом по умолчанию используется предыдущая версия файла. Такой подход позволяет менеджеру проекта или ведущему архитектору системы не только отменить принятие новой версии, которая не оправдала ожиданий, но и без проблем отслеживать изменения между различными версиями. С точки зрения менеджмента такая возможность является очень важной.

Во всех системах управления версиями используется хранилище, которое предназначено для хранения копий файлов (обычно прямо на жестком диске, но иногда и как часть более сложной базы данных) и структуры каталогов проекта. Правила доступа к этому хранилищу определяются конкретной системой. Типичный сценарий использования хранилища будет рассматриваться в этом приложении ниже.

Параллельное и исключающее управление версиями

Системы контроля версий значительно различаются в реализации описанных выше принципов, а не только в подходах к извлечению файлов из хранилища.

В любой системе контроля версий при извлечении файла из хранилища разработчик всегда получает его последнюю версию. При этом соответствующая локальная версия заменяется последней версией.

Однако в тех системах, в которых реализован исключающий режим контроля версий (*exclusive versioning*), после извлечения файла из хранилища он сразу же блокируется. Если файл был извлечен одним разработчиком, другие разработчики по-прежнему могут получить его последнюю версию, но вместе с тем они не смогут с ней работать. Обычно при использовании такого подхода локальная копия файла становится доступной “только для чтения”. Конечно, подобная защита во многом является виртуальной, поэтому для ее эффективного применения между разработчиками должно быть достаточное взаимопонимание. Блокировка с файла снимается после того, как использующий его разработчик вернул файл на прежнее место. После этого хранилище обновляется для учета самой последней версии.

Параллельный контроль версий является совершенно другим подходом. При его использовании операция извлечения последней версии файла комбинируется с операцией его получения для работы таким образом, чтобы они, по существу, были одним и тем же действием. Другими словами, при работе с файлом все разработчики должны быть уверены в том, что ими используется его самая последняя версия. После того как каждый разработчик завершил свою работу, он возвращает свою версию файла обратно в хранилище. Именно в этот момент и происходит нечто интересное. Если один разработчик извлек файл после того, как другой разработчик сделал то же самое, и пытается вернуть его в хранилище после того, как другой разработчик вернул измененную версию файла, эти две переданные версии объединяются.

Пример параллельного управления версиями

Для прояснения вышесказанного рассмотрим пример. Ниже приведен файл `helloworld.php`, который выводит сообщение Здравствуй, мир! в окне браузера. Как можно увидеть, в строки кода добавлены их номера. Конечно, при сохранении файла номера строк сохранять не нужно.

```

1:  <?php
2:      $strToPrint = "Здравствуй, мир!";
3:  ?>
4:  <html>
5:      <body>
6:          <?=strToPrint?>
7:          <br /><br />
8:      </body>
9:  </html>

```

Будем считать, что этот файл имеет версию 1.0.

Предположим, что Джейн До и Джон До работают над проектом. Допустим, Джон работает в Нью-Йорке, а Джейн — в Лос-Анджелесе.

В какой-то момент состоялось обсуждение с заказчиком. Было решено изменить код файла `helloworld.php` таким образом, чтобы вместо строки Здравствуй, мир! выводилось сообщение До свидания, мир!. Менеджер Джона До поручил ему внести требуемые изменения в программный код.

При обсуждении заказчик также захотел, чтобы выводимый текст был снизу подчеркнут горизонтальной линией. Менеджер Джейн До поручил ей модифицировать код и добавить дополнительную линию.

При использовании исключающего контроля версий Джон и Джейн не смогли бы внести соответствующие изменения одновременно. Сначала Джон должен извлечь файл, внести изменения, а затем вернуть этот файл на прежнее место. После этого аналогичные действия должна выполнить Джейн.

При параллельном контроле версий такие ограничения не налагаются. Другими словами, допустим, что в 12 часов Джон извлекает последнюю версию файла, на данный момент 1.0. После этого он приступает к внесению изменений. В 12 часов 1 минуту Джейн также извлекает файл для работы и получает его самую последнюю версию. При этом последней версией по-прежнему является 1.0. Джон еще не вернул модифицированный файл. Джейн приступает к внесению своих изменений.

В 12 часов 5 минут Джон закончил свою работу. Код прекрасно работает, так что он решил разместить результаты своей работы в хранилище. Именно так он и сделал, так что в хранилище сохраняется переданная Джоном новая версия файла — 1.1, которая становится последней версией. Версия 1.1 выглядит следующим образом.

```

1:  <?
2:      $strToPrint = "До свидания, мир!";
3:  ?>
4:  <html>
5:      <body>
6:          <?=strToPrint?>
7:          <br /><br />
8:      </body>
9:  </html>

```

В 12 часов 9 минут Джейн также закончила свою работу. Она удовлетворена полученными результатами и решает разместить их в хранилище. Ее код имеет следующий вид.

```

1:  <?
2:      $strToPrint = "Здравствуй, мир!";
3:  ?>
4:  <html>
5:      <body>
6:          <?=strToPrint?>
7:          <br /><br />

```

```
8:      <hr />
9:    </body>
10:   </html>
```

Когда Джейн загрузила файл обратно в хранилище, система контроля версий обнаружила, что получена измененная версия 1.0, а не 1.1, которая на данный момент является последней. Тот факт, что рабочая версия была модифицирована разработчиком, обычно определяется по специальной строке. (Эта строка автоматически добавляется во все файлы с исходным кодом и обычно встраивается в дескрипторы комментариев, чтобы избежать конфликтов с компиляторами и интерпретаторами.) Эта строка создается и обновляется в хранилище и разработчиками обычно не модифицируется.

Теперь в хранилище нужно объединить изменения, внесенные Джоном в версию 1.0 (которая стала версией 1.1), и изменения Джейн в версии 1.0.

Подсистема хранения системы контроля версий определяет, что Джон внес следующие изменения. Стока 2

```
2: $strToPrint = "Здравствуй, мир!" ;
```

стала выглядеть следующим образом.

```
2: $strToPrint = "До свидания, мир!" ;
```

Было также определено, что после строки 7 Джейн вставила новую строку.

```
8: <hr />
```

После анализа внесенных изменений подсистема хранения систематически вносит модификации Джона и Джейн в последнюю версию (1.0). В результате объединения код получил следующий вид.

```
1: <?
2: $strToPrint = "До свидания, мир!" ;
3: ?>
4: <html>
5:   <body>
6:     <?=strToPrint?>
7:     <br /><br />
8:     <hr />
9:   </body>
10:  </html>
```

Как можно увидеть, изменения, внесенные обоими разработчиками, были успешно добавлены в новую версию файла `helloworld.php`. Теперь эта версия помечается как 1.2 и при обработке всех последующих запросов на получение последней версии предоставляется именно эта версия. Можно не сомневаться, что ни один из разработчиков не сможет лучше выполнить объединение изменений, чем было показано выше.

Конфликты, возникающие при параллельном контроле версий

Может возникнуть ситуация, когда два или больше разработчиков работают над одной и той же версией файла и вносят изменения, которые несовместимы друг с другом. В этом случае возникает конфликт.

Предположим, что было проведено еще одно совещание с заказчиком и были выдвинуты два новых требования: вместе с сообщением `До свидания, мир!` нужно выводить и время дня, а также добавить дату. Менеджер проекта поручил Джону добавить вывод времени дня, а Джейн — реализацию отображения даты.

Джон взял версию 1.2 и модифицировал ее следующим образом.

```

1:  <?
2:  $strTime = time("H:i:s");
3:  $strToPrint = "До свидания, мир, сейчас $strTime!";
4:  ?>
5:  <html>
6:    <body>
7:      <?=strToPrint?>
8:      <br /><br />
9:      <hr />
10:     </body>
11:   </html>
```

Джейн также воспользовалась версией 1.2 и изменила ее так.

```

1:  <?
2:  $strTime = time("Y-m-d");
3:  $strToPrint = "До свидания, мир, сегодня $strTime!";
4:  ?>
5:  <html>
6:    <body>
7:      <?=strToPrint?>
8:      <br /><br />
9:      <hr />
10:     </body>
11:   </html>
```

Сохранение внесенных изменений (в зависимости от того, кто из разработчиков справился с задачей быстрее) приведет к созданию версии 1.3. Что же произойдет, когда второй разработчик также сохранит свою модифицированную версию? В каждом случае в код добавляется новая строка и модифицируется уже существующая строка. Хотя система контроля версий может корректно скомбинировать две новые строки (просто объединив их), ей абсолютно не известно, как можно объединить изменения, внесенные в одну и ту же строку (в обоих случаях это строка 3). Любой разработчик на языке PHP сможет решить эту проблему следующим образом.

```
3:  $strToPrint = "До свидания, мир, сейчас $strTime $strDate";
```

Однако система контроля версий не обладает такими “интеллектуальными способностями” и просто инициирует возникновение конфликта. После этого она обратится к разработчику, который последним сохранил свои изменения и стал причиной конфликта, и предложит его разрешить. На практике будет создана временная новая версия файла 1.4, в которой будут содержаться детали конфликтной ситуации. Эта временная версия не станет рабочей до тех пор, пока конфликт не будет разрешен. Система управления версиями уведомит о возникшей проблеме разработчика, который сохранил свои изменения последним, и предложит ему отредактировать временную версию. Обязанность по разрешению конфликта возлагается именно на последнего разработчика, поскольку с точки зрения системы контроля версий именно он привел к его возникновению.

Созданная хранилищем временная версия 1.4 может выглядеть следующим образом.

```

1:  <?
2:  $strTime = time("Y-m-d");
<<<<< helloworld.php
3:  $strToPrint = "До свидания, мир, сейчас $strTime!";
=====
3:  $strToPrint = "До свидания, мир, сейчас $strDate!";
```

```
>>>>> 1.3
4:  ?>
5:  <html>
6:    <body>
7:      <?=strToPrint?>
8:        <br /><br />
9:        <hr />
10:      </body>
11: </html>
```

Как можно увидеть, система управления версиями отметила два различных варианта строки 3.

Приведенный пример иллюстрирует очень простой конфликт, который нужно разрешить. В данном случае достаточно обеспечить объединение изменений обоих разработчиков, а затем активизировать результирующую версию в хранилище.

Конечно, в реальной жизни разрешение конфликтов может оказаться гораздо более утомительной задачей. По существу, параллельное управление версиями является палкой о двух концах. Его полезно использовать, поскольку в этом случае разработчики могут одновременно работать с одним и тем же файлом. В то же время они должны быть готовы к разрешению конфликтов, которые могут возникнуть в процессе работы.

Выбор требуемой стратегии

На практике параллельное управление версиями в хорошо организованном проекте находит лишь ограниченное применение. Как настоятельно рекомендуется в этой книге, лучше всего разделить ресурсы проекта на несколько отдельных компонентов, каждый из которых рассматривался бы как один файл.

Поэтому необходимость работы двух разработчиков над одним файлом возникает достаточно редко. Если это все же потребовалось, можно рассмотреть возможность разделения данного файла на две меньшие части.

Более того, давайте снова вернемся к предыдущему примеру. Насколько вероятно, чтобы Джейн и Джону поручили реализацию таких двух очень похожих требований? С этой задачей без особых проблем может справиться и один разработчик. Следует отметить, что то же самое имеет место и при реализации более сложных требований. Зачастую один разработчик владеет целым компонентом и, как следствие, отвечает за внесение в него всех требуемых изменений.

И наконец, параллельное управление версиями позволяет разработчикам совместно участвовать в выполнении проекта без какого бы то ни было взаимодействия друг с другом. Короткий диалог между двумя разработчиками при исключающем контроле версий, когда требуемый файл был извлечен из хранилища и заблокирован (“Джейн, ты сейчас работаешь с файлом `helloworld.php`? Мне нужно внести в него небольшие изменения.”), на самом деле приводит к необходимости взаимодействия. Параллельное управление версиями не требует никакого взаимодействия, поэтому разработчики общаются друг с другом гораздо реже. А это не очень хорошо. Ни для кого не является секретом тот факт, что параллельный контроль версий часто используется при разработке программного обеспечения с открытым кодом, когда в этом процессе участвуют тысячи разработчиков, многие из которых никогда друг с другом не встречались. Тем не менее все они работают над одним проектом. Хотя разработка программного обеспечения с открытым кодом и представляет собой философию, которая заслуживает всяческого внимания, она все же приводит к возникновению некоторых побочных эффектов. Так, например, это привело к повышению сложности настройки модуля PHP на платформе UNIX.

Альтернативой является исключающий контроль версий, при котором двум разработчикам запрещено использовать один и тот же файл одновременно. После того как файл был извлечен из хранилища, для других разработчиков он блокируется до тех пор, пока первый разработчик не вернет его обратно. Именно поэтому конфликты, о которых говорилось выше, просто не могут возникнуть.

Конечно, исключающий контроль версий также имеет свои недостатки. Они проявляются даже в том случае, когда работы над проектом разделены так, чтобы двум разработчикам никогда не понадобилось использовать один и тот же файл одновременно. В качестве примера можно привести ситуацию, когда кто-либо из разработчиков пошел отдохнуть и случайно оставил какой-нибудь файл заблокированным. И именно с этим файлом понадобилось работать другому разработчику. Конечно, можно подойти к рабочей станции и сохранить данный файл в хранилище. Однако что делать, если отсутствующий разработчик еще не все закончил? Можно отменить операцию извлечения файла из хранилища, однако это приведет к потере всех сделанных им изменений. На самом деле эту ситуацию следует отнести к проблемам организации работ, которую можно решить путем внедрения подходящей политики.

Выбор требуемой стратегии контроля версий является очень важным. Пять или шесть разработчиков, выполняющих проект на языке PHP 5 в одном и том же офисе, могут без проблем воспользоваться какой-либо разновидностью исключающего управления версиями. А для проекта PHP 3, выполняемого тысячами разработчиков со всего мира, лучше всего подойдет параллельный контроль версий.

Топология контроля версий

Ниже вы кратко познакомитесь с несколькими программными пакетами, предназначенными для контроля версий. Однако перед этим нужно разобраться с тем, как этот подход используется в реальной жизни.

Обычно механизм контроля версий используется независимо от выбранного для разработки программного обеспечения, а также независимо от того, какой подход к управлению версиями вам больше нравится. Рассмотрим рис. A.1. На нем проиллюстрировано, что Джон, Джейн и Дэвид работают над одним и тем же проектом `foo`. Каждый из них имеет доступ только к своей рабочей станции.

Очевидно, что модуль PHP не запускается на каждой станции, так что все разработчики совместно используют специально выделенный для разработки сервер. Каждый разработчик использует свой собственный экземпляр виртуального сервера. Этот виртуальный сервер связан с отдельной копией проекта `foo`, соответствующей рабочему каталогу каждого пользователя на этом сервере.

Например, Джон пользуется ресурсом `http://john.projectfoo.example.com`, который связан с исходным кодом в его рабочем каталоге `/home/john/public_html/projectfoo` на сервере разработки. Аналогичные ресурсы имеются и для других разработчиков.

Имеется также тестовый сервер, который используется главным архитектором системы для тестирования и проверки ее последней версии. Этот сервер может применяться также и для внутренних демонстраций. Внешний тестовый сервер предназначен для презентации результатов заказчику.

Когда Джон, Джейн или Дэвид приступают к работе над программным кодом, они используют его копию из своего собственного рабочего каталога, к которому не имеет доступа ни один из остальных разработчиков. Эти рабочие каталоги расположены на

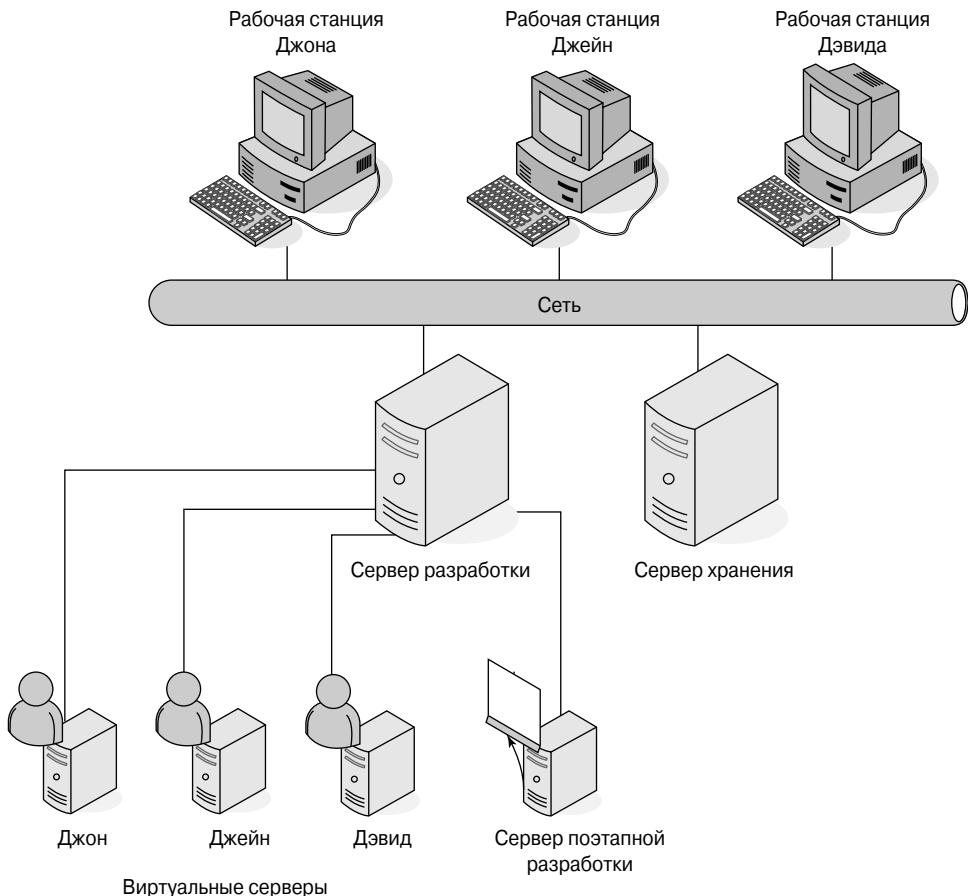


Рис. A.1.

сервере, так что для подключения сетевого диска на рабочей станции к каталогу на сервере используется пакет Samba (по адресу www.samba.org о нем можно получить дополнительную информацию). Каждый из разработчиков напрямую редактирует свои копии файлов, которые образуют их рабочее пространство. Ни один из файлов проекта не хранится на рабочей станции, однако благодаря использованию сетевого диска вся работа выполняется так же, как и с локальными файлами.

Однако Джон, Джейн и Дэвид должны использовать систему контроля версий. Это означает, что как только они начинают работать с каким-либо файлом, это событие должно регистрироваться соответствующим программным обеспечением, установленным на рабочей станции (например, Microsoft Visual SourceSafe). Кроме этого, с сервера хранения в рабочий каталог на сервере разработки должна загружаться последняя версия этого файла. Эта копия должна быть доступна для записи, чтобы все разработчики могли ее использовать. Если Джон, Джейн или Дэвид не зафиксировали факт начала использования файла перед его открытием в среде разработки, то он оказывается доступным только для чтения. В этом случае в него нельзя внести никаких

изменений. Другими словами, после того как файл был извлечен из хранилища одним из разработчиков, никто другой работать с ним не сможет.

После завершения работы с файлом его последнюю версию нужно записать в хранилище. После этого при извлечении этого файла из хранилища другие разработчики будут получать его последнюю версию со всеми сделанными изменениями.

Поскольку Джон, Джейн и Дэвид работают над проектом одновременно, им нужно периодически выполнять операцию получения последних версий. Это означает, что последние версии всех файлов будут локально скопированы в соответствующие рабочие каталоги на сервере разработки, даже если часть из них в данный момент для работы не требуется. Это является важным по двум причинам. Иногда может потребоваться некоторая дополнительная функциональность, которая была кем-то из разработчиков недавно добавлена, чтобы проверить корректность работы какого-либо компонента. Кроме того, такой подход позволяет разработчикам быстро увидеть, над чем работают их коллеги, и при необходимости сообщить об ошибках или высказать критические замечания.

И наконец, ведущий архитектор системы Поль может скопировать все последние версии в каталог на тестовом сервере (а не в свой собственный). Это позволит оценить состояние всего проекта в целом, основываясь на содержимом хранилища, и при необходимости высказать замечания в адрес группы разработчиков.

При использовании какого-нибудь другого пакета контроля версий описанная топология может немного измениться. Например, после установки системы CVS разработчики вряд ли будут использовать какое-либо программное обеспечение на своих рабочих станциях. Скорее, они будут устанавливать терминальное соединение с сервером разработки (например, с помощью SSH) и запускать на нем клиента CVS.

Стоит отдельно также отметить, что роль сервера хранения также может несколько варьироваться. В системе CVS для обмена данными применяется архитектура "клиент/сервер". В то же время пакетом Visual SourceSafe просто применяется совместно используемый раздел данных на сетевом диске. В результате отдельный физический сервер хранения может и не понадобиться. Для этого можно адаптировать и сервер разработки.

Программное обеспечение контроля версий

Для реализации в рамках проекта контроля версий существуют различные пакеты. В этом небольшом разделе вы познакомитесь с тремя из них, узнаете об их достоинствах и недостатках, а также увидите, как в этом программном обеспечении реализована базовая функциональность и основные принципы контроля версий.

В данном приложении невозможно подробно описать процесс установки и использования систем управления версиями. В Web имеется очень много документации по этому вопросу. Однако материал данного раздела должен обеспечить обоснованный выбор специализированного пакета, который больше всего подходит для вашего проекта.

Microsoft Visual SourceSafe

Пакет SourceSafe используется уже достаточно давно и в настоящее время входит в комплект поставки Visual Studio .NET. На первый взгляд, может показаться, что он не применим при разработке программ на языке PHP, однако на самом деле с ним стоит познакомиться.

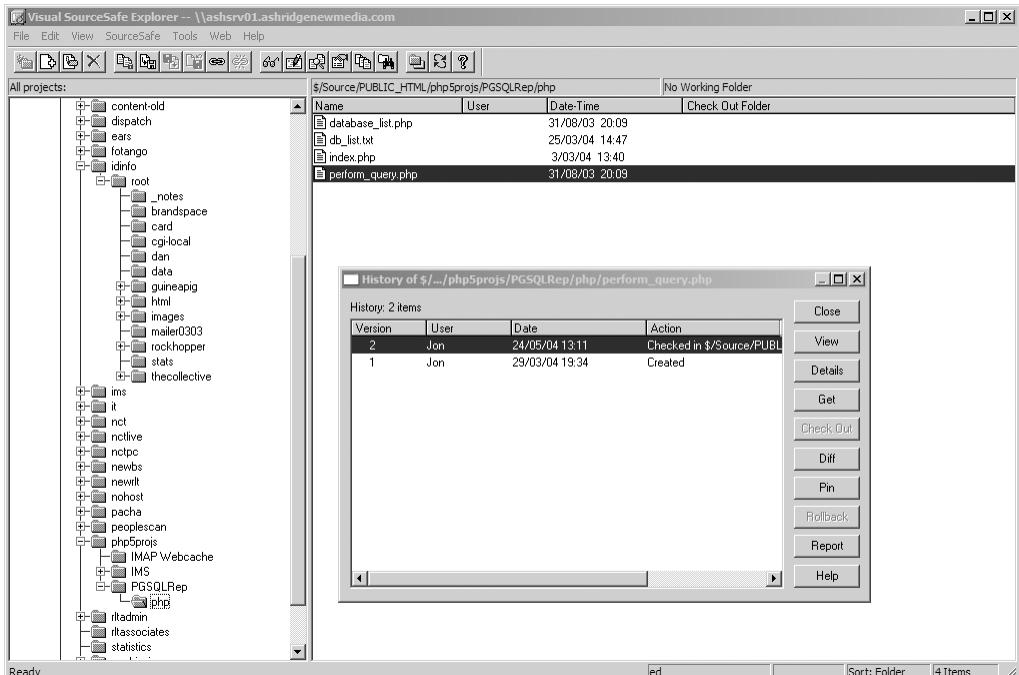


Рис. А.2.

Хотя пакет SourceSafe и не является бесплатным, входящий в него клиент для системы Windows очень удобен и прост в использовании. Несмотря на то, что приверженцы языка PHP никогда не избегали трудностей, однако именно сейчас нужно разобраться с тем, как можно использовать клиента данной системы контроля версий. Пакет SourceSafe настолько просто использовать, насколько можно представить. На рис. А.2 показано его типичное главное окно. Как нетрудно заметить, оно очень напоминает диалоговое окно программы Проводник Windows.

Однако для поддержки хранилища пакетом SourceSafe используется разделяемое дисковое пространство, а не архитектура “клиент/сервер”. Это приводит к возникновению типичных проблем, связанных с применением совместно используемых баз данных Access, и снижению производительности, если используются большие файлы со множеством исправлений. Существует также и ряд неудобных моментов, которые сводятся к различиям между системами Windows и UNIX, связанных, например, с учетом регистра символов в именах файлов.

Пакет SourceSafe поддерживает только исключающий контроль версий. Другими словами, если кем-то из разработчиков файл извлечен из хранилища, то этого больше никто не сможет сделать. Как и во многих других пакетах, которые поддерживают исключающий контроль версий, в системе SourceSafe блокировка локальной копии файлов осуществляется путем установки атрибута “только для чтения”. Поскольку в данном случае используется клиент Windows, разработчики практически всегда должны обращаться к сетевому диску, что не очень эффективно. При этом рабочие станции функционируют в качестве средства передачи данных, что является далеко не самым удачным подходом.

Пакет SourceSafe хранит свою базу данных в специальном формате, поэтому соответствующую область хранения нельзя автоматически использовать на тестовом сервере. Однако этой системой поддерживается богатый набор синтаксических конструкций командной строки, на основе которых можно конструировать автоматизированные сценарии загрузки последних версий на тестовый сервер. Естественно, эти сценарии нужно запускать в среде Windows, поскольку в состав пакета клиенты системы UNIX не входят.

Имейте также ввиду, что при выполнении небольших проектов стоимость лицензии может оказаться чрезмерной. В то же время понадобится заплатить лишь за копии SourceSafe. Поскольку этим пакетом используется разделяемое дисковое пространство, для его работы сервер не требуется. Более того, сетевой диск можно развернуть и в системе Linux с установленным пакетом Samba.

CVS

Аббревиатура CVS означает Concurrent Versioning System (система параллельного контроля версий). Как следует из названия, эта система является одним из наиболее известных средств контроля версий, которым могут пользоваться разработчики. Пакет CVS разработан в рамках общей лицензии GNU, поэтому он является полностью бесплатным и доступным для использования.

Хранилище системы CVS расположено в папке, которая практически полностью зеркально отражает реальную структуру каталогов разрабатываемого в рамках проекта приложения. При этом для хранения вспомогательных данных о файлах (кроме самых последних версий) в них используются специальные строки, скрытые файлы каталогов и несущественные модификации имен. Это принципиально отличается от принципов хранения файлов в системе SourceSafe.

Клиенты CVS могут подключиться к хранилищу одним из двух способов: напрямую (если оно расположено на локальной машине) либо с использованием протокола pserver. Теоретически можно воспользоваться и совместно используемым сетевым диском (как в системе SourceSafe), чтобы предоставить доступ к хранилищу первым способом, однако лучше с помощью протокола pserver развернуть систему CVS на платформе “клиент/сервер”.

Система CVS в основном используется на платформе UNIX и обычно устанавливается на той же машине, на которой содержатся локальные копии файлов. В предыдущем примере топологии система CVS запускалась каждым пользователем путем установки соединения с сервером разработки через командную строку. Для пользователей UNIX такой подход является предпочтительным. Для тех же, кто предпочитает работать в более дружественной среде Windows (как при использовании SourceSafe), можно воспользоваться клиентом WinCVS, который предоставляет практически ту же функциональность, что и при работе в системе UNIX. На рис. А.3 показано типичное диалоговое окно клиента WinCVS, интерфейс которого очень напоминает SourceSafe.

Разрешение конфликтов в системе CVS выполняется относительно просто. Сначала выполняются все возможные операции объединения, а затем пользователю системы предоставляется новая версия, в которой проиллюстрирован конфликт. Разработчик модифицирует эту версию файла, разрешает конфликт и передает его обратно системе. После этого этот файл сохраняется в качестве самой последней версии.

Система CVS может оказаться прекрасным выбором, однако поддерживаемый ею режим параллельного контроля версий может понравиться далеко не всем. Соответ-

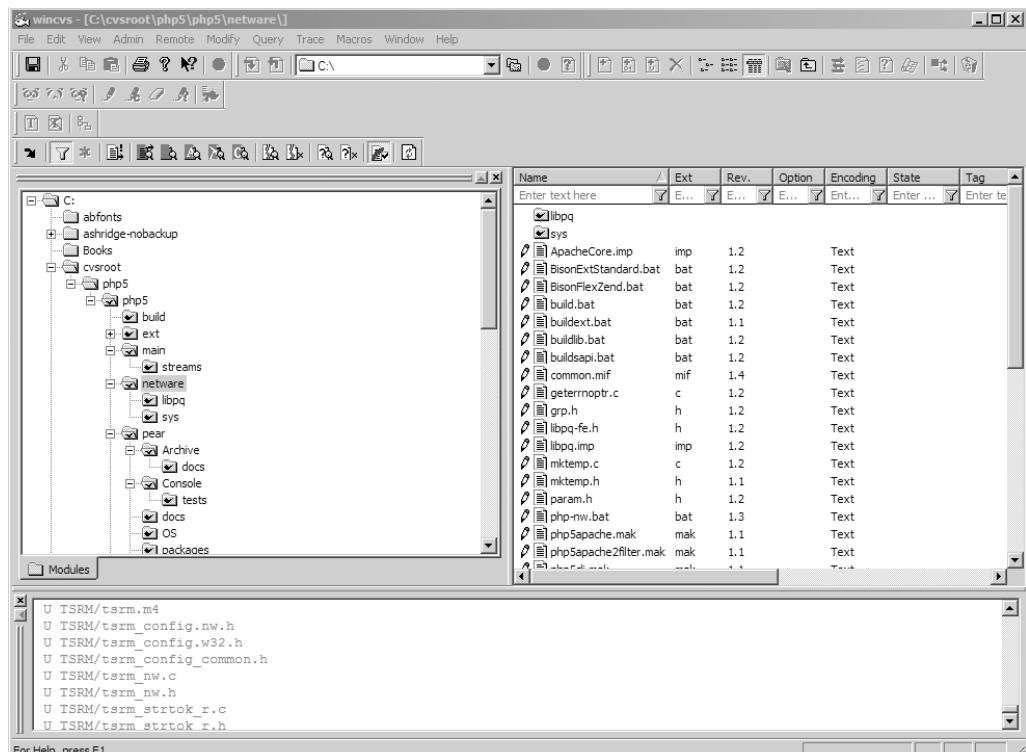


Рис. А.3.

ствующие причины уже рассматривались в этом приложении выше. Внимательно проанализируйте, подходит ли эта система для конкретного проекта, и если это не так, оцените, может ли ее использование привести к возникновению каких-либо проблем.

RCS

Предшественник системы CVS и составляющий ее основу пакет RCS поддерживает модель исключающего контроля версий. Он появился около двадцати пяти лет назад. Для тех проектов, для которых система SourceSafe является слишком дорогостоящей, медленной или чрезмерно громоздкой, система RCS является достаточно привлекательной альтернативой, хотя в настоящее время она поддерживается гораздо хуже, чем CVS.

Кроме того, в системе RCS не поддерживаются многие полезные функции CVS, которые напрямую не связаны с параллельным управлением версиями. К таким функциям относится группирование файлов в одну коллекцию (которая интерпретируется как один объект) и автоматическое определение состояния файлов. Фактически система RCS представляет собой менее развитую версию CVS. Она способна удовлетворить некоторых разработчиков, однако для многих из них может показаться слишком неудобной для повседневного использования.

Другие системы контроля версий

В настоящее время появляются и другие системы контроля версий, заслуживающие пристального внимания. Наиболее мощной из них является программа Subversion Грега Штейна (Greg Stein), которую можно найти по адресу <http://subversion.tigris.org>. В отличие от CVS, система Subversion основана на использовании устаревшего протокола RCS. Она была разработана “с нуля” и обладает некоторыми достаточно интересными отличительными особенностями, такими как использование HTTP в качестве транспортного протокола и поддержка двоичных разностей для уменьшения размера хранилища.

Дополнительные приемы контроля версий

О системах контроля версий можно сказать гораздо больше, чем в данном небольшом приложении. Как и при описании языка PHP, этот вопрос можно более подробно изложить лишь в отдельной книге. Однако существует несколько интересных приемов, которыми можно воспользоваться в контексте выбранной для конкретного проекта системы контроля версий. Эти приемы и будут кратко рассмотрены ниже.

Ветвление

Ветвление предполагает разделение одного проекта на два параллельных проекта. Это чрезвычайно полезный прием для повторного использования кода.

Предположим, что вы разрабатываете две системы управления содержимым для двух отдельных заказчиков и их функциональность практически полностью совпадает. Имеются лишь незначительные или эстетические различия. Очевидно, что можно повторно использовать код одной системы для разработки другой. Однако при таком подходе существует одна проблема. Если сделать важное изменение в первой системе, то вполне возможно, что его придется перенести и на второй проект. Однако вовсе необязательно, чтобы все изменения во втором приложении потребовалось распространить и на первый проект, поскольку они могут иметь отношение лишь к некоторым дополнительным настройкам.

Решением описанной проблемы является ветвление. Используя первый проект для создания второго проекта, можно обеспечить, чтобы система контроля версий автоматически распространяла все изменения, сделанные в первом проекте, на код второго проекта. При этом все оставшиеся изменения, которые будут выполнены во втором проекте, останутся уникальными.

Ветвление можно использовать также для поддержки двух различных направлений развития системы. Например, одна группа разработчиков может обеспечивать поддержку текущей версии программного продукта, а другая группа будет заниматься разработкой его следующей версии. В этом случае ветвление позволяет получить моментальный снимок системы и в дальнейшем продолжать его развитие в двух различных друг от друга направлениях. При этом все существенные изменения, внесенные в код приложения до ветвления, будут распространены на обе версии, которые развиваются независимо друг от друга.

Тегирование

Не путайте этот прием с использованием дескрипторов, которые добавляются системой CVS в отдельные файлы для служебного использования. В данном случае речь идет о применении дополнительного атрибута к данной версии каждого файла, что позволяет создавать отдельные версии программного продукта.

Например, может потребоваться создать бета-версию приложения. Для этого в один пакет понадобится объединить почти все файлы проекта. Однако может оказаться, что в этот пакет нужно добавить различные версии каждого файла: версию 1.3 одного файла, версию 1.5 другого файла и т.д. Выполнив тегирование требуемых версий файлов, системе контроля версий можно сообщить о необходимости формирования определенной версии программного продукта на их основе. После этого можно удалить дескрипторы и, поскольку они являются вспомогательными, в любой момент можно обеспечить доступ к определенной версии приложения: бета-версии, версии 1 и т.д.

Комментарии

Одной из возможностей систем контроля версий, о которой часто забывают, является возможность использования комментариев при сохранении файлов в хранилище. Если разработчик решил сохранить файл в хранилище, ему будет предложено добавить комментарии. Проверить доступность такой возможности и затем ее использовать очень важно, поскольку даже короткое описание, такое как “исправленная ошибка 21301”, может существенно упростить дальнейшую работу. А при анализе истории определенного файла эта информация может оказаться еще полезной.

Двоичные файлы

При добавлении в хранилище двоичных файлов соблюдайте осторожность. Хотя на первый взгляд идея хранения всего проекта в одном месте может показаться очень удачной, многие системы контроля версий в этом случае оказываются совсем неэффективными. При обработке нескольких десятков изображений JPEG скорость их работы может существенно снизиться.

Подумайте, не существует ли другого способа решить эта проблему. Например, проанализируйте все GIF-файлы, в которых содержатся заголовки страниц. Можно ли их размещать на страницах с помощью одного сценария PHP, который с помощью функции `imagefttext()` по требованию может генерировать требуемый заголовок? Такой подход имеет очень важное значение, поскольку двоичные файлы представляют собой один из примеров, когда использование системы контроля версий может оказаться абсолютно неприемлемым.

Резюме

В этом приложении были рассмотрены основные принципы и методология, положенные в основу механизма контроля версий, а также их роль при выполнении больших проектов. Вы познакомились с различиями между параллельным и исключающим управлением версиями. Кроме того, была кратко рассмотрена топология типичной инфраструктуры контроля версий, а также способы ее реализации с помощью нескольких популярных пакетов. И наконец, вы узнали о нескольких дополнительных приемах, которые можно использовать вместе с выбранной системой контроля версий и тем самым повысить эффективность работы над проектом.

Б

Интегрированные среды разработки для языка PHP

В процессе программирования на языке PHP многие разработчики предпочитают использовать свой любимый редактор, а для отладки кода применяют собственные подходы. В то же время следует заметить, что в мире PHP использование интегрированных сред разработки (IDE – Integrated Development Environment, или среда разработки – Development Environment (DE)) еще не нашло такого широкого применения, как, например, при использовании Visual Studio .NET или ASP .NET. И это трудно объяснить. Возможно, это обусловлено тем, что многие разработчики не рассматривают PHP 5 как полнофункциональный язык программирования (авторы надеются, что после прочтения этой книги вы измените свое мнение на противоположное).

В любом случае можно с уверенностью утверждать, что PHP 5 представляет собой мощный и универсальный язык программирования, для которого существует ряд интегрированных сред разработки. Если вы раньше не использовали подобных средств, это приложение именно для вас. В IDE встроено много полезных функций, которые заслуживают всяческого внимания. Конечно, наиболее значимой из них является возможность сложной отладки программного кода.

Хотя в одном приложении просто невозможно разместить исчерпывающее описание каждой интегрированной среды разработки, здесь все же имеет смысл разобраться, что же можно ожидать от подобных средств. В любом случае, для того чтобы воспользоваться каким-либо из специализированных пакетов, придется подробно познакомиться с его справочным руководством. В этом приложении более подробно будет рассмотрен пакет Zend Studio, поскольку он является наиболее мощным продуктом, предназначенным для работы на языке PHP. Кроме того, вы кратко познакомитесь и с другими программными продуктами.

Выбор IDE

Основная задача среды IDE заключается в предоставлении разработчику всех средств, необходимых для эффективной разработки рабочих приложений. Подобную поддержку можно реализовать различными способами, которые по-разному представлены в каждом программном продукте. Какое бы программное обеспечение вы ни использовали, перед разработкой сначала нужно решить, позволяет ли выбранная среда разработки получить все необходимые преимущества. Другими словами, нужно оценить, упрощает ли используемая среда IDE разработку приложения. Если да, то выбранный программный продукт можно развернуть и использовать. Если же нет, нужно продолжить поиск.

При оценке выбранной среды разработки нужно помнить об одном нюансе. Он связан с тем, что как и при использовании любого другого программного обеспечения, нужно потратить определенное время, чтобы достаточно глубоко осознать все предоставляемые встроенные возможности. Поэтому на решение этой задачи очень важно потратить столько времени, сколько потребуется. В следующих разделах содержится информация, которая сможет вам в этом помочь.

Zend Studio Client

Первой интегрированной средой разработки, с которой стоит познакомиться, является Zend Studio. При этом следует учитывать тот факт, что для ее приобретения потребуется \$249. Конечно, перед приобретением какого-либо продукта сначала с ним нужно подробно познакомиться. Пробную версию Zend Studio можно найти по адресу <http://www zend com/store/products/zend-studio.php>.

В момент написания данной книги текущей версией Zend Studio была 3.5.0. Она предоставляет полную поддержку языка PHP 5. Пакет Zend Studio можно использовать на платформах Windows, Linux и Mac.

Если вы не знакомы с IDE, то, на первый взгляд, такие пакеты могут показаться слишком сложными. Однако главное не забывать о том, что подобные программные продукты предназначены для упрощения работы, а не для создания проблем. Оболочка Zend Studio — это гораздо больше, чем просто IDE. Она предоставляет также и много других возможностей. Например, в состав серверной версии входит скомпилированная версия модуля PHP и Web-сервера Apache. В данном приложении мы в основном сосредоточимся на знакомстве со средой разработки Zend (Zend Development Environment — ZDE), которая облегчает разработку программного кода и управление им. На рис. Б.1 показано основное окно ZDE, которое появится на экране после загрузки архивного файла приложения из Web, его установки и запуска.

Перед выполнением каких-либо действий выберите команду Tools⇒Preferences и познакомьтесь с параметрами, установленными по умолчанию. После этого можно приступить к практическому использованию среды разработки. Вместе с оболочкой ZDE поставляется очень много документации, поэтому мы не будем подробно рассматривать все ее возможности. Вместо этого давайте лучше познакомимся с наиболее важными функциями ZDE. С остальными вопросами вы без особых проблем разберетесь самостоятельно.

Управление файлами и проектами

В основном окне оболочки ZDE можно найти две панели, позволяющие управлять файловой системой и проектами PHP. Их можно увидеть в левой верхней части экрана

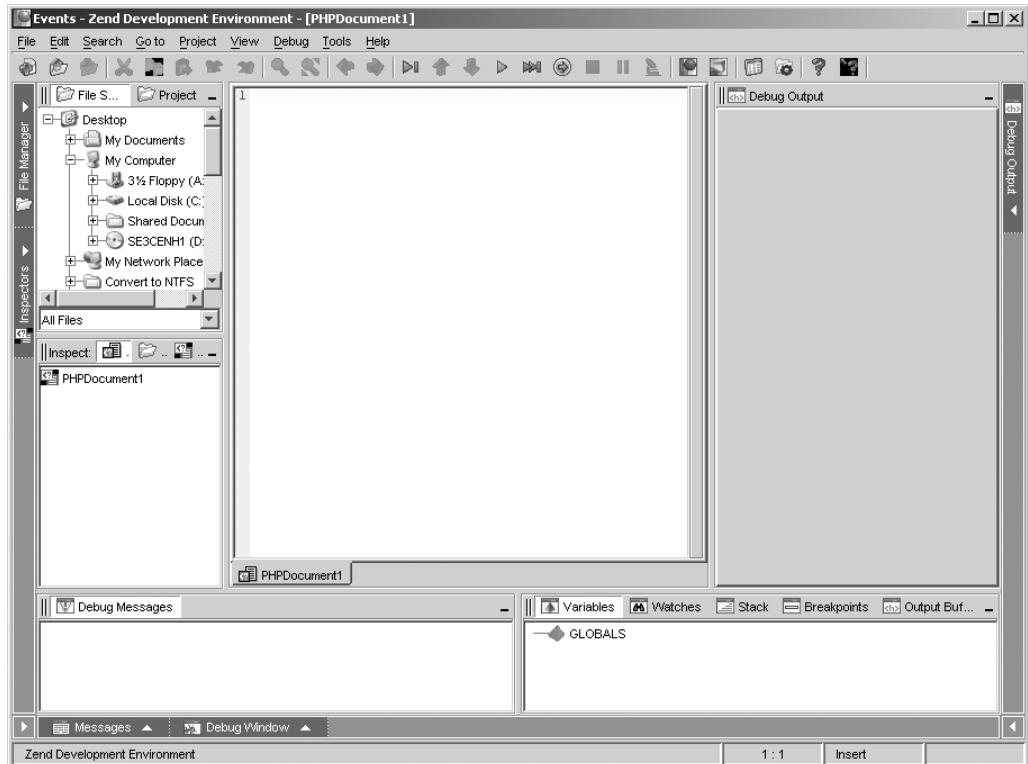


Рис. Б.1.

на рис. Б.1, при этом активной является панель **File System**. Ее можно использовать для поиска файлов с помощью раскрывающегося списка, который расположен сразу под ней. Этот элемент управления позволяет выполнять фильтрацию.

С помощью панели **Project** можно перемещаться между проектами и выполнять различные действия, такие как добавление файлов в проект или удаление их из проекта. Следует также заметить, что с помощью команды **Project** основного меню можно создавать, открывать, сохранять и закрывать проекты.

Проекты позволяют задать рабочее окружение для файлов в контексте конкретного проекта. Например, при разработке нового Web-узла все нужные файлы можно сгруппировать в один проект и таким образом обеспечить эффективную работу с ними.

Редактирование кода

Вы вправе ожидать, что среда разработки должна предоставлять определенные возможности по редактированию кода. Именно так и есть. В большом окне редактирования Zend Studio можно легко набирать и редактировать код (см. рис. Б.1). Кроме базовых возможностей, разработчику предоставляется также и много дополнительных режимов. Один из них связан с возможностью автоматического завершения фрагмента кода как на языке PHP, так и HTML. В этом режиме на экране отображаются всплывающие меню с параметрами функций, объявленными функциями, ключевыми словами и константами. Пример такого меню показан на рис. Б.2.

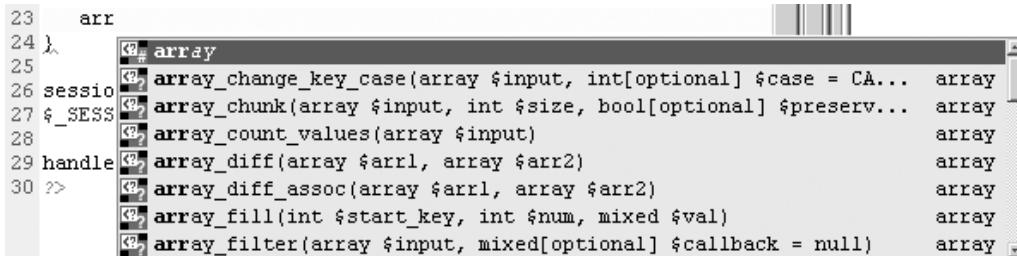


Рис. Б.2.

Конечно, режим завершения кода можно настроить, как и другие режимы среды разработки. Соответствующие параметры можно найти, выбрав команду Tools⇒ Preferences⇒Code Completion (диалоговое окно свойств показано на рис. Б.3).

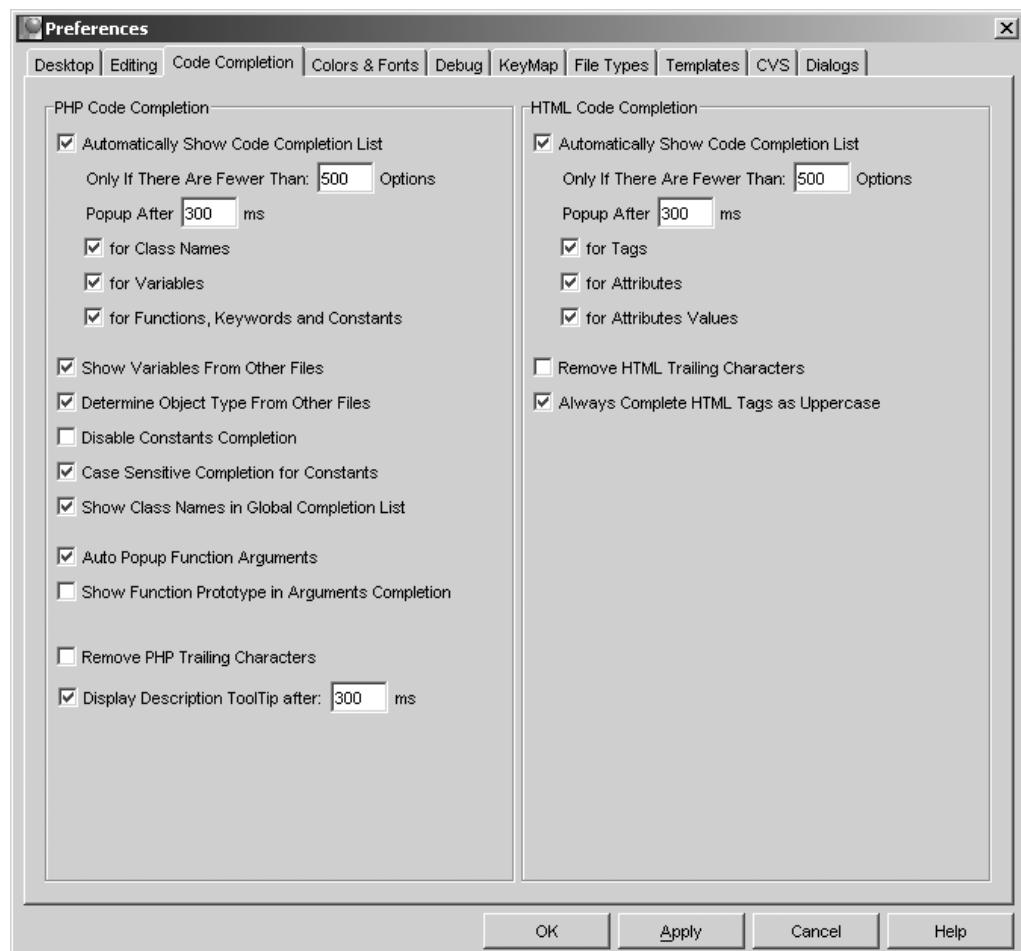


Рис. Б.3.

В окне свойств **Preferences** содержится и много других конфигурационных параметров, используемых, например, для настройки режима редактирования и отладки.

Пакет ZDE предоставляет также возможность структурирования программного кода. Можно либо вручную создавать отступы в процессе его набора, либо отформатировать весь код за один раз. Эта прекрасная возможность способна существенно облегчить жизнь. При переходе на страницу свойств **Editing** диалогового окна **Preferences** вы получите доступ к параметрам редактирования. Один из наиболее полезных режимов, который можно активизировать на этой вкладке, позволяет выполнить поиск парных скобок, как показано на рис. Б.4.

```
function secure_handler(){
    if ($_SESSION['name'] == "David"){
        $this->handled_event();
    }else{
        echo "Sorry $_SESSION[name] you are not
    }
}
```

Рис. Б.4.

Размещение курсора непосредственно перед начальным элементом или сразу же после завершающего элемента приводит к выделению соответствующего элемента. В этот момент можно воспользоваться комбинацией клавиш <Ctrl+M>, чтобы перейти к соответствующему элементу и не тратить время на прокрутку экрана. Для того чтобы познакомиться с другими комбинациями горячих клавиш, выберите в главном меню команду **Go to**.

Шаблоны позволяют повысить скорость выполнения повторяющихся действий. Многие из них поддерживаются по умолчанию, однако вы можете добавить свои собственные шаблоны или модифицировать уже существующие. Доступные шаблоны можно увидеть на странице свойств **Templates** диалогового окна **Preferences**. Каждому шаблону соответствует специальное условное обозначение. Например, на рис. Б.5 показаны результаты ввода в окно редактирования строки **my**.

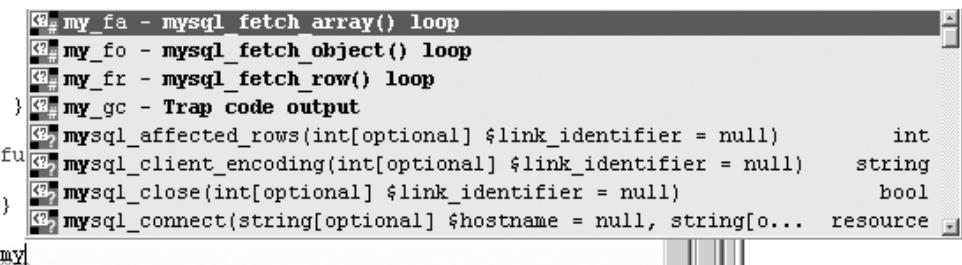


Рис. Б.5.

На приведенном рисунке в четырех верхних строках представлены шаблоны, входящие в комплект поставки оболочки ZDE. При щелчке на одном из них связанный с шаблоном код будет автоматически добавлен в окно редактирования. На рис. Б.6 показаны результаты щелчка на втором параметре.

Доступны и многие другие возможности редактирования кода, начиная с горячих комбинаций клавиш, которые можно использовать для ускоренного набора дескрип-

```

while ($row = mysql_fetch_object($query)) {
}

```

Рис. Б.6.

торов HTML, и заканчивая выделением синтаксических ошибок. Единственный способ ознакомления с ними заключается в их практическом использовании. Кроме того, при изучении отличительных особенностей среды разработки хорошим подспорьем окажутся справочные файлы. Так что найдите и используйте их. А теперь рассмотрим другие особенности оболочки ZDE.

Инспектирование кода

Как можно увидеть на рис. Б.1, в левой нижней части диалогового окна ZDE содержится панель **Inspect**. На ней расположены три инструмента: **File**, **Project** и **PHP**. С помощью первой кнопки можно получить доступ к списку всех элементов файла, который в данный момент загружен в окно редактирования (рис. Б.7).

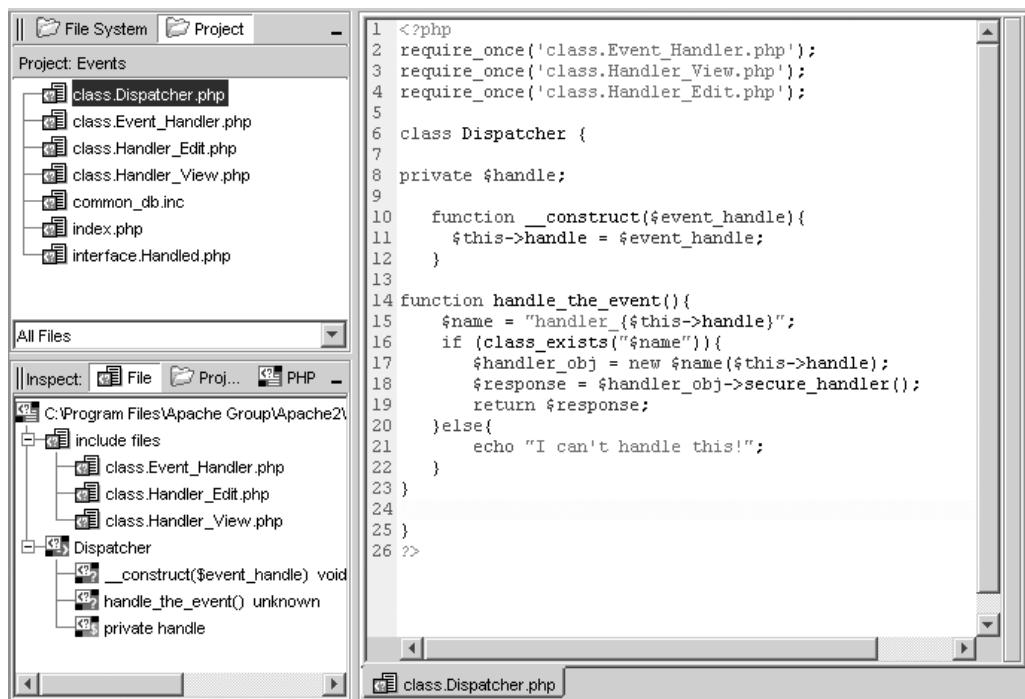


Рис. Б.7.

Этот список можно использовать для навигации по программному коду. При двойном щелчке в окне инспектирования на любом из этих элементов курсор переместится в соответствующее место в исходном коде.

При щелчке на вкладке **Project** можно просмотреть всю структуру текущего проекта, а при необходимости любой файл или элемент. При щелчке на этом файле или элементе он будет загружен в область редактирования.

И наконец, при щелчке на третьей кнопке **PHP** панели **Inspect** можно получить перечень всех функций PHP и описание их синтаксиса. Конечно, иногда о конкретной функции требуется получить более подробную информацию. Для этого в окне инспектирования нужно выделить требуемую функцию, а затем нажать клавишу <F1>. В результате в окно браузера будет загружена более детальная информация о функции. Естественно, этот же прием можно использовать и в области редактирования. Для этого просто выделите требуемую функцию и нажмите клавишу <F1>.

Отладка с использованием ZDE

В данном случае предполагается, что все необходимое программное обеспечение (т.е. был загружен и установлен клиент Zend Studio) запущено на одной машине, и вы можете отлаживать приложения локально. Следует заметить, что вполне возможно развернуть и многопользовательское окружение. Для этого на центральном, т.е. нелокальном, сервере потребуется установить пакет Studio Server. Для получения более подробной информации по этому вопросу обращайтесь к справочному руководству.

Одно из самых существенных преимуществ использования среды разработки заключается в возможности применения современных средств отладки. Для решения этой задачи можно прибегнуть и к изнурительному анализу журналов регистрации, однако такой подход выглядит устаревшим при его сравнении с более мощными и гибкими современными методами. При использовании внутреннего отладчика среды разработки основное внимание можно уделить тому, что происходит в программном коде. В частности, отладчик должен обеспечивать выполнение следующих задач.

- Мониторинг хода выполнения программы.
- Предоставление окна отладки.
- Предоставление окна с отладочными сообщениями.
- Генерация выходной информации.

Проанализируем эти задачи более подробно. Первый вопрос, на который нужно найти ответ, связан с тем, как перемещаться по коду в процессе его отладки. Для перемещения от одной точки кода к другой оболочка ZDE предоставляет различные способы. Некоторые из них предполагают установку точек прерывания, а другие позволяют выполнить программу пошагово. На панели инструментов основного диалогового окна Zend Studio можно найти набор различных стрелок (рис. Б.8), которые могут использоваться для пошагового прохода, перемещения между точками прерывания или простого выполнения программного кода.

Обратите внимание на самую правую кнопку приведенной панели инструментов. Она позволяет запустить анализатор кода, предназначенный для поиска ошибок в коде, открытом в окне редактирования. Для того чтобы увидеть, как он работает, внесите в код ошибку и щелкните на соответствующей кнопке.



Рис. Б.8.

После приостановления выполнения кода можно получить важную информацию о его состоянии. Правда, для этого придется обратиться к специальному окну отладки (*Debug Window*). По умолчанию оно распо-

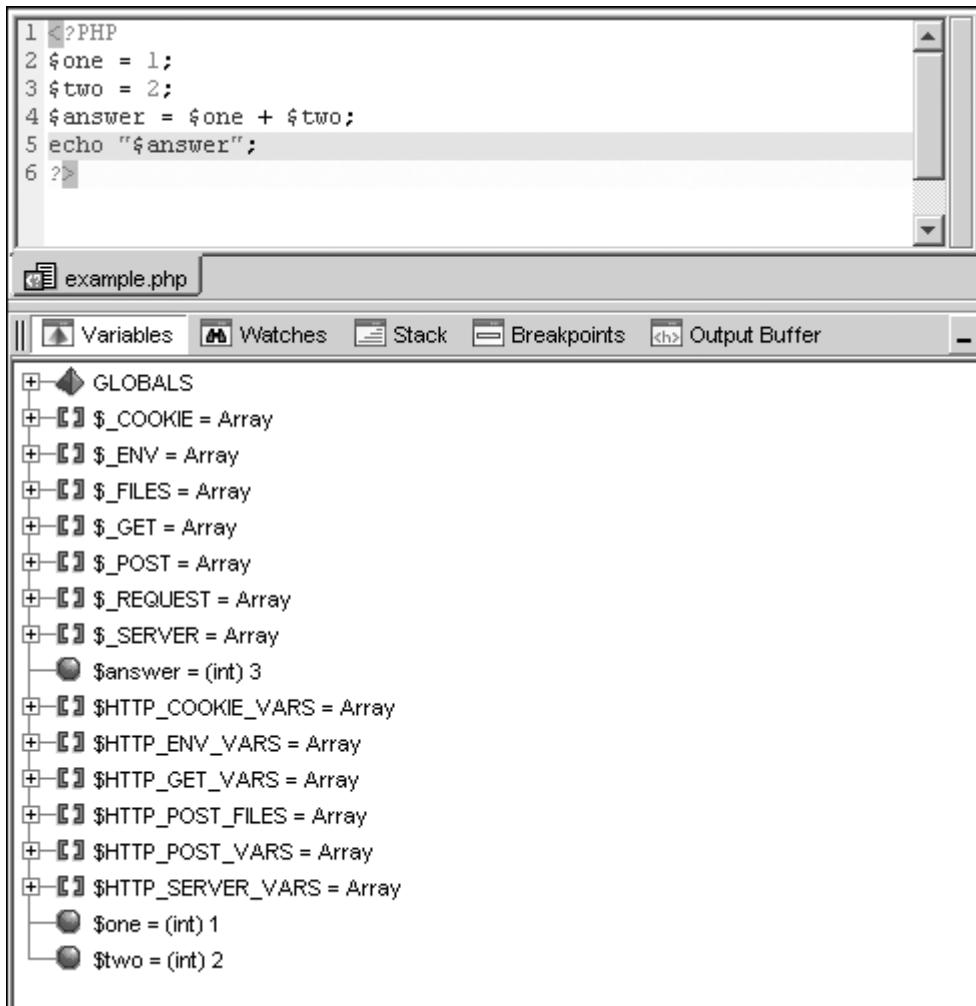


Рис. Б.9.

ложено в правой нижней части основного окна оболочки ZDE. Для выполнения мониторинга и управления процессом отладки проекта можно воспользоваться пятью вкладками. В окне отладки можно просмотреть значения переменных, позиции просмотра и точек останова, а также выполнить трассировку стека и просмотр буфера. Каким средством лучше воспользоваться, зависит от конкретной ситуации.

Предположим, что выполняется пошаговое выполнение очень простого сценария PHP. Тогда во вкладке **Variables** можно найти информацию обо всех переменных, в том числе и глобальных (рис. Б.9).

Обратите внимание, что в окне редактирования выделена пятая строка. Это обусловлено тем, что отладчик остановился именно на этой строке. Значения, которые можно увидеть во вкладке **Variables**, содержатся в соответствующих переменных в точке останова. Если вам известен фрагмент кода, который нужно проверить, можно

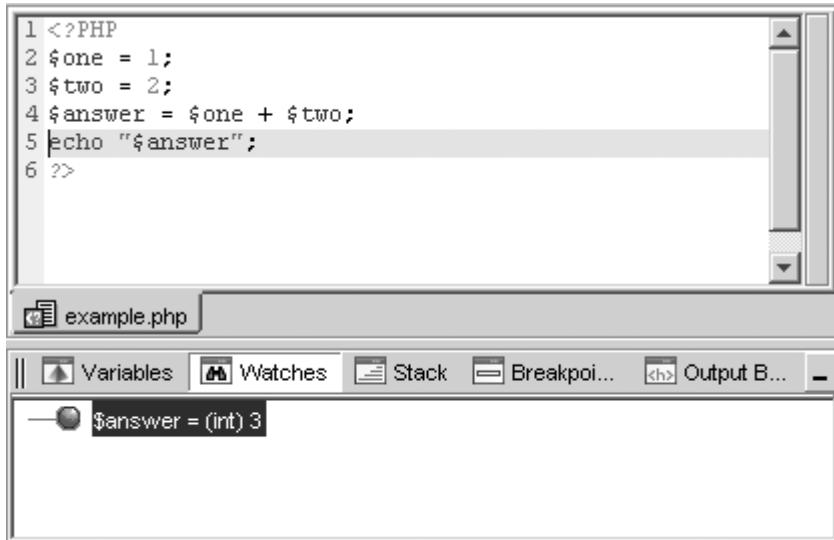


Рис. Б.10.

перейти именно к нему, а затем установить курсор в строку, которую нужно отладить. Щелкните на кнопке Go to Cursor (пятой, слева направо на рис. Б.9), чтобы перейти к строке, в которой находится курсор.

Следующей вкладкой является **Watches**. Может потребоваться, чтобы отладчик отслеживал состояние различных фрагментов кода. Тогда для них нужно установить режим наблюдения, или трассировки. В приведенном примере мы выделили переменную \$answer и выбрали команду Debug⇒Add Watch. В результате в процессе отладки можно получить окно, показанное на рис. Б.10.

Отслеживание состояния стека также является важной задачей, при решении которой можно определить, что же происходит при выполнении программного кода. Вкладка **Stack** позволяет увидеть текущее положение в стеке программы. Эта возможность может оказаться очень полезной, поскольку если в какой-то части кода вызывается функция, то управление из текущей позиции сценария передается вызываемой функции. Если в этой функции используются другие функции, то управление передается им, соответственно. Возможность точного определения текущей точки выполнения и является той особенностью, которая предоставляется отладчиком ZDE.

На рис. Б.11 показан пример содержимого вкладки **Stack**, которое можно использовать для отладки простого сценария, содержащего две функции.

Для того чтобы обеспечить остановку отладчика в том месте программного кода, где это нужно, необходимо установить точки останова. Точки останова — это простые ограничительные метки, которые сообщают отладчику о том, что выполнение сценария нужно приостановить. После приостановления выполнения можно принять решение о том, какие последующие отладочные действия лучше осуществить. В приведенном на рис. Б.11 примере было установлено две точки останова. Для этого понадобилось щелкнуть на номере строки в окне редактирования. После щелчка соответствующая строка выделяется розовым цветом (хотя вы не можете увидеть цвет на приведенном рисунке, все же на нем без проблем можно заметить выделенные строки). Перейдя во вкладку **Breakpoints**, можно проверить правильность установки точек прерывания (рис. Б.12).

The screenshot shows a code editor window with the file `example.php` containing the following PHP code:

```

1 <?PHP
2 function level_1(){
3     echo "This is level 1 in the stack!";
4     level_2();
5 }
6
7 function level_2(){
8     echo "This is level 2 in the stack!";
9 }
10 level_1();
11 ?>

```

Below the code editor is a toolbar with tabs: Variables, Watches, Stack, Breakpoints, and Output Buffer. The Stack tab is selected, displaying the current call stack:

- level_2() C:\Documents and Settings\David Mercer\My Documents\example.php line 8
- level_1() C:\Documents and Settings\David Mercer\My Documents\example.php line 4
- main() C:\Documents and Settings\David Mercer\My Documents\example.php line 10

Рис. Б.11.

The screenshot shows a code editor window with the file `example.php` containing the same PHP code as in Figure B.11. In this version, the lines `level_2();` at line 4 and `echo "This is level 2 in the stack!";` at line 8 are highlighted in gray.

Below the code editor is a toolbar with tabs: Variables, Watches, Stack, Breakpoints, and Output Buffer. The Stack tab is selected, displaying the current call stack:

- C:\Documents and Settings\David Mercer\My Documents\example.php, line:4
- C:\Documents and Settings\David Mercer\My Documents\example.php, line:8

Рис. Б.12.

The screenshot shows the Zend Studio interface. In the top-left pane, there is a code editor with the following PHP script:

```

1 <?php
2 ob_start();
3 print "I should show up in the output buffer window";
4 $buffer = ob_get_contents();
5 ob_end_clean();
6 print "$buffer";
7 ?>

```

The code editor has a tab labeled "example.php*". Below the code editor is a toolbar with tabs: Variables, Watches, Stack, Breakpoints, and Output Buffer. The "Output Buffer" tab is selected. In the main workspace below the toolbar, the text "I should show up in the output buffer window" is displayed.

Рис. Б.13.

И наконец, если в сценарии используются функции буферизации языка PHP, то разработчику очень пригодится вкладка *Output Buffer*. В этой вкладке можно увидеть содержимое буфера. Пример содержимого вкладки *Output Buffer* показано на рис. Б.13.

Стоит обратить внимание также на окно *Debug Messages*, которое по умолчанию можно найти слева от окна отладки. В нем приводятся уведомления, предупреждения и ошибки, генерируемые в процессе выполнения сценария. При двойном щелчке на выбранном сообщении осуществляется переход на строку в окне редактирования, при выполнении которой оно было сгенерировано.

И наконец, в окне *Debug Output* размещаются выходные данные сценария. Это окно можно найти в правой части основного окна оболочки ZDE. По содержимому окна *Debug Output* можно определить, корректные ли данные генерируются вашим сценарием.

Пакет Zend Studio предоставляет и другие возможности по отладке приложений. Например, в удаленном режиме можно отлаживать приложение после его размещения на Web-узле. Такой подход имеет ряд преимуществ, поскольку при его использовании можно отлаживать работающее приложение. А зачастую применить другой подход просто невозможно. Следует отметить, что результаты работы сценария можно отображать в окне браузера. Для использования всех перечисленных возможностей на Web-сервере должен быть запущен пакет Zend Server. Однако в данном приложении этот вопрос рассматриваться не будет.

После описания основных функциональных возможностей пакета Zend Studio, связанных с отладкой приложений и редактированием их программного кода, стоит повторить еще раз: загрузите его пробную версию и попробуйте оценить ее возможности на практике. Перед завершением ознакомления с программным продуктом Zend Studio давайте посмотрим, какие еще средства предлагает компания Zend.

Другие средства Zend Studio

Компания Zend предлагает разработчикам профайлер (доступ к которому можно получить с помощью команды Tools главного меню оболочки ZDE), который предназначен для анализа производительности приложений. Путем поиска проблемных

областей профайлер позволяет получить ответ на вопрос, как повысить эффективность функционирования приложения. При этом разработчику на различных вкладках предоставляется самая различная информация, начиная с круговых диаграмм, на которых содержатся данные о времени выполнения отдельных сценариев, и заканчивая диаграммами со статистикой о различных вызываемых функциях и времени, прошедшем между отдельными вызовами.

Кроме того, среда разработки Zend предоставляет возможность интегрирования системы контроля версий CVS. Более подробную информацию по этому вопросу можно найти в справочном руководстве.

И наконец, при желании можно установить комплект средств обеспечения безопасности. Тогда к разрабатываемому программному обеспечению можно применять электронное лицензирование. Этот набор средств содержит два основных компонента: Encoder и License Manager. Первый компонент просто создает зашифрованные двоичные файлы, которые впоследствии можно распространять. Такой подход позволяет защитить права разработчиков и предотвратить несанкционированное копирование или модификацию приложения. Компонент License Manager позволяет создавать лицензионные ключи, с использованием которых на применение программного продукта можно налагать определенные ограничения в соответствии с выбранным критерием.

На этом краткое знакомство с возможностями интегрированной среды разработки Zend Studio Client будет завершено. Ниже будут рассмотрены другие средства поддержки разработки приложений, которые имеются на рынке в настоящее время.

Komodo

Пробную версию программного продукта Komodo компании Active State можно найти по адресу <http://www.activestate.com/Products/Komodo/>. Профессиональная версия, которую можно использовать в коммерческих проектах, обойдется вам в \$295. Так что перед покупкой этого пакета стоит сначала познакомиться с соответствующей пробной версией. Исторически так сложилось, что систему Komodo можно было использовать на платформах Linux и Windows, однако самую последнюю версию 2.5 можно развернуть и на платформе Solaris.

Пакет Komodo представляет собой среду разработки на основе открытого кода, которая кроме языка PHP поддерживает также языки Perl, Python, Tcl и XSLT. Однако в момент написания данной книги в состав этого пакета еще не входил отладчик PHP 5 . Так что при отладке приложений на языке PHP придется ограничиться его более ранними версиями. Самую последнюю информацию о пакете Komodo можно получить на соответствующем Web-узле.

Управление проектами

Среда разработки Komodo предоставляет диспетчер проектов (Project Manager), который можно использовать для выполнения самых разнообразных действий. По умолчанию он располагается в левой части основного диалогового окна. Доступ к командам работы над проектами можно получить с помощью команды Project главного меню. С помощью диспетчера проектов можно выполнять в том числе следующие действия.

- Создавать и открывать проекты.
- Добавлять файлы и папки.
- Добавлять команды, шаблоны, адреса URL, Web-службы и диалоговые окна.
- Манипулировать проектами, а также сохранять или закрывать их.

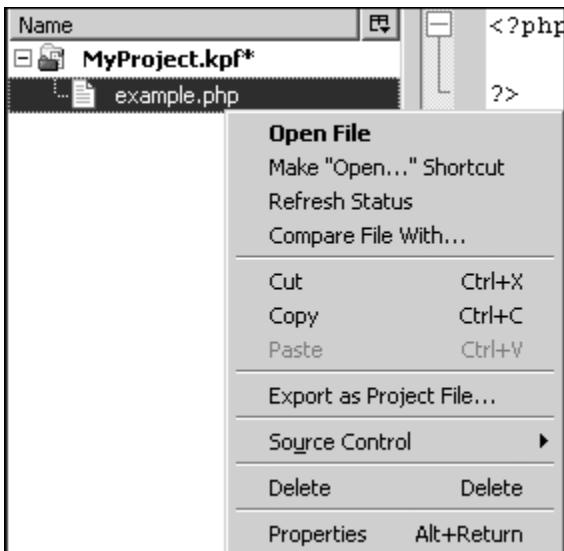


Рис. Б.14.

Кроме того, щелкнув правой кнопкой мыши на элементе проекта, можно получить доступ к контекстному меню (рис. Б.14).

Перечень команд контекстного меню зависит от типа выбранного элемента проекта.

Редактирование кода

Кроме стандартных отступов и выделения цветом редактор кода среды разработки Komodo обладает и некоторыми интересными особенностями. Например, в этом редакторе можно сворачивать отдельные разделы программного кода и таким образом использовать его различное представление. Окно редактирования со свернутым кодом показано на рис. Б.15.

The screenshot shows the Komodo IDE editor window displaying a PHP script. The code is as follows:

```
<?php
Function one () {
Function two () {
Function three () {
    echo "I am the third function in my script!";
}
?>
```

The 'Function one ()', 'Function two ()', and the entire 'Function three ()' block are collapsed, indicated by a minus sign icon to the left of the code. The expanded part of the code is 'echo "I am the third function in my script!"';.

Рис. Б.15.

При работе на языке PHP редактор кода позволяет также использовать функцию автозаполнения для классов, функций и переменных. При вводе ключевого слова new на экране отображаются классы в текущем и включаемых файлах. При этом методы классов отображаются при вводе операции ->. Аналогично, пользовательские функции,

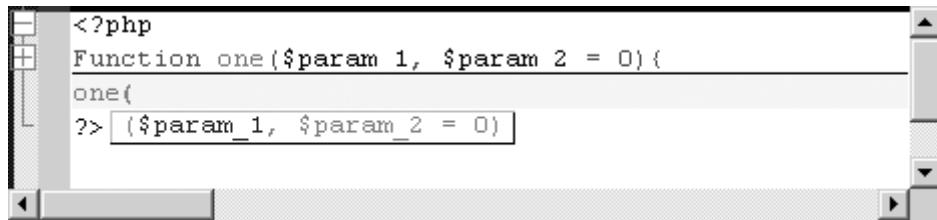


Рис. Б.16.

а также функции PHP будут выведены на экран после совпадения начальных символов в их имени с первыми четырьмя введенными символами.

Еще одна удобная особенность редактора связана с отображением *подсказок* (в них содержится перечень параметров, которые можно использовать для данной функции). На рис. Б.16 показана подсказка для пользовательской функции.

Редактор среды разработки Komodo обладает и многими другими отличительными особенностями, большинство из которых достаточно интуитивно и, как следствие, не требует какого бы то ни было отдельного описания. Вот их краткий перечень.

- Поддержка списка последних использованных файлов.
- Отслеживание измененных файлов.
- Предварительный просмотр в браузере.
- Отслеживание синтаксиса в фоновом режиме.
- Комментирование программного кода.

Отладка с помощью пакета Komodo

Мы не будем подробно рассматривать возможности отладчика Komodo, поскольку в момент написания этой книги он еще не поддерживал PHP 5 . Для того чтобы обеспечить работоспособность отладчика, входящего в комплект поставки среды разработки Komodo, необходимо выполнить определенную настройку параметров. Соответствующие подробные инструкции можно найти в документации. За исключением вышеуказанного, полнофункциональный отладчик Komodo предоставляет следующие возможности.

- Управление точками прерывания.
- Пошаговое выполнение программ.
- Просмотр значений переменных в процессе выполнения приложений.
- Просмотр стека вызовов.
- Возможность передачи сценариюм входных данных.
- Возможность добавления параметров командной строки.

Конечно, в комплект поставки пакета Komodo входит и набор других средств поддержки редактирования кода и его отладки. Ниже они будут кратко рассмотрены.

Другие средства Komodo

Поскольку пакет Komodo позволяет добавить в проект набор таких элементов, как папки, файлы, команды, шаблоны и т.д., то он обеспечивает также и хранение этих элементов, что упрощает получение к ним доступа. Доступ к этому хранилищу можно

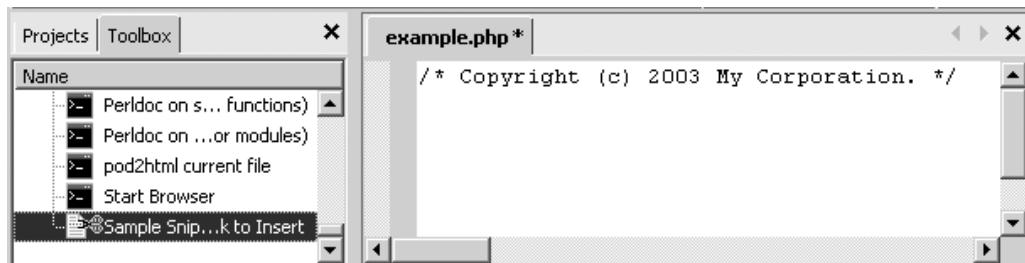


Рис. Б.17.

получить либо с помощью вкладки **Toolbox**, расположенной в левой части основного диалогового окна, либо с помощью команды **Toolbox** главного меню. Пример вкладки **Toolbox** показан на рис. Б.17. В данном случае при двойном щелчке на требуемом элементе соответствующий фрагмент кода будет добавлен в окно редактирования.

Среду разработки Komodo можно интегрировать с системой CVS и Perforce. (Perforce является программным продуктом, предназначенным для управления конфигурацией программного обеспечения (SCM — Software Configuration Management).) Для того чтобы воспользоваться этой возможностью, выберите в главном меню команду **Edit⇒Preferences⇒Source Code Control⇒CVS**. При этом на экране появится диалоговое окно, показанное на рис. Б.18. (В данном случае видно, что система CVS еще не была установлена.) В этом диалоговом окне при необходимости можно загрузить систему CVS или указать путь к соответствующему выполняемому файлу. Обратите внимание, что в дереве категорий конфигурационные параметры для системы Preforce расположены чуть-чуть ниже.

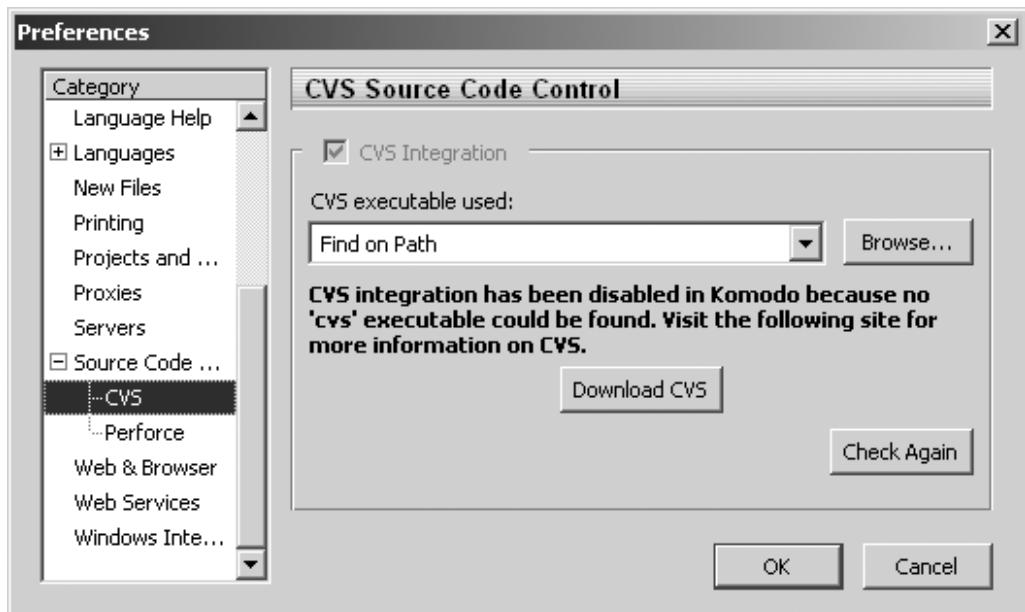


Рис. Б.18.

На этом краткое знакомство со средой разработки Komodo будет завершено. Вам осталось только загрузить пробную версию и познакомиться с ней более подробно. С точки зрения стоимости этого пакета следует заметить, что, по существу, за дополнительную плату разработчику предлагается функциональность, которая используется не так уж и часто. Например, некоторые средства, входящие в комплект поставки Komodo (например, Perl Dev Kit), предназначены исключительно для языка Perl. Вполне естественно, что это отражается на общей стоимости пакета Komodo, однако никак не расширяет возможностей разработчиков на языке PHP.

Другие IDE и редакторы

Для профессионального разработчика на языке PHP доступны и другие среды разработки. Некоторые из них оказываются более полезными, чем другие. Все зависит от задач, которые необходимо с их помощью решить. Например, средство PHPEdit для системы Windows предоставляет достаточно мощный редактор, а также отладчик. Этот пакет можно бесплатно загрузить с Web-узла <http://www.waterproof.fr/products/PHPEdit>.

Вот некоторые из основных возможностей, предоставляемых пакетом PHPEdit.

- ❑ Выделение синтаксиса.
- ❑ Генерация подсказок.
- ❑ Интегрированный отладчик PHP.
- ❑ Генератор справочных файлов.
- ❑ Поддержка настраиваемых горячих комбинаций клавиш.
- ❑ Поддержка более 100 команд, которые можно использовать для написания сценариев.
- ❑ Поддержка клавиатурных шаблонов.
- ❑ Генератор отчетов.
- ❑ Поддержка дополнительных модулей.

Ниже перечислены другие среды разработки и редакторы, знакомство с которыми может оказаться весьма полезным.

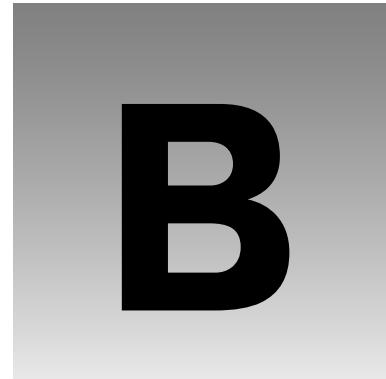
- ❑ PHPEclipse, http://www.phpeclipse.de/tiki-view_articles.php.
- ❑ EditPlus, <http://www.editplus.com/>.
- ❑ NuSpherePHPEd, <http://www.nusphere.com/products/index.htm>.
- ❑ PHPCode, <http://www.phpide.de/>.

На этом мы завершим краткий обзор средств разработки приложений на языке PHP. При этом следует заметить, что если не были рассмотрены доступные параметры всех вышеперечисленных пакетов, то это вовсе не означает, что они не заслуживают самого пристального внимания. Например, пакет PHPEd предоставляет самые разнообразные возможности, связанные с автоматическим завершением фрагментов кода, отладкой, профилированием и развертыванием приложений, а также позволяет выполнить много других действий. Так что с ним также имеет смысл познакомиться.

Резюме

В этом приложении были кратко рассмотрены доступные в настоящее время интегрированные среды разработки, а также основные возможности, которые они предоставляют при разработке приложений на языке PHP. В частности, был рассмотрен пакет Zend Studio Client, который предназначен для поддержки PHP-приложений, а также Komodo, представляющий собой программный продукт с открытым кодом, поддерживающий PHP 5.

Исследование преимуществ оболочек IDE является очень важным, особенно с точки зрения повышения производительности процесса разработки при их использовании. Поскольку приобретение средства разработки, подобного описанным выше, требует определенных денежных затрат, то перед окончательным выбором стоит потратить время на тщательное изучение отличительных особенностей каждого из соответствующих программных продуктов.



Настройка производительности PHP

Сценарии PHP являются чрезвычайно быстрыми. Фактически без особых проблем можно добиться того, чтобы PHP-приложения (как простые, так и сложные) были более динамичными, чем приложения Java, ASP, ColdFusion и ASP .NET.

Однако иногда высокая скорость является наиболее приоритетным требованием, степень реализации которого определяет общий успех проекта. В этом случае даже несущественный прирост (или уменьшение) производительности может привести к успешности всего проекта в целом (или к его краху). С этой точки зрения вопрос планирования производительности приложения является чрезвычайно важным.

Язык PHP стоит использовать для разработки и по другим причинам. Например, от сторонних разработчиков вы можете получить в свое распоряжение не очень удачно реализованный компонент, и вам может понадобиться повысить общую производительность системы без дополнительных финансовых затрат на приобретение нового оборудования. Хотя использование чужого кода редко доставляет удовольствие, особенно в тех случаях, когда его авторы не прочитали данной книги, вы будете удивлены, когда узнаете, насколько просто выиграть несколько секунд.

В данном приложении вы научитесь находить различные узкие места и узнаете, как их можно обойти. После этого вы познакомитесь с приемами правильного проектирования и кодирования, которые позволят добиться оптимальной эффективности и скорости, а также со способами, с помощью которых можно повысить производительность уже существующего кода.

Проблемы производительности

Первое уведомление о том, что приложение работает слишком медленно, вы наверняка получите от какого-нибудь пользователя. К сожалению, в этом “контексте” пользователи очень редко оказываются полезными. Возглас “Это работает так мед-

ленно!”, прозвучавший в офисе, практически ни на йоту не продвинет вас к решению проблемы. Поэтому придется провести небольшое исследование.

Типы узких мест, связанных с производительностью

Давайте ненадолго вернемся к основам Web-программирования и вспомним структуру HTTP-запроса GET или POST. Когда клиентский Web-браузер генерирует запрос, он устанавливает соединение с портом 80 Web-сервера или с его портом 443 при установке SSL-соединения. Эти действия являются блокирующими, т.е. при их выполнении браузер не может выполнять никаких других действий до тех пор, пока соединение не будет успешно установлено. Однако на практике большинство современных браузеров разрешает пользователям отменять попытку установки соединения.

При использовании быстрых каналов связи соединение устанавливается в течение всего лишь нескольких долей секунды. Если же сервер слишком перегружен, то установка соединения с ним потребует дополнительных усилий. Это приведет к тому, что выполнение других приложений или процессов замедлится (необязательно только данного приложения).

После успешной установки соединения Web-браузер не ожидает никакого ответа, поскольку протокол HTTP этого не требует, а сразу отправляет очень маленький пакет с запросом, который обычно занимает не больше нескольких байтов. Кроме других данных, в этом запросе содержится имя запрашиваемого документа, а также любые параметры, передаваемые пользователем как часть запроса.

Время, прошедшее между получением Web-сервером запроса и началом возврата запрошенных данных, называется *временем обработки сценария*. В большинстве случаев обратно браузеру не передается никаких данных до тех пор, пока полностью не будет завершено выполнение всего сценария или размер сгенерированных данных не превысит значения `output_buffering`, заданного в конфигурационном файле `php.ini`. Это означает, что время обработки примерно равно промежутку времени, прошедшему между началом выполнения сценария и его завершением. Именно в течение этого промежутка времени и может возникнуть задержка, или потеря производительности. Время, прошедшее между началом передачи данных обратно Web-браузеру и завершением этой передачи, называется *временем доставки*, которое, по сути, никак не связано с модулем PHP. Время доставки во многом определяется пропускной способностью сети либо со стороны сервера (например, перегруженное соединение), либо со стороны клиента (медленный модем). Если размер Web-страницы не превышает 55К, то это, как правило, не приводит к каким-либо ощутимым временным задержкам.

Идентифицировать узкое место, которое снижает производительность, проще всего с помощью какого-нибудь простого средства, позволяющего генерировать HTTP-запрос и проанализировать полученные результаты.

Различные типы узких мест

Предположим, нужно проанализировать, где происходит снижение производительности при передаче запроса GET с параметром `foo`, который имеет значение `bar`, на получение страницы `/example.php` на сервере `www.example.com`. Конечно, это словесное описание эквивалентно строке `http://www.example.com/example.php?foo=bar`. Откройте консольное окно и установите следующее telnet-соединение.

```
ed@genesis:~$ telnet www.example.com 80
Trying 192.168.1.2...
```

```
Connected to www.example.com
Escape character is '^]'.

GET /example.php?foo=bar HTTP/1.1
Host: www.example.com
```

```
<HTML>
<BODY>
    Здравствуй, мир!
</BODY>
</HTML>
```

В приведенном примере для получения реальных выходных данных нужно подставить существующее имя узла и адрес URL. Кроме того, при вставке пустой строки нужно нажать клавишу <Enter> и ввести пробелы в точности так, как показано выше. Для простоты всю необходимую информацию можно также набрать в блокноте, а затем вставить при установке соединения telnet.

При выполнении вышеописанных действий совсем не помешает иметь под рукой секундомер. Проанализируем временные задержки, возникающие в процессе установки telnet-соединения.

- Временной интервал между нажатием клавиши <Enter> и получением строк Trying 192.168.1.2 свидетельствует о задержке, связанной с получением IP-адреса сервера на сервере имен. Такое бывает достаточно редко. Эта задержка может свидетельствовать о недостаточной скорости обработки запросов сервером имен либо вашим собственным, либо предоставляемым провайдером услуг Интернет. Рассмотрение причин существования подобных временных задержек не входит в задачи этой книги. В данном случае достаточно заметить, что средства поддержки PHP к этому не имеют никакого отношения. На практике такая временная задержка может возникнуть только один раз при первом обращении к Web-узлу, поскольку большинство Web-браузеров кешируют результаты поиска, полученные от сервера имен.
- Задержка между получением строки Trying... и строки Connected свидетельствует о том, что сам сервер потратил некоторое время для генерации ответа на запрос. Этот временной интервал может оказаться достаточно большим, поскольку для передачи страницы с изображениями может потребоваться двадцать или даже тридцать HTTP-ответов. Если с каждым запросом будет связана своя задержка, то страница будет отображаться слишком медленно даже в том случае, если сам сценарий выполнялся достаточно быстро. К сожалению, временная задержка может возникнуть в двух местах: либо в сети, используемой для передачи/получения данных с сервера, либо на самом сервере из-за его ограниченных возможностей, связанных с генерацией ответов на полученные запросы. Во втором случае причиной может оказаться чрезмерная загруженность сервера, не очень удачная настройка модуля PHP (необязательно самого сценария) или другие выполняемые на сервере процессы. Ответ на этот вопрос поможет получить анализ использования оперативной памяти и центрального процессора. Если оперативная память используется не оптимально, то решение этой проблемы следует искать в других книгах. Если же центральный процессор используется далеко не лучшим способом, то анализ нужно продолжить

и определить, какой сценарий вызывает проблему. Если это сценарий PHP, то для его исследования можно применить те же методы.

- ❑ Временная задержка между двойным нажатием клавиши <Enter> после ввода запроса HTTP и отображением кода HTML, переданного сервером, почти наверняка свидетельствует о недостаточно высокой производительности в процессе обработки. Это можно проверить с помощью добавления в программный код временного счетчика.

Если предположить, что основная задержка возникает во время обработки сценария, то сейчас самое время определить, какой фрагмент(ы) сценария к этому приводит.

Причины недостаточной производительности

Не учитывая более широкие аспекты архитектуры PHP, сейчас нужно отметить, что язык PHP предназначен для написания сценариев. Эти сценарии запускаются, выполняются и завершают свою работу, как и сценарии на любом другом языке программирования. На самом высоком упрощенном уровне выполнение сценария PHP можно описать следующим образом.

- ❑ Считывание входных параметров.
- ❑ Использование значений входных параметров для принятия решений. Как правило, в этом процессе задействуются внешние источники данных.
- ❑ Генерация выходных данных.

Первый и последний шаги приведенной последовательности не приводят к возникновению узких мест, связанных с потерей производительности, поскольку они являются неотъемлемой частью модуля PHP и Web-сервера. Другими словами, они не имеют отношения к самому программному коду. К задержке может привести лишь реальная обработка данных.

Временные задержки могут возникать по следующим причинам.

- ❑ Неоптимальные алгоритмы: неэффективный код приводит к слишком длительному времени выполнения.
- ❑ Недостаточно мощное оборудование. В данном случае код не играет особой роли, но используемое аппаратное обеспечение является не очень новым или работает в перегруженном режиме.
- ❑ Внешние узкие места, например база данных.

Следует отметить, что оборудование редко является причиной возникновения проблемы. Если не учитывать отдельные случаи, когда для получения изображения интенсивно используются средства библиотеки GD, для PHP не нужно использовать самое современное серверное оборудование.

Поиск узких мест

Существуют простые шаги, при выполнении которых можно идентифицировать каждое из узких мест. Именно этот вопрос и будет рассмотрен в оставшейся части данного приложения.

Однако первый шаг заключается в определении того, где может возникнуть узкое место и к какой из трех вышеперечисленных категорий оно относится.

Запросы к базе данных

К потере производительности зачастую приводят медленные запросы к базе данных. Практически все корпоративные приложения PHP используют какую-либо базу данных. В данной книге чаще всего упоминается СУБД PostgreSQL, однако во многих приложениях, особенно разработанных сторонними производителями, используется самая популярная СУБД MySQL.

Перед тем как приступить к оптимизации программного кода, нужно устранить узкие места, связанные с базой данных. Поскольку большая часть кода так или иначе зависит от базы данных, даже если она задействована только в процессе накопления информации, анализировать код *до* оптимизации источника данных не стоит.

Для определения узких мест в сценарии, связанных с базой данных, проще всего временно адаптировать уровень абстракции базы данных (рассматриваемый в главе 8) и добавить метод-таймер.

Рассмотрим метод генерации запроса с использованием уровня абстракции для СУБД PostgreSQL. Генерируя временные метки до и после каждого запроса, можно создать удобный для анализа журнал и оценить, как долго обрабатывался конкретный запрос. Следующий пример демонстрирует, как можно измерить промежуток времени, который потребовался для выполнения запроса уровнем абстракции базы данных.

```
$intTimeNow = microtime();
$ql_handle = pg_exec($this->link_ident, $sql);
$intTimeTaken = microtime() - $intTimeNow;
error_log("ОТЛАДКА: ЗАПРОС: $sql\n");
error_log("ОТЛАДКА: ВРЕМЯ ВЫПОЛНЕНИЯ: $intTimeTaken\n");
```

При выполнении приведенного фрагмента будет сгенерирован журнал со следующей структурой.

```
[Sun May 16 22:10:19 2004] [error] ОТЛАДКА: ЗАПРОС SELECT id, logged_in, user_id
FROM "user_sessions" WHERE session_id='98ce552be0a2ea6b6f69fbebcd14997c' AND
user_agent = 'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)' AND
ip_address='192.168.4.3'
[Sun May 16 22:10:19 2004] [error] ОТЛАДКА: ВРЕМЯ ВЫПОЛНЕНИЯ: 0.003752
```

Выполнение типичного сценария PHP с уровнем абстракции базы данных, модифицированного как показано выше, приведет к получению последовательности строк ОТЛАДКА: ЗАПРОС *хх* и ОТЛАДКА: ВРЕМЯ ВЫПОЛНЕНИЯ.

Полученную информацию будет гораздо проще анализировать, если будет запрещен вывод всех остальных предупреждений, сообщений об ошибках или какой-либо другой отладочной информации.

Путем анализа полученного журнала вы сможете быстро выявить узкие места, в которых появляется временная задержка. Как правило, все фрагменты, выполнение которых занимает больше, чем половина секунды, должны тщательно анализироваться. Следует также обратить внимание на общую длительность обработки всех запросов данной страницы. Если она занимает более трех секунд, эту ситуацию никак нельзя считать приемлемой, и вы почти наверняка услышите соответствующие комментарии от пользователей.

Алгоритмы

После устранения узких мест, связанных с обращениями к базе данных, можно перейти к внимательному просмотру самого кода. Язык PHP является чрезвычайно быст-

рым, поэтому неоптимальная реализация алгоритмов оказывает гораздо меньшее влияния на общую производительность приложения, чем взаимодействие с базой данных.

Однако встраивание в сценарий фрагментов, предназначенных для получения временных меток, как при оценке производительности взаимодействия с базой данных, позволит определить узкие места, где возможна потеря эффективности.

В то же время по очевидным причинам это нельзя осуществить за один раз. Сначала необходимо разместить операторы “запуска часов” и “останова часов” по обе стороны блока кода, который считается проблематичным. После этого можно двинуться дальше и добавить внутрь блока другие операторы, чтобы более точно идентифицировать узкое место. Не бойтесь выводить значения переменных с помощью оператора `error_log`, чтобы, например, получить данные о ходе выполнения циклов `for`.

Описанный подход до определенной степени можно автоматизировать, воспользовавшись пакетом APD (Advanced PHP Debugger, <http://pecl.php.net/package/apd>). Однако для небольших приложений функциональность этого пакета может оказаться избыточной.

Если не задействовано никакого внешнего источника данных, то выполнение фрагмента кода, как правило, происходит мгновенно. В то же время временная задержка между логическими блоками сценария более чем на доли секунды может свидетельствовать о существовании в программном коде серьезных проблем.

Повышение производительности

Теперь, когда идентифицированы узкие места и известно, где они расположены, можно сосредоточиться на модификации кода и тем самым улучшить производительность.

В то же время вам должен быть известен тот факт, что вы являетесь не единственным разработчиком PHP, которому приходится сталкиваться с узкими местами, подобными описанным выше. Поэтому существует ряд проверенных и протестированных приемов, которые можно применять для их устранения.

Устранение временных задержек при работе с базой данных

Выше вы увидели, как можно идентифицировать запросы, которые приводят к возникновению временных задержек при выполнении сценария. Причиной плохой работы запроса могут оказаться самые различные факторы. Оптимизация операторов SQL — это очень обширная тема, которую можно подробно рассмотреть лишь в отдельной книге. (Можно найти издания, посвященные оптимизации запросов для PostgreSQL, MySQL или других баз данных.) Однако следует учитывать следующие советы.

- ❑ В качестве основного индекса числовые первичные ключи всегда оказываются быстрее по сравнению с алфавитно-цифровыми первичными ключами. Даже если существуют уникальные алфавитно-цифровые значения, которые кажутся наиболее очевидной реализацией первичного ключа, лучше воспользоваться столбцом `id`.
- ❑ Убедитесь, что столбцы и комбинации столбцов (это может оказаться более важным), по которым будет выполняться фильтрация или упорядочение в операторе `SELECT`, имеют соответствующие индексы.
- ❑ Сценарии, запускаемые по графику для очистки таблиц (например, с использованием оператора `VACUUM` системы PostgreSQL), содержимое которых часто обновляется, могут существенно повысить производительность.

- ❑ Никогда не используйте оператор `SELECT *`. Если вам известны имена требуемых столбцов, укажите их в запросе. Это позволит значительно повысить производительность.
- ❑ Попробуйте минимизировать количество запросов, позволяющих достигнуть требуемого результата. Формирование строки значений `id`, соответствующих запросу, а затем применение нескольких запросов для извлечения содержимого из базы данных может оказаться очень удачным решением с точки зрения разработанной объектной модели. Однако оно будет очень неэффективным. Существует ли другой способ для решения этой задачи, предполагающий реализацию промежуточного уровня? Например, класс `GenericObjectCollection` из главы 7 именно это и позволяет сделать. На его основе можно воспользоваться объектно-ориентированным подходом и обеспечить минимизацию количества SQL-запросов, требуемых для извлечения требуемых данных.
- ❑ Несколько вложенных операторов `SELECT` использовать гораздо проще, чем выполнять соединение таблиц, однако это существенно снижает производительность. Избегайте такого подхода. (При этом в качестве побочного эффекта можно добиться и большей переносимости.) Для получения дополнительной информации по этому вопросу обращайтесь к документации по конкретной системе управления базами данных.

Многие советы, подобные приведенным выше, позволяют лишь на доли секунды повысить скорость выполнения запросов. Однако это обеспечивает получение значительного интегрального эффекта при работе со страницами, на которых используется большое количество запросов.

Не забывайте о том, что при отслеживании запросов SQL внимание нужно уделять не только самому сценарию PHP, особенно если вы руководствуетесь практическими советами, которые были приведены выше. Если вы разработали объектную модель, которая сильно зависит от базы данных, может оказаться, что потери производительности при выполнении запросов приводят к возникновению проблем и в других местах приложения. Поэтому при внесении изменений в высокоуровневые классы нужно также внимательно протестировать все компоненты, которые зависят от этих классов и убедиться, что функциональность отвечает вашим ожиданиям.

Устранение узких мест в коде

В данном случае можно применять те же самые принципы, что и при оптимизации запросов к базе данных. Однако имейте в виду, что недостаточно просто знать, в каком блоке кода имеется узкое место. Необходимо также определить, какой оператор PHP приводит к временной задержке и в каком случае.

Логические ошибки

Иногда в логике приложения содержатся простые ошибки. В таких случаях оно функционирует не так, как ожидается, и обычные приемы оценки качества не позволяют выявить ошибку. Однако некоторые “человеческие” ошибки в коде могут стать причиной потери производительности. В то же время их проще всего устраниТЬ.

Рассмотрим следующий пример.

```
switch($i) {
    case 0:
        array_pop($arMediumArray);
```

```

break;
case 1:
    array_reverse($arAnotherHugeArray);
case 2:
    $arHugeArray = array_unique($arHugeArray);
    break;
}

```

Как легко заметить, в операторе `case`, в котором переменная `$i` содержит значение 1, был пропущен оператор `break`. Это означает, что если в переменной `$i` содержится значение 1, то будет также автоматически выполнен и оператор `case` для значения 2. В данном случае все, что необходимо сделать, связано с обработкой массива `$arAnotherHugeArray`. Однако фактически будут выполнены операции и над элементами массива `$arHugeArray`. Вполне возможна ситуация, когда массив `$arHugeArray` никогда не понадобится сразу после выполнения оператора `switch` со значением 1 в переменной `$i`. В результате будет ошибочно вызван еще один метод. По существу, описанная ситуация приведет к удвоению времени выполнения блока. Добавление оператора `break` позволит предотвратить вызов лишнего метода и в результате повысить общую скорость выполнения сценария.

Методы поиска узких мест должны позволять идентифицировать и такие проблемы. Распечатав время выполнения всего блока и время выполнения тех его операторов, которые, по вашему мнению, должны быть выполнены, можно сравнить их и без проблем найти очевидные различия. Хотя они могут быть связаны и с другими проблемами, вполне возможно, что временные задержки возникли из-за логической ошибки. Следовательно, перед тем, как погрузиться в оптимизацию запросов и другие сложные приемы повышения производительности, имеет смысл проверить наличие таких ошибок, особенно при использовании таких операторов, как `switch`.

Как избежать узких мест

Внимательное использование рассмотренных выше приемов должно позволить однозначно определить узкое место или как минимум локализовать его местоположение.

Ниже перечислены встроенные средства PHP, которые по той или иной причине могут работать очень медленно.

- ❑ Любой метод, который обращается к внешнему источнику данных. Выше взаимодействие с базой данных было рассмотрено отдельно, однако к таким методам следует также отнести обращение к загруженному жесткому или сетевому диску, использование запросов HTTP и FTP, разрешение IP-адресов и взаимодействие с другими объектами через такие протоколы Web-служб, как SOAP и RPC.
- ❑ Любой процесс, который интенсивно работает с оперативной памятью, сам по себе не приводит к возникновению проблем. Однако размещение в физической памяти других экземпляров сценария иногда приводит к использованию виртуальной памяти (т.е. жесткого диска), а в результате — к потере производительности. Обычно весь программный код, в котором используются встроенная графическая библиотека GD, следует рассматривать как потенциальное узкое место, особенно при работе с большими изображениями. Стоит также проанализировать код и попробовать определить те его фрагменты, в которых используется кеширование или выполняются задачи, запускаемые по расписанию.
- ❑ Механизмы взаимодействия с сокетами сильно зависят от сетевой производительности. Гораздо лучше использовать взаимодействие сокетов в автономном

режиме, когда пользователь уведомляется о своем статусе только при поступлении запроса, а не обмениваться последовательностью запросов напрямую. Рассмотрите возможность поддержки очереди почтовых сообщений в таблице базы данных, которая обрабатывалась бы каждые пять минут. Вставляйте почтовые сообщения, которые генерируются внутри сценария, в эту очередь, а не отправляйте их из самого сценария.

- Соблюдайте осторожность при работе с совместно используемыми сетевыми дисками (в частности, доступными через файловую систему NFS) как с источниками двоичных файлов. Например, группа Web-узлов может разрешать пользователям загружать свои собственные фотографии в формате JPEG и использовать их в своих профилях. В многосерверном окружении эти JPEG-файлы должны быть доступны всем Web-серверам. Для реализации такого хранилища имеет смысл использовать диск, а не базу данных (которая при хранении двоичных данных оказывается малоэффективной), однако совместно используемый сетевой диск является чрезвычайно медленным. Вместо этого двоичные данные лучше реплицировать между всеми серверами с помощью специальных сценариев или других средств. Такой сценарий каждые пять минут может выполняться в фоновом режиме и обеспечивать передачу данных между серверами. Так можно добиться того, чтобы локальная копия всех данных была доступна уже через пять минут после ее начальной загрузки.

Можно привести и много других примеров, когда могут возникнуть узкие места и потеря производительности. В разделах интерактивной документации по языку PHP, посвященных определенным методам и классам, зачастую содержатся замечания о потенциальных проблемах потери производительности.

Если в вашем приложении используются классы модуля PEAR, убедитесь, что на сервере установлена его самая последняя поддерживаемая версия. Практически во всех программных пакетах обнаружаются ошибки (которые в дальнейшем исправляются). Поэтому при использовании последних версий можно исключить любые проблемы потери производительности, которые могут из-за этих ошибок возникнуть.

Тестирование

Любые изменения, внесенные в программный код или SQL-запросы по результатам проведенного анализа, должны быть тщательно протестированы. Убедитесь, что в процессе тестирования как отдельных сценариев, так и всего приложения в целом, были применены базовые принципы оценки качества, которые обсуждались в главе 23 “Обеспечение качества”

Это предполагает тестирование не только входных параметров и сценариев использования, но и всех возможных параметров и сценариев, на которые могут повлиять сделанные изменения. Лучше провести избыточное тестирование, чем оставить в приложении неисследованные фрагменты.

Если вы внесли изменения в среде разработки, а не в реальном окружении, попробуйте скопировать используемую базу данных или другое хранилище данных и протестировать их локально. Если это невозможно, попробуйте установить виртуальный сервер Web-сервера Apache (или другого используемого Web-сервера) и воспользоваться им как временным окружением, в рамках которого будет использоваться новый программный код и реальная база данных.

В любом случае убедитесь, что при переходе к реальному использованию приложения сделанные изменения не только обеспечили улучшение производительности, но и не повлияли на его прежнюю функциональность.

Предупреждение неприятностей

Все, что обсуждалось до настоящего момента, касалось исправления ошибок в уже существующем приложении. Это очень хорошо и правильно. Однако, к сожалению, подобная деятельность инициируется по просьбе конечных пользователей, которые столкнулись с низкой производительностью используемого приложения. А это уже может привести к серьезным коммерческим проблемам.

Более эффективный и профессиональный подход заключается в проектировании приложения с учетом требований к его производительности. Конечно, этот вопрос заслуживает изложения в отдельной книге. Вообще говоря, принципы, положенные в основу данной книги, всесторонне учитывают производительность, поэтому ни один из них не может неблагоприятно повлиять на ваше конкретное приложение, если оно используется соответствующим образом и вместе с корректными наборами данных. Проблема потери производительности может появиться в вашей собственной архитектуре, так что при ее проектировании необходимо все тщательно продумать.

Рекомендации по реализации высокопроизводительной архитектуры

Ниже приведено несколько полезных рекомендаций, касающихся проектирования архитектуры приложения. При их использовании можно обеспечить быструю разработку эффективной программной архитектуры с самого начала выполнения проекта.

- ❑ **Правильно используйте оборудование.** На неудачно подобранным оборудовании даже самые быстрые сценарии PHP будут выполняться не слишком быстро. Обработка Web-страниц является относительно простой операцией, так что слабое аппаратное обеспечение лучше использовать для сервера Apache, а более мощное оборудование — оставить для развертывания баз данных. Кроме того, позаботьтесь о комплектации серверов высокопроизводительными SCSI-дисками и, если возможно, объедините их в RAID-массив.
- ❑ **Используйте кеширование на самом низком из возможных уровней.** Если некоторые из сценариев многократно генерируют одни и те же выходные данные, нужно ли при этом каждый раз обращаться к базе данных? Да, на некотором уровне базу данных можно использовать для кеширования ответов на запросы, однако если каждый раз генерируется один и тот же HTML-код, почему бы не кешировать именно его? В настоящее время можно найти много пакетов от сторонних производителей, специально предназначенных для поддержки кеширования. В то же время можно реализовать и свой собственный механизм кеширования, воспользовавшись сериализацией параметров GET, POST и COOKIE, передаваемых в сценарий. Сравнив данные, соответствующие текущему и предыдущему запросу, можно оценить уникальность текущего запроса и по возможности воспользоваться кешированными данными, а не обращаться к базе данных.
- ❑ **Выполняйте случайные процессы в автономном режиме.** Если в одном из ваших сценариев нельзя точно определить условия запуска некоторого процесса, всерьез подумайте о его автономном выполнении. Наиболее очевидным

примером такого процесса является обработка кредитной карточки с использованием системы обработки платежей. Если в приложении нужно обеспечить авторизацию в реальном времени (например, если покупатели имеют возможность приобретать товары в интерактивном режиме), используйте простую автоматически обновляемую страницу и выполните переход на завершающую страницу после обновления базы данных. Запускайте внешний сценарий по заданному расписанию, например, каждые 60 секунд. В этом сценарии должна выполняться авторизация кредитной карточки и обновление базы данных в случае успешной авторизации.

- **Аккуратно используйте базы данных.** Не всю информацию нужно хранить в базе данных. Например, в системе управления содержимым данные лучше хранить в файле XML, поскольку базы данных — не лучшее место для хранения больших фрагментов текста.
- **Оптимизируйте запросы к базе данных.** Исследуйте слабые места сценариев, связанных с обращением к базе данных, и постараитесь ускорить выполнение запросов. Например, СУБД PostgreSQL относительно медленно обрабатывает вложенные запросы. Поэтому их лучше заменить на внутреннее соединение. Кроме того, проверьте необходимость используемых индексов и удалите лишние, поскольку они существенно замедляют взаимодействие с базой данных.
- **Тестируйте нагрузку.** Тестируйте нагрузку на отдельные компоненты с учетом реального трафика и объемов данных, моделируя производительность реального приложения на этапе разработки. Этот вопрос рассматривается ниже.

Тестирование нагрузки

Функциональное тестирование — это важная часть оценки качества, однако не меньшую роль в обеспечении производительности завершенного приложения играет тестирование нагрузки.

По существу, тестирование нагрузки состоит в имитационном моделировании большого количества одновременных соединений с Web-сервером и выполнении типичных сценариев пользователя. В процессе тестирования нагрузки необходимо оценивать производительность сценариев, а также общую нагрузку на сервер.

Результаты тестирования необходимо записывать в соответствующие таблицы, анализ которых позволит определить среднее время отклика при заданном числе одновременно подключенных пользователей. Пусть это время отклика составляет N . Очевидно, что с ростом количества пользователей N стремится к бесконечности. Однако открытым остается вопрос: когда N достигает неприемлемо большого значения? Или: каково максимальное количество одновременно подключенных пользователей к системе?

Некоторые пакеты, такие как ApacheBench (<http://codeflux.com/ab/>), позволяют эмулировать очень простые сценарии. Но существуют и коммерческие программы, в частности LoadRunner, реализующие более реалистичные сценарии за счет введения случайных задержек между запросами каждого пользователя и случайного выбора последовательности страниц. Такое поведение более характерно для реальных пользователей.

При тестировании нагрузки целесообразно использовать отдельный сервер (в идеале серверы), который будет эмулировать клиентов и подключаться к тестируемому Web-серверу. При этом тестовое программное обеспечение не будет находиться на

самом проверяемом Web-сервере. Кроме того, для обеспечения презентабельности результатов необходимо воспроизводить максимально реалистичную среду тестирования и конфигурацию.

Результаты тестирования (конечно же, положительные) полезно ненавязчиво продемонстрировать заказчику. Это позволит ему спланировать процесс будущего расширения системы.

Резюме

В этом приложении были рассмотрены различные типы узких мест в программном коде и способы их быстрого устранения. Вы также узнали, как избежать проблем с потерей производительности за счет устранения некоторых распространенных недостатков программного обеспечения и системной архитектуры.

И наконец, вы познакомились со способами тестирования нагрузки и некоторыми факторами, которые необходимо принимать во внимание при разработке проекта.



Практические советы по установке PHP

В настоящее время практически любой сможет выполнить простую установку программного обеспечения. Даже ваши родители в состоянии установить последнюю версию операционной системы Windows на персональном компьютере.

Исключением из этого правила являются серверные приложения, такие как модуль PHP. Хотя на соответствующем Web-узле и в документации вопросы установки и запуска кратко рассматриваются, по существу, там можно найти лишь перечень действий, которые нужно для этого выполнить. Обеспечение работоспособности серверного приложения требует определенных знаний.

В данном коротком приложении будет подробно рассмотрен вопрос корректной установки модуля PHP в конфигурации, которая лучше всего подходит для разработки и развертывания корпоративных приложений. Кроме того, вы узнаете о различиях между версиями PHP для системы Windows и системы UNIX, а также о том, какую из них предпочтительнее применять для разработки корпоративных приложений.

Если до сих пор на вашем компьютере еще не был установлен модуль PHP, займитесь этим прямо сейчас и лишь после этого переходите к изучению оставшейся части книги.

Введение в установку PHP

PHP — это язык для разработки приложений. Он принципиально отличается от таких языков, как Pascal или C++, для которых разработаны специальные компиляторы. Соответствующий модуль, который позволяет интерпретировать сценарии, написанные на языке PHP, обязательно нужно установить поверх какого-либо Web-сервера, например Apache.

Таким образом, фраза “установка PHP” является недостаточно точной. А фраза “установка PHP и Apache” звучит гораздо лучше, однако в ней никак не отражен тот факт, что развитая функциональность PHP во многом определяется используемыми

сторонними библиотеками и приложениями, например библиотекой `libxml`, для анализа кода XML.

Что еще хуже, большинство приложений, рассматриваемых в данной книге, а также те приложения, которые вы разработаете самостоятельно, требует применения какой-либо системы управления базами данных. Здесь в основном рассматривается СУБД PostgreSQL, а не MySQL, однако на практике они предоставляют одни и те же функциональные возможности.

Поэтому последовательность шагов, которые потребуется выполнить для установки PHP, может выглядеть следующим образом.

- Установка и настройка различных внешних библиотек.
- Установка и настройка системы управления базами данных.
- Установка и настройка Web-сервера.
- Установка и настройка модуля PHP.

Как можно увидеть, приведенный перечень не очень-то похож на безобидное развлечение. Следует также заметить, что точный порядок выполнения вышеперечисленных шагов может изменяться в зависимости от выбранной операционной системы (Windows или UNIX), выбранного Web-сервера (Apache, IIS, Zeus и т.д.) и от платформы базы данных (MySQL, PostgreSQL, MS SQL Server, Oracle, IBM DB2, Informix).

Попытка описания в данном приложении всех возможных вариантов конфигурации не просто привела бы к увеличению объема этой книги, но и стала бы настоящим кошмаром, поскольку в пользу одной определенной комбинации программных средств всегда можно найти веский аргумент.

Выбор платформы

Сейчас сразу следует заметить, что в этой книге не рассматривается “спор” между операционными системами. Те из читателей, которых интересуют обоснованные и аргументированные мнения специалистов по этому вопросу, могут познакомиться с материалами Web-узла *Slashdot* (slashdot.org).

Тем не менее можно сформулировать следующее важное утверждение: “Система Windows является неудачной платформой для развертывания сервера приложения, а система UNIX — для использования в качестве операционной системы рабочей станции”.

Это несколько грубое обобщение и для него можно найти ряд исключений. Однако десятки тысяч провайдеров услуг Интернет и больших корпораций США не могут ошибаться. Хотя Windows в корпоративной среде зачастую используется в качестве сервера обмена или контроллера домена, для реализации службы DNS, маршрутизации почты на уровне протокола SMTP или хранения Web-узлов обычно применяют систему UNIX. Конечно же, существуют любители операционной системы Linux с оконными надстройками типа Gnome, KDE и т.д., но таких меньшинство. Большинство пользователей считают Windows операционной системой для рабочих станций, а UNIX — серверной платформой. Причины такого разделения предельно просты. Интерфейс Windows оптимизирован для рабочих станций в течение многих лет разработки. Основная цель создателей Windows сделать ее интерфейс интуитивным и эффективным. Интерфейс Windows нельзя назвать совершенным, но он существенно превосходит своих оппонентов в UNIX-подобных системах.

Система Windows не очень хороша в качестве сервера услуг Интернет. На момент написания этой книги самым распространенным сервером для хранения Web-узлов на платформе Windows являлся IIS (Internet Information Services). Его последние версии существенно усовершенствованы, но он по-прежнему страдает от множества изъянов в области безопасности, а системные администраторы считают его ненадежным и неустойчивым. Кроме того, Windows – это слишком ресурсоемкая среда для обслуживания Интернет. Из 2 Мбайт памяти 512 Мбайт может отводиться для работы операционной системы с ее различными службами и лишь 64 Мбайта могут реально использоваться для работы Web-сервера. Операционная система UNIX является более экономичной и надежной. Ее не так просто настроить, но она работает несравненно быстрее и устойчивее, требуя при этом гораздо меньше ресурсов.

Сервер IIS на платформе Windows лучше использовать для Web-узлов на базе технологии ASP. Однако данная книга посвящена языку PHP, а запускать PHP на платформе Windows это все равно, что использовать ASP на платформе UNIX.

Если же вы являетесь “убежденным сторонником” сервера на базе Windows, то вам придется вступить в борьбу с ветряными мельницами. Провайдеры услуг Интернет наверняка не захотят устанавливать приложения PHP на платформе Windows по перечисленным выше причинам. Код PHP должен работать на платформе UNIX – и точка. Если же вы хотите разрабатывать PHP-приложения для Windows, то вам придется иметь в виду, что в конечном итоге оно все равно будет работать в среде UNIX.

Возможно, вам придется разрабатывать корпоративные приложения для клиента, сеть которого работает на платформе Windows. Для такой ситуации в конце этой главы приводится ряд отличительных особенностей различных платформ. Однако подобная ситуация является исключением, а не правилом, поскольку чаще всего Web-приложения устанавливаются на серверах провайдера.

Поэтому авторы данной книги исходят из предположения, что приложения PHP устанавливаются на платформе UNIX, а не Windows. На этом предположении базируются все примеры этой книги. PHP на платформе UNIX – это выбор профессионалов, а данная книга рассчитана именно на них.

Не расстраивайтесь. Это вовсе не означает, что вам придется установить систему UNIX на своей машине. В предыдущих приложениях рассматривались различные среды разработки, в том числе Zend Studio, и отмечалось, что Windows – лучшая платформа для рабочих станций. И это действительно так. На сервере разработки, который может быть установлен рядом с вашим персональным компьютером, вам действительно придется установить систему UNIX. Но на своей рабочей станции вы можете использовать привычную операционную систему.

Лучший Web-сервер

Выбрав операционную систему, вы окажетесь перед не столь широким ассортиментом Web-серверов. Например, вы не сможете установить сервер IIS на платформу UNIX. Для UNIX остается несколько возможных вариантов: Apache, Zeus, AOL Server и Pi3Web. PHP можно скомпилировать и установить двумя способами: как выполняемый файл CGI или как интегральную часть Web-сервера, известную под названием модуля SAPI. Последний вариант гораздо быстрее и может обрабатывать множество параллельных соединений. При использовании двоичного файла CGI для обработки каждого отдельного HTTP-запроса требуется загрузить свой экземпляр интерпрета-

тора PHP, что очень неэффективно. В данной книге по умолчанию считалось, что PHP установлен в качестве модуля SAPI.

В таких условиях практически безальтернативным выбором является сервер Apache. Он используется на 70% Web-серверов во всем мире. Вероятнее всего, именно этот Web-сервер установлен у провайдера, у которого вы планируете разместить свое приложение. Поэтому, ориентируясь на Apache, вы существенно упростите себе жизнь.

Это хороший выбор. Apache показал себя устойчивым, безопасным, легко управляемым. Он постоянно развивается. Интересно отметить, что на момент написания этой книги Web-узел сервера AOL Server (www.aolserver.com) работает не на AOL Server, а на Apache.

Таким образом, предполагается, что приложение PHP будет работать на основе сервера Apache на платформе UNIX. К счастью, в отличие от операционной системы, тип используемого Web-сервера никак не влияет на синтаксис PHP. Поэтому все примеры кода данной книги рассчитаны на операционную систему UNIX, но никак не связаны с конкретным Web-сервером.

Лучшая база данных

Сервер базы данных может быть физически установлен на другой машине, а приложения с интенсивным трафиком могут потребовать двух или даже трех отдельных серверов баз данных.

Выбор таких серверов достаточно широк. Среди возможных вариантов: PostgreSQL, MySQL, Oracle, Informix, DB2, Microsoft SQL Server, SUP DB и многие другие. Все эти варианты достаточно хороши. Поскольку серверы баз данных не имеют выхода в Интернет, вопросы безопасности при их выборе не играют ключевой роли. Если брандмауэр настроен правильно, то все серверы баз данных являются одинаково безопасными. Все эти серверы являются достаточно надежными и быстрыми. Реальные различия в производительности зачастую определяются структурой базы данных, а не особенностями самого сервера. При таких характеристиках сложно сделать правильный выбор. При написании этой книги мы столкнулись с такими же трудностями, но остановились на PostgreSQL. Этот выбор объясняется множеством причин.

Во-первых, мы хотели максимально придерживаться свободно распространяемого программного обеспечения, поскольку Apache и PHP являются бесплатными. Строго говоря, база данных MySQL не является бесплатной. PostgreSQL – бесплатна, поэтому удовлетворяет первому критерию.

Во-вторых, PostgreSQL столь же близка к промышленным стандартам, как и Oracle. Ее синтаксис ANSI-совместим, поэтому приложения легко переносить с базы данных PostgreSQL на любую другую базу данных.

И наконец, эта СУБД хороша во всех отношениях. Она является достаточно быстрой, очень устойчивой, чрезвычайно простой в использовании, а также хорошо поддерживается средствами PHP, что чрезвычайно важно в контексте этой книги.

Инсталляция

Теперь, когда мы разобрались с основами, можно перейти к самому процессу установки. Ниже предполагается, что в вашем распоряжении имеется рабочая машина под управлением операционной системы UNIX, которая полностью готова для установки программного обеспечения.

Если у вас нет машины с системой UNIX, но вы хотите работать именно на этой платформе, рассмотрите вопрос использования системы Linux, прекрасной UNIX-подобной операционной системы, которая идеально подходит для аппаратной архитектуры Intel. При ее установке позаботьтесь об инсталляции и всех служебных утилит, поскольку они понадобятся при развертывании и запуске модуля PHP.

Кроме того, предполагается, что машина с системой UNIX работает в сети, имеет собственный IP-адрес и доступна с вашей рабочей станции и, наоборот, ваша рабочая станция “видна” с UNIX-машины (это можно проверить с помощью утилиты ping). Для практического усвоения приведенного ниже материала необходимо обладать также знанием основных команд и утилит командной строки системы UNIX.

Если все вышеперечисленное (знания и оборудование) имеется в вашем распоряжении, можно приступать к инсталляции. Зарегистрируйтесь на UNIX-машине с правами root (или получите эти права с помощью команды su) и двигайтесь дальше.

Загрузка и установка СУБД PostgreSQL

Поскольку используемая среда разработки предназначена исключительно для ваших нужд, СУБД PostgreSQL нет необходимости развертывать на отдельном узле. Для этого вполне подойдет компьютер с Web-сервером Apache и интерпретатором PHP.

Если это необходимо, загрузите полный архив PostgreSQL (www.postgresql.org), в котором содержится исходный код, а не двоичные файлы. Этот код вы скомпилируете самостоятельно, чтобы добиться от этой СУБД максимальной скорости и гибкости. Загружаемый файл должен иметь примерно следующее имя: `postgresql-7.4.3.tar.gz`.

Файл можно загрузить с помощью браузера lynx или команды `ftp`, однако авторы предпочитают использовать команду `wget`.

```
# wget ftp://ftp.postgresql.org/pub/latest/postgresql-7.4.3.tar.gz
```

Затем распакуйте загруженный файл.

```
# tar -xzvf postgresql-7.4.3.tar.gz
```

При этом будет создан каталог `postgresql-7.4.3`, в котором будут содержаться файлы с исходным кодом. Перейдите в этот каталог с помощью следующей команды.

```
# cd postgresql-7.4.3
```

Теперь нужно запустить сценарий, создающий `make`-файл, соответствующий используемой системе UNIX.

```
# ./configure
```

Если все пройдет успешно, на экране не появятся какие-либо сообщения об ошибках, а отобразится служебная информация примерно со следующей последней строкой.

```
config.status: linking ./src/makefiles/Makefile.linux to src/Makefile.port
```

Если вся информация, которая была выведена на экран, не вызывает у вас подозрений, можно переходить к процессу компиляции. Для этого достаточно ввести следующую команду.

```
# make
```

Компиляция может занять некоторое время, так что сейчас можно не опасаться и пойти выпить чашку кофе. Когда вы вернетесь, процесс компиляции успешно завершится. После этого можно приступить к установке.

```
# make install
```

Вообще говоря, в данный момент можно сказать, что система PostgreSQL уже установлена. Однако осталось выполнить ее дополнительную настройку. В следующем фрагменте задается рабочий каталог базы данных (в котором будут храниться данные).

```
# adduser postgres
# mkdir /usr/local/pgsql/data
# chown postgres /usr/local/pgsql/data
# su - postgres
# /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

Теперь почти все готово для запуска демона (сервера) PostgreSQL. Осталось выполнить один завершающий шаг: необходимо сообщить системе, какие соединения являются разрешенными. Как правило, входящие соединения могут устанавливаться с того же самого сервера или в крайнем случае с других компьютеров сети.

Допустим, IP-адресом сервера является 192.168.1.1 и вы хотите разрешить доступ только с этого IP-адреса. Откройте файл /usr/local/pgsql/pg_hba.conf с помощью любого редактора и добавьте в его конец следующую строку.

```
Host      all      all      192.168.1.1    255.255.255.255  trust
```

Сохраните этот файл и закройте редактор. Теперь можно запустить систему PostgreSQL следующим образом.

```
# su - postgres
# /usr/local/pgsql/bin/postmaster -i -D /usr/local/pgsql/data &
```

Первая команда является очень важной. Процесс postmaster должен запускаться с правами нового пользователя postgres, а не с правами суперпользователя root или любого другого пользователя с эквивалентными правами. Параметр -i также очень важен, поскольку он разрешает использовать соединения TCP/IP, которые по умолчанию являются запрещенными.

Вот и все. Теперь можно создать базу данных, снова воспользовавшись правами пользователя postgres.

```
# /usr/local/pgsql/bin/createdb имя_базы_данных
```

После этого с помощью консоли PostgreSQL этой базой данных можно манипулировать.

```
# /usr/local/pgsql/bin/pgsql имя_базы_данных
```

И наконец, можно создать нового пользователя.

```
# /usr/local/pgsql/bin/createuser имя_пользователя
```

На этом интенсивный курс по установке СУБД PostgreSQL завершается. Сейчас лишь следует заметить, что дополнительную информацию об этой системе можно найти в соответствующей документации либо при чтении данной книги.

Установка дополнительных библиотек

В идеальном мире достаточно было бы нажать одну клавишу и сразу установить интерпретатор PHP и Web-сервер Apache. К сожалению, реальная жизнь является совсем не такой совершенной. В результате почти наверняка вы столкнетесь не с самой приятной задачей, связанной с необходимостью установки дополнительных библиотек для PHP.

Перечень таких библиотек зависит от того, какая функциональность вам требуется. Например, если нужно обеспечить синтаксический анализ кода XML и XSL, придется установить соответствующую библиотеку поддержки.

Для рассмотрения примеров этой книги необходимо иметь в своем распоряжении следующие библиотеки.

- ZLib версии 1.1.3 или выше, <http://www.qzip.org/zlib/>.
- libJPEG версии 6b или выше, <http://www.ijg.org/>.
- libPNG версии 1.0.8 или выше, <http://libpng.org/pub/png/libpng.html>.
- Expat XML Parser 1.95.x, <http://sourceforge.net/projects/expat>.
- Freetype 2.1.4, <http://www.freetype.org>.
- LibXML версии 2.6.4 или выше, <http://www.xmlsoft.org>.
- LibXSLT версии 1.1.4 или выше, <http://www.xmlsoft.org>.

К счастью, все вышеперечисленные библиотеки являются бесплатными.

Каждый из перечисленных пакетов загружается и устанавливается одинаково. Для этого достаточно просто выполнить следующую последовательность действий.

1. Посетите соответствующий Web-узел и найдите соответствующую ссылку, с помощью которой можно загрузить требуемый пакет.
2. С помощью команды `wget` загрузите требуемый архивный файл с исходным кодом.
3. Распакуйте архив с помощью команды `tar -xzvf имя_файла.tar.gz`.
4. Перейдите в созданный каталог, который обычно имеет имя `имя_файла` (имя загруженного файла, но без расширения `.tar.gz`).
5. Введите команду `./configure --enable-shared`.
6. Введите команду `make`.
7. Введите команду `make install`.
8. Перейдите обратно в исходный каталог (`cd ..`) и приступите к установке следующего пакета.

Вместе с тем имеется и ряд исключений.

- Перед настройкой библиотеки Freetype с помощью любого текстового редактора нужно удалить символы комментария в строке `#define TT_CONFIG_OPTION_BYTECODE_INTERPRETER` файла `include/freetype/config/ftoption.h`. Это является очень важным, если в сценариях PHP нужно обеспечить корректную генерацию шрифтов TrueType. После этого можно продолжить выполнение обычных действий по установке пакета.
- В комплект библиотеки libPNG не входит сценарий `configure`. Поэтому сначала нужно скопировать корректный `make`-файл из каталога со сценариями,

например, следующим образом: `cp scripts/makefile.lnx ./Makefile.` После этого продолжайте установку.

После установки всех вышеперечисленных пакетов можно приступать к установке модуля PHP и Web-сервера Apache.

Установка PHP и Apache

PHP и Apache должны быть установлены одновременно, именно поэтому их инсталляция и рассматривается в одном разделе.

Во-первых, загрузите самые последние версии PHP и Apache с соответствующих Web-узлов: www.apache.org и www.php.net. Следует заметить, что сервер Apache версии 2 еще далек от совершенства, так что лучше воспользоваться последней версией 1.3.x (на момент написания книги 1.3.31).

Два загруженных файла должны иметь примерно следующие имена: `apache-1.3.31.tar.gz` и `php-5.0.0.tar.gz`. Распакуйте их обычным образом.

```
# tar -xzvf apache-1.3.31.tar.gz
# tar -xzvf php-5.0.0.tar.gz
```

Далее нужно выполнить начальную настройку сервера Apache. Перейдите в соответствующий каталог (`apache-1.3.31`) и запустите сценарий `configure` без параметров. На экране появится предупреждение, однако на него внимания обращать не нужно.

```
# ./configure
Configuring for Apache, Version 1.3.31
+ Warning: Configuring Apache with default settings.
+ This is probably not what you really want.
```

После завершения работы сценария нужно выполнить настройку модуля PHP. Для этого выйдите из каталога Apache и перейдите в каталог PHP следующим образом.

```
# cd ..
# cd php-5.0.0
```

Теперь модуль PHP нужно настроить с помощью очень длинной команды, которая должна быть введена как одна строка.

```
./configure --with-apache=../apache_1.3.31 --with-libxml-dir=/usr/local/lib --
with-gd --with-gettext --without-mysql --with-pgsql --enable-sockets --with-
jpeg-dir=/usr/local/lib --with-png-dir=/usr/local/lib --with-zlib-
dir=/usr/local/lib --enable-gd-native-ttf --with-freetype-dir=/usr/local/lib -
-with-xmlrpc --with-dom -enable-xslt --with-expat-dir=/usr/local/lib --with-xsl
```

Не забудьте, что при использовании версии сервера Apache, более новой чем 1.3.31, нужно соответствующим образом модифицировать и директиву `--with-apache`.

Хотя процесс ввода такой команды и потребует немного времени, однако если вы внимательно проделали все действия, перечисленные до сих пор в этом приложении, то не должны получить сообщений об ошибках. Весь процесс должен завершиться успешно.

Теперь можно перейти к развертыванию модуля PHP.

```
# make
# make install
```

Выполнение команды `make` может занять несколько минут, в зависимости от быстродействия сервера. В то же время команда `make install` должна завершиться го-

раздо быстрее. При возникновении проблем проанализируйте действия, которые были выполнены ранее. В частности, удостоверьтесь в том, что были корректно сконфигурированы все служебные пакеты, которые упоминались выше.

В результате выполнения команды `make` в каталоге Web-сервера будет создан модуль PHP. Теперь можно настроить и установить Apache таким образом, чтобы он использовал этот модуль.

```
# cd ..
# cd apache-1.3.31
@ ./configure --prefix=/usr/local/apache --activate-
module=src/modules/php5/libphp5.a
```

Не забывайте о том, что команда `configure` должна быть введена как одна строка. После выполнения этой команды на экране должна появиться информация об успешной настройке сервера Apache и активизации модуля PHP. Это будет свидетельствовать о том, что на данный момент все действия выполнены успешно.

Теперь можно двигаться дальше и скомпилировать Apache.

```
# make
# make install
```

Остался один завершающий шаг. Серверу Apache нужно сообщить о том, как требуется обрабатывать расширение `.php`. Для этого откройте файл `/usr/local/apache/conf/httpd.conf` и добавьте в него следующие строки.

```
AddType application/x-httdp-php .php .php4 .php3 .php5
AddType application/x-httdp-php-source .phps
```

Теоретически эти строки можно добавить в любое место файла, однако лучше их разместить рядом с другими директивами `AddType` и `AddHandler`.

Тестирование рабочего окружения

На данный момент осталось выполнить общее тестирование установленного программного обеспечения. Запустите сервер Apache с помощью следующей команды.

```
# /usr/local/apache/bin/apachectl start
```

В каталоге `/usr/local/apache/htdocs` создайте и сохраните файл `test.php` со следующим кодом.

```
<?php
    phpinfo();
?>
```

В строке адреса Web-браузера введите IP-адрес UNIX-машины и строку `/test.php`. Например, если IP-адресом сервера является 192.168.1.1, то в браузере должен быть введен следующий адрес: `http://192.168.1.1/test.php`. В результате в окне браузера вы должны увидеть следующую информацию (рис. Г.1).

Если эта информация появилась на экране, вас можно поздравить: установка модуля PHP удачно завершилась! Вполне возможно, что для каждого рабочего проекта в дальнейшем вам понадобится создать свой собственный виртуальный сервер. Для получения более подробной информации по этому вопросу читайте главу 24 “Развертывание”.

Когда использовать систему Windows

Может возникнуть ситуация, когда вам придется использовать именно систему Windows. Классический пример — необходимость установки закрытой системы в информационной Windows-инфраструктуре большой компании, которая не должна быть связана с внешним миром. При этом установка сервера UNIX может вызывать резкое неприятие.

The screenshot shows a Microsoft Internet Explorer window displaying the results of a `phpinfo()` call. The title bar reads "phpinfo() - Microsoft Internet Explorer". The main content area displays the PHP configuration information in a table format. A large "php" logo is visible on the right side of the page. At the bottom, there is a note about Zend Engine usage and a "Powered By Zend Engine" logo.

System	Linux www 2.4.26 #1 Wed Jun 16 16:01:25 BST 2004 i686
Build Date	Jun 18 2004 16:49:02
Configure Command	<code>'./configure' --with-apache=../apache_1.3.29' --libdir=/usr/local/lib-php5' --with-libxml-dir=/usr/local/lib' --with-gd' --with-gettext' --without-mysql' --without-pgsql' --enable-sockets' --with-jpeg-dir=/usr/local/lib' --with-png-dir=/usr/local/lib' --with-zlib-dir=/usr/local/lib' --enable-gd-native-ttf' --with-freetype-dir=/usr/local/lib' --with-xmlrpc' --with-dom' --enable-xslt' --with-expat-dir=/usr/local/lib' --with-xsl' --with-imap=/usr/local/src/ashridge-changes/imap-2002e'</code>
Server API	Apache
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/lib-php5/php.ini
PHP API	20031224
PHP Extension	20040412
Zend Extension	220040412
Debug Build	no
Thread Safety	disabled
IPv6 Support	enabled
Registered PHP Streams	php, file, http, ftp, compress.zlib
Registered Stream Socket Transports	tcp, udp, unix, udg

This program makes use of the Zend Scripting Language Engine:
Zend Engine v2.0.0RC2, Copyright (c) 1998-2004 Zend Technologies

Powered By

Рис. Г.1.

С помощью небольших усилий можно обеспечить работоспособность всех примеров из данной книги и на платформе Windows. Следующие советы помогут вам упростить решение этой задачи.

Модификация путей

Не забывайте о том, что пути в системе Windows выглядят иначе, чем в системе UNIX. Поэтому пути к каталогам (/data/res/1230.jpg) должны быть заменены на их эквиваленты (C:\data\res\1230.jpg).

Язык PHP является очень гибким и позволяет использовать косую черту (/) и в системе Windows. Если вам удастся использовать только относительные пути (без символов логических дисков), то переносимость Web-приложения существенно повысится.

Странные различия

Некоторые различия языка PHP в системах UNIX и Windows практически невозможны объяснить. Например, в Windows нельзя использовать класс Variant, тогда как в UNIX он прекрасно работает. Наверняка существуют и другие подобные различия. Если вы столкнулись с такой проблемой, сделать можно не так-то и много, особенно с учетом отсутствия какой-либо информации в официальной документации. Лучше всего воспользоваться системой поиска Google и посмотреть, сталкивались ли другие разработчики с аналогичной проблемой, и если да, то как она была ими разрешена.

Внешние библиотеки

Поскольку в состав системы Windows не входит встроенный компилятор C/C++, как в системе UNIX, то для работы модулю PHP не требуется множество внешних библиотек. Вместо этого можно пользоваться набором уже скомпилированных динамически подключаемых библиотек, которые и обеспечивают возможность использования расширений, упоминавшихся в этом приложении (LibXML, LibXSLT и т.д.).

Для того чтобы воспользоваться таким расширением, скопируйте соответствующую библиотеку DLL в каталог PHP (C:\PHP) и каталог windows\system32, а затем перезапустите Web-сервер. После этого расширение будет полностью готово к использованию.

Использование пакета PEAR

Установка библиотек из пакета PEAR в системе Windows несколько отличается от аналогичного процесса в системе UNIX.

Перед тем как вы сможете использовать это пакет, придется его установить с помощью следующего сценария, который входит в комплект поставки.

```
C:\php\PEAR>..\php go-pear.php
Welcome to go-pear!
Go-pear will install the 'pear' command and all the files needed by
it. This command is your tool for PEAR installation and maintenance.
```

Этот сценарий будет запрашивать различные параметры, однако после завершения его работы в вашем распоряжении появится команда pear, которую можно использовать точно так же, как и в системе UNIX.

Резюме

В этом коротком приложении были кратко сформулированы причины, по которым в примерах данной книги использовалась операционная система UNIX, Web-сервер Apache и система управления базами данных PostgreSQL. Вы также узнали, почему это же программное обеспечение стоит установить и в вашем рабочем окружении.

Кроме того, вы изучили все основные шаги, которые нужно выполнить на практике для установки всего необходимого программного обеспечения в системе UNIX.

И наконец, в конце этого приложения были кратко рассмотрены основные различия PHP в системах Windows и UNIX, а также способы эффективного решения связанных с ними проблем.

Предметный указатель

A

Accessor method, 32; 50

Activation, 67

Activity, 64
diagram, 63

Actor, 54

Advanced PHP Debugger (APD), 576

Aggregate, 60

AOL Server, 586

Apache, 585
FOP, 534

ApacheBench, 581

Association, 58

Attribute, 57

B

Bi-directional navigability, 59

C

Callback, 127

CGI (Common Gateway Interface), 301

Chat-bot, 350

Class, 27

Committed Information Rate (CIR), 401

Component diagram, 68

Composite, 75

Composition, 94

Concurrent Versioning System (CVS), 550

Constructor, 33

Cookie, 305

D

Decision point, 64

Dependency, 68

Deployment diagram, 68

Development Environment (DE), 554

Document Object Model (DOM), 533

Domain, 53
modeling, 56

DSN (Data Source Name), 179

E

EditPlus, 569
eFax, 300
Encapsulation, 27; 49
Exclusive versioning, 541
Expat XML Parser, 589
Expert, 53
Extract method, 442
eXtreme Programming, 361; 407
EzSOAP, 232

F

Finite state machine, 343

Fork, 64

Freetype, 589

G

Generalization, 60; 75

H

HTTP, 301

I

Image_Graph, 535
Infrastructure, 250
Inheritance, 27; 38
Instantiation, 28
Integrated Development Environment (IDE), 554
Interface, 27; 47
Internal member variable, 32
Iteration, 417

J

Join, 64

K

Komodo, 565

L

Lazy instantiation, 121

LibJPEG, 589

License Manager, 565

Lifeline, 66

LoadRunner, 392; 581

M

Mail Transfer Agent (MTA), 415

Mantis, 392; 501

Microsoft Visual SourceSafe, 548

MrProject, 396

Multiplicity, 75

MVC (Model–View–Controller), 248

N

NAT (Network Address Translation), 302

Native templating, 252

Navigability, 59

Net_SMTP PEAR, 294

Node, 68

NuSOAP, 232

NuSpherePHPEd, 569

O

Object, 27

 diagram, 98

ODBC, 179

OpenBSD, 404

Operation, 57

P

PDF, 534

PEAR, 179; 185; 354; 593

 SOAP, 232

Perfect-engineering time unit, 410

Perforce, 568

Perl Dev Kit, 569

PHPCode, 569

PHPEclipse, 569

PHPEdit, 569

PHPUnit, 328; 331

Polymorphism, 27; 43

PostgreSQL, 34; 173

Property, 28

pserver, 550

Push Down Automation (PDA), 345

Q

Quality assurance, 493

R

RAID-массив, 405

RCS, пакет, 551

Realization, 59

Requirements-gathering phase, 53

rsync, утилита, 516

S

Sablotron, пакет, 533

Samba, 510; 547

Scenario, 54

Self-call, 67

Sequence diagram, 64

Session, 304

Singleton, 173

Smarty, 270; 298

SOAP (Simple Object Access Protocol), 232

Software Configuration Management
(SCM), 568

Starting point, 64

State diagram, 67

Studio Server, 560

Subversion, 552

Swimlane, 64

T

Ten-point plan, 391

Test-driven development, 419

Transition, 64

TrueType, 589

У

UML (Unified Modeling Language), 52
 Unidirectional navigability, 59
 Use case diagram, 54

В

Visibility, 49
 Visual
 SourceSafe, 514
 Studio .NET, 548
 VoiceXML, 249

W

Web-сервер, 403
 Web-служба, 231; 236
 Widget, 101
 WinCVS, 550
 WSDL (Web Service Description Language), 232
 WW-SFAT, учебный проект, 359

Х

XML, 231

З

ZDE, 555
 Zend Studio, 555
 ZLib, 589

А

Абстрактная функция, 47
 Абстрактный класс, 51
 Автоинкрементное поле, 183
 Агрегация, 60
 Анализ производительности, 564
 Ассоциативный массив, 272
 Ассоциация, 58
 Атрибут, 27; 28; 57

Б

База данных, 172
 уровень абстракции, 173

Балансировщик нагрузки, 404
 Бизнес-логика, 29
 Блок активации, 67
 Брандмауэр, 302; 405

В

Ветвление, 552
 Вид, 249
 Видимость, 49
 Виртуальный сервер, 546
 Внутренняя переменная-член, 32
 Возвратное тестирование, 335
 Время
 обработки сценария, 572
 ожидания сеанса, 304

Д

Двоичная разность, 552
 Двунаправленная ассоциация, 59
 Действие, 64
 Дерево решений, 499
 Деструктор, 51
 Диаграмма
 видов деятельности, 63
 классов, 56
 компонентов, 68
 объектов, 98
 последовательностей, 64
 прецедентов, 54
 развертывания, 68
 состояний, 67
 Диспетчер событий, 202
 Документ WSDL, 239
 Дорожка, 64

Е

Естественный шаблон, 252

Ж

Журнал, 216

З

Зависимость, 68
 Задание типов, 103

И

Идеальная реализация, 410
 Идентификатор сеанса, 304
 Иерархия классов, 39
 Извлечение метода, 442
 Инициализация объекта, 33
 Инкапсуляция, 27; 49
 Инспектирование кода, 559
 Инстанцирование, 28
 позднее, 126; 131
 Интегрированная среда разработки, 554
 Интерфейс, 27; 47; 285
 Builder, 116
 ExceptionFactory, 198
 Factory, 193
 Iterator, 140; 141
 IteratorAggregate, 145
 Observer, 101; 102
 Traversable, 142
 и реализация, 329
 общего шлюза, 301
 Информирование пользователя, 281
 Инфраструктура, 250
 Исключающий контроль версий, 541; 549
 Исключительное уведомление, 281
 Искусственный интеллект, 350
 Исполнитель, 54
 Источник данных, 179
 Итерация, 417

К

Калькулятор для обратнойпольской записи, 343
 Карта страниц, 391
 Каскадный процесс разработки, 363; 385
 Качество, 495
 Класс, 25; 27
 AbstractWidget, 102
 API, 347
 Collection, 120
 CollectionIterator, 143
 Communication, 287
 Config, 354
 constraint, 262
 Database, 175; 185
 DataManager, 82

DB, 179
 Debugger, 227
 Dir, 337
 Dispatcher, 204
 DomDocument, 534
 EmailCommunication, 289
 EmailRecipient, 285
 Exception, 123
 FSM, 345
 GenericObject, 150; 155; 326; 523
 GenericObjectCollection, 164; 577
 Home, 163
 Logger, 214; 220
 Recipient, 284
 request, 262
 RichEmailCommunication, 299
 StudentFactory, 126
 TemplatedEmailCommunication, 298
 UserHome, 167
 UserSession, 316; 418
 Variant, 593
 XML_Util, 354
 XSLTProcessor, 534
 абстрактный, 51
 родительский, 38
 Класс-агрегат, 75
 Класс-диспетчер, 202
 Класс-наследник, 38
 Класс-фабрика, 163
 Клиент SOAP, 234
 Клонирование объекта, 148
 Ключевое слово
 class, 29
 extends, 38
 interface, 47
 private, 33
 public, 31
 Комментарий, 68
 Композиция, 60; 75; 94
 Компонент, 251
 Конечный автомат, 343
 Конструктор, 33
 Контроллер, 249
 Контроль версий, 514; 540
 Концепция последовательности, 183
 Краткое описание проекта, 367
 Кратность, 75

Л

Линия жизни объекта, 66
 Лицензия GNU, 550

М

Метод, 27; 28
 __clone(), 148
 доступа, 32; 50; 76
 Моделирование предметной области, 56
 Модель, 250
 IPO, 250
 MVC, 248
 конечных автоматов, 343
 Модульное тестирование, 328; 393; 497

Н

Направление ассоциации, 59
 Наследование, 27; 37
 Начальное состояние, 64
 Нормализация Бойса-Кодда, 530

О

Обеспечение
 безопасности, 208
 качества, 493
 Обобщение, 60; 75
 Обработка событий, 211
 Обработчик события, 204
 Обратный вызов, 127
 Общение с пользователем, 280
 Объединение, 64
 Объект, 24; 27; 28
 Объект-имитатор, 488
 Объектная модель, 50
 Объектно-ориентированное программное обеспечение, 24
 Объектно-ориентированный подход, 25
 Однонаправленная ассоциация, 59
 Оператор
 ->, 30
 clone, 147
 foreach, 142
 new, 29
 вызова статического метода, 84

Операция, 57
 Определение требований, 53
 Отношение
 композиции, 60
 реализации, 59
 Отслеживание состояния стека, 562
 Отчет о функциональном тестировании, 498

П

Параллельный контроль версий, 541; 550
 Перегрузка методов, 42
 Переменная \$this, 31
 Перехват идентификатора сеанса, 308
 Переход, 64
 План из десяти пунктов, 391
 Поведение, 27
 Повторное использование кода, 25; 38
 Подбор идентификатора сеанса, 306
 Поддержка сеансов, 301
 Позднее инстанцирование, 120; 126; 131
 Полиморфизм, 27; 43; 63
 Полоса синхронизации, 64
 Получатель
 сообщения, 282
 электронной почты, 286
 Пошаговое выполнение программы, 560
 Поэтапная разработка, 510
 Предметная область, 53
 Принцип модульности, 26
 Проверка пользователя, 281
 Программное обеспечение управления версиями, 540
 Проектирование, 390
 Профайлер, 564
 Процесс разработки, 385

Р

Рабочая среда, 512
 Рабочее пространство, 547
 Рабочие данные, 519
 Развёртывание корпоративных приложений, 583
 Разработка
 на основе тестирования, 393; 419
 спецификации, 388
 Разрешение конфликтов, 550

Регистрация событий, 212
 Регулярное выражение, 285
 Редактирование кода, 558
 Резервное копирование, 405
 Репликация данных, 404
 Рефакторинг, 394; 438; 441
 Родительский класс, 38

C

Самовызов, 67
 Свойство, 27
 Сеанс, 302
 Сервер
 баз данных, 403
 разработки, 398; 509
 хранения, 547
 Система
 генерации отчетов, 520
 параллельного контроля версий, 550
 управления ошибками, 501
 Системная архитектура, 398
 Событие, 200
 Сообщение, 238; 283
 Сохранение сеанса, 304
 Спиральный процесс, 385
 Среда разработки, 554
 Стек, 344
 Структурное программирование, 25
 Сценарий, 54; 251
 использования, 408
 тестирования, 392

T

Таблица переходов, 345
 Тегирование, 553
 Тест Тьюринга, 350
 Тестирование, 392
 модульное, 498
 нагрузки, 392; 403; 499; 581
 удобства использования, 500
 Тестовая стратегия, 328
 Тестовый
 пакет, 330
 сервер, 546
 Техническая спецификация, 389

Точка
 ветвления, 64
 останова, 562
 прерывания, 560
 Транзакция, 189; 198
 Трудоемкость реализации, 417

Y

Уведомление пользователя, 280
 Удобство использования, 496; 500
 Узел, 68
 Унифицированный язык моделирования, 52
 Уничтожение объектов, 34
 Управление
 изменениями, 395
 проектом, 367
 сеансами, 312; 316
 событиями, 200
 Управляющий элемент, 101
 Уровень
 абстракции базы данных, 173
 видимости, 49

Ф

Фаза проектирования, 385
 Фасад, 112
 Формальное описание проекта, 370
 Формулировка бизнес-требований, 370
 Функциональная спецификация, 388
 Функциональное тестирование, 392; 498
 Функция, 28
 call_user_func(), 128
 func_get_args(), 44
 is_callable(), 131
 parse_ini_file(), 353
 parse_url(), 219
 session_start(), 312
 var_dump(), 180
 абстрактная, 47

X

Хеш-таблица, 272
 Хранилище, 541

Ц

Целостность ссылок, 190

Ч

Чат-бот, 350

Ш

Шаблон проектирования, 70; 94; 249

Builder, 113

Composite, 95

Decorator, 106

Director, 116

Facade, 112

Factory, 193

Observer, 100

Reactor, 211

Singleton, 173; 190; 215

модель–вид–контроллер, 248

Э

Экземпляр класса, 27

Эксперт по предметной области, 53

Экстремальное программирование, 361;
393; 407

Электронный отчет, 521

Я

Язык описания Web-служб, 232

Научно-популярное издание

Эд Леки-Томпсон, Алек Коув, Стивен Новицки, Хъяо Айде-Гудман

PHP 5 для профессионалов

Литературный редактор *Е.П. Перестьюк*

Верстка *В.И. Бордюк*

Художественные редакторы *О.Л. Васilenко, В.Г. Павлютин*

Корректоры *Т.А. Корзун, О.В. Мишутина*

Издательский дом “Вильямс”
101509, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 13.07.2006. Формат 70x100/16.

Гарнитура Times. Печать офсетная.

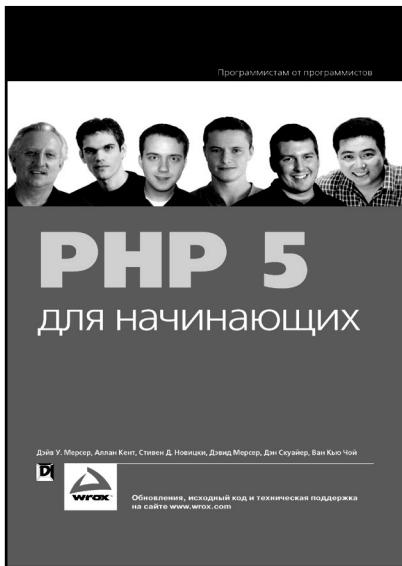
Усл. печ. л. 38,0. Уч.-изд. л. 35,02.

Тираж 3 000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15

PHP 5 ДЛЯ НАЧИНАЮЩИХ

**Дэйв У. Мерсер
Аллан Кент
Стивен Д. Новицки
Дэвид Мерсер
Дэн Скуайер
Ван Кью Чой**



www.dialektika.com

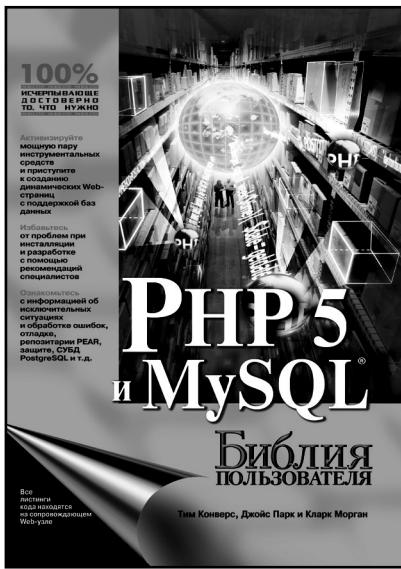
Эта книга представляет собой подробное учебное пособие для желающих освоить современную версию PHP. В книге описывается установка и конфигурирование PHP, основные понятия программирования, такие как переменные, циклы, условные операторы и массивы, а также основы объектно-ориентированного программирования и возможности его применения в PHP 5. Здесь также рассматриваются такие темы, как работа из PHP с HTTP-данными, использование XML, СУРБД (MySQL и SQLite), работа с изображениями и создание PHP-сценариев командной строки. Кроме того, в книге описано проектирование приложений с помощью UML, PEAR-пакеты и методика повторного использования PHP-кода, а также обработка ошибок, тестирование и отладка приложений.

ISBN 5-8459-1039-0

в продаже

PHP 5 И MYSQL БИБЛИЯ ПОЛЬЗОВАТЕЛЯ

**Тим Конверс,
Джойс Парк
и Кларк Морган**



www.dialektika.com

В книге приведены исчерпывающие сведения по созданию динамических Web-узлов на основе программных средств с открытым исходным кодом: языка PHP, сервера Apache и СУБД MySQL, а также показано, как обеспечить бесперебойную эксплуатацию таких узлов под управлением операционных систем Windows или Linux. Многочисленные сценарии и готовые программы, представленные в книге, подробно описаны, тщательно прокомментированы и составляют основу практически значимых приложений. Книга дополняет электронную документацию, содержит все необходимые справочные данные и рассчитана на широкий круг читателей.

ISBN 5-8459-1022-6

в продаже

PHP 5 для "ЧАЙНИКОВ"

Джанет Валейд



www.dialektika.com

Данная книга является введением в область Web-программирования на языке PHP 5. С ее помощью можно быстро написать сценарий для Web, обеспечить взаимодействие с файлами и базами данных, а также решить другие задачи. Материал книги также позволит избежать многих распространенных ошибок. Описание основных возможностей языка сопровождается примерами. В книге можно также найти рекомендации по установке модуля PHP 5 для Web и для работы в командной строке, а также установке и настройке популярных Web-серверов Apache и IIS. Данная книга будет полезна для начинающих разработчиков, а также всех тех, кто интересуется вопросами программирования для Web.

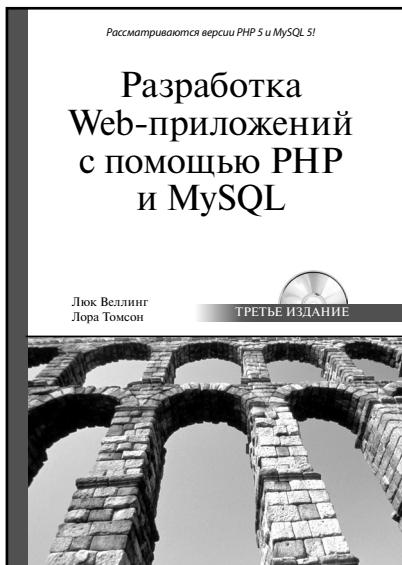
ISBN 5-8459-0851-5

в продаже

РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ PHP И MYSQL

3-е издание

**Люк Веллинг,
Лора Томсон**



www.williamspublishing.com

В книге подробно описано применение PHP и MySQL для построения крупных коммерческих Web-сайтов. Основное внимание уделяется реальным приложениям. Здесь рассматриваются как простые интерактивные системы приема заказов, так и различные аспекты электронных систем продажи и безопасности во взаимосвязи с созданием реального Web-сайта. Подробно описаны все стадии разработки множества типовых проектов на PHP и MySQL, в числе которых, помимо прочих, система управления содержимым, почтовый Web-сайт, приложение поддержки Web-форумов и электронный книжный магазин. Основное отличие этого издания от предыдущего состоит в том, что материалы и весь исходный код полностью переписаны для новых версий PHP5 и MySQL 5.0. Книга ориентирована на профессиональных разработчиков, но будет полезной и для начинающих программистов.

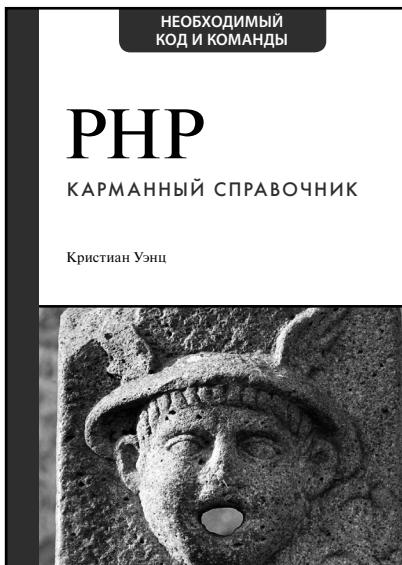
ISBN 5-8459-0862-0

в продаже

PHP

КАРМАННЫЙ СПРАВОЧНИК

Кристиан Уэнц



www.williamspublishing.com

НЕОБХОДИМЫЙ
КОД И КОМАНДЫ

PHP

КАРМАННЫЙ СПРАВОЧНИК

Кристиан Уэнц

Данная книга построена в форме набора решенных задач. Все решения представлены в форме кода, сопровождаемого подробным разбором всех действий и рекомендациями по решению сходных задач. Разобраны наиболее важные функции PHP, приводятся примеры их использования. Материал разбит по основным темам, связанным с использованием PHP: работа со строками и массивами, датой и временем, Web-формами, HTTP и FTP-серверами, а также различными базами данных.

Кристиан Уэнц (Christian Wenz) — автор, преподаватель и консультант по вопросам, связанным с Web-технологиями. Он является автором или соавтором более 50 книг, часто пишет статьи для журналов информационных технологий и выступает на конференциях по всему миру. Кристиан участвовал в разработке нескольких пакетов PHP в архиве PEAR, является основателем PHP Security Consortium и первым гражданином Германии, получившим звание Zend Certified Professional.

ISBN 5-8459-0973-2

в продаже

ПОЛНЫЙ СПРАВОЧНИК ПО MYSQL

Викрам Васвани



www.williamspublishing.com

ISBN 5-8459-0979-1

Книга известного профессионала в области компьютерных технологий посвящена новой версии популярной в настоящее время СУБД MySQL. Подробно рассматриваются такие вопросы, как установка и настройка СУБД MySQL, администрирование и оптимизация работы сервера. Книга содержит детальное описание диалекта SQL, который используется в MySQL. Подробно описаны команды, предназначенные для создания и управления базами данных и таблицами, добавления, модификации и удаления записей. Не забыты и разработчики приложений для MySQL. Эта книга содержит специальный раздел, предназначенный для разработки приложений, работающих с MySQL.

Книга рассчитана на разработчиков и администраторов разной квалификации, а также может быть полезна для студентов и преподавателей соответствующих специальностей.

в продаже