Hibernate

Hibernate — один из первых инструментов объектно-реляционного отображения (Object-Relational Mapping, ORM) данных для Java-окружения. Целью Hibernate является освобождение разработчика от большинства общих работ, связанных с задачами получения, сохранения данных в СУБД. Эта технология помогает удалить или инкапсулировать зависящий от поставщика SQL-код, а также решает стандартную задачу преобразования типов Java-данных в типы данных SQL и наборов данных из табличного представления в объекты Java-классов. Hibernate - это механизм отображения в реляционной базе данных объектов java.

ORM (*объектно-реляционное отображение*) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных»

Установка

- Загрузить и установить сервер баз данных MySQL или Oracle.
- 2. Загрузить и подключить Hibernate с сервера **http://hibernate.org/**
- 3. Загрузить и подключить JDBC-драйвер используемой базы данных
- (в случае использования Ant, в папку **lib** проекта), в данном случае
- **mysql-connector-java-[***версия***].jar** или **ojdbc[***версия***].jar**. Обычно JDBC-
- драйвер предоставляется на сайте разработчика сервера баз данных.
- Библиотеки **hibernate3.**[*версия*]**.jar** и **hibernate-annotations-3.**[*версия*]**.jar**
- являются основными. Кроме них, устанавливается еще целый ряд необходи-
- мых библиотек.

Hibernate

- Для решения своих задач Hibernate должен получить от разработчика следующую информацию:
- Параметры соединения с базой данных. В общем-то от этого никуда нам не уйти. Пакет пакетом, но соединение с базой надо как-то указывать
- Описание отображения классов на таблицы. Данное описание включает в себя связь между колонкой в таблице и полем в классе. Если вспомнить тех же студентов, то для каждого поля в классе Student у нас было поле в таблице. Вот это нам и надо будет сделать.
- Описание отношений между классами

Hibernate

- Поначалу все эти сложности покажутся излишними — ну в конце концов, что нам стоит написать несколько SQL-запросов на получение данных — добавление, запрос, исправление и удаление. Но если у вас таблиц не 3-4, а 500-600 ? И если вам потребуется добавить какое-то поле и связь между таблицами ? Такого рода исправления становятся не тривиальной задачей, которая требует много времени и последующего тестирования.
  С другой стороны обольщаться не стоит — Hibernate не так уж и здорово решает абсолютно все задачи. Например, сложные запросы, которые часто требуются для создания головоломных отчетов, часто проще и эффективнее сделать на SQL. Hibernate позволяет это сделать — в нем можно создать обычный SQL-запрос. Также массовые изменения в нем делаются не очень эффективно. И еще я бы выделил не всегда удачную реализацию отношений — особенно это касается отношения многие-ко-многим — при редактировани такого списка Hibernate просто удаляет все старые отношения и заменяет их на новые. В общем смотрите, сравнивайте, оптимизируйте.
  Но все-таки Hibernate существенно облегчает работу с базой данных и значительно упрощает код Вашего приложения.

Hibernate. Example

- P 856

Java Persistence API

Java Persistence API (*JPA*) — **API**, входящий с версии Java 5 в состав платформ Java Se и Java EE, предоставляет возможность сохранять в удобном виде Java-объекты в базе данных[1].
Существует несколько реализаций этого интерфейса, одна из самых популярных использует для этого Hibernate. JPA реализует концепцию ORM.
Поддержка сохранности данных, предоставляемая *JPA*, покрывает области:

- непосредственно API, заданный в пакете javax.persistence.;
- платформо-независимый объектно-ориентированный язык запросов JAVA Persistence Query Language;
- метаинформация, описывающая связи между объектами.
- Генерация DDL для сущностей

Сервлеты

- **Сервлет** является интерфейсом Java, реализация которого расширяет функциональные возможности сервера. Сервлет взаимодействует с клиентами посредством принципа запрос-ответ.
- Хотя сервлеты могут обслуживать любые запросы, они обычно используются для расширения веб- серверов. Для таких приложений технология Java Servlet определяет HTTP-специфичные сервлет классы.

Сервлеты

- Сервлет:
- ——компонент приложений Java Enterprise Edition;
- ——загружается веб-сервером в контейнер;
- ——выполняется на стороне сервера;
- ——обрабатывает клиентские запросы;
- ——динамически генерирует ответы на запросы;
- ——находится в состоянии ожидания, если запросы отсутствуют;
- ——принимает запросы от других сервлетов (Servlet chaining);
- ——поддерживает соединения с ресурсами

Сервлеты

- Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP.
- Контейнер сервлетов поддерживает также протокол HTTPS (HTTP и SSL) для защищаемых запросов.
- Сервлеты в промышленном программировании используются для:
- ——приема входящих данных от клиента;
- ——взаимодействия с бизнес-логикой системы;
- ——динамической генерации ответа клиенту.

Сервлеты

- Жизненный цикл сервлета начинается с его инициализации и загрузки в память контейнером сервлетов при старте контейнера либо в ответ на первый клиентский запрос. Сервлет готов к обслуживанию любого числа запросов. Завершение существования происходит при выгрузке его из контейнера.
- Первым вызывается метод **init()**.

Сервлеты

- После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом **service(HttpServletRequest request, HttpServletResponse response)** сервлета, вызываемый контейнером, а все параметры запроса упаковываются в экзем пляр **request** интерфейса **HttpServletRequest**, передаваемый в сервлет. Еще одним параметром этого метода является экземпляр **response** интерфейса **HttpServletResponse**, в который загружается информация для передачи клиенту. Для каждого нового клиента при обращении к сервлету создается независимый поток, в котором производится вызов метода **service()**. Метод **service()** предназначен для одновременной обработки множества запросов.

Сервлеты

- При выгрузке приложения из контейнера, то есть по окончании жизненного цикла сервлета, вызывается метод **destroy()**, в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

Сервлеты

- Жизненный цикл сервлета
- В случае отсутствия сервлета в контейнере.
  - Класс сервлета загружается контейнером.
  - Контейнер создает экземпляр класса сервлета.
  - Контейнер вызывает метод init(). Этот метод инициализирует сервлет и вызывается в первую очередь, до того, как сервлет сможет обслуживать запросы. За весь жизненный цикл метод init() вызывается только один раз. Сервлет может выбросить UnavailableException или ServletException на этом этапе. В обоих случаях сервлет не будет создан. В случае UnavailableException контейнер сервлетов попробует выполнить процедуру инициализации снова через время, указанное в исключении.
- Обслуживание клиентского запроса. Каждый запрос обрабатывается в своем отдельном потоке. Контейнер вызывает метод service() для каждого запроса. Этот метод определяет тип пришедшего запроса и распределяет его в соответствующий этому типу метод для обработки запроса. Разработчик сервлета должен предоставить реализацию для этих методов. Если поступил запрос, метод для которого не реализован, вызывается метод родительского класса и обычно завершается возвращением ошибки инициатору запроса.
- В случае если контейнеру необходимо удалить сервлет, он вызывает метод destroy(), который снимает сервлет из эксплуатации. Подобно методу init(), этот метод тоже вызывается единожды за весь цикл сервлета.

Методы HttpServlet

- doGet for handling HTTP GET requests
- doPost for handling HTTP POST requests
- doPut for handling HTTP PUT requests
- doDelete for handling HTTP DELETE requests
- doHead for handling HTTP HEAD requests
- doOptions for handling HTTP OPTIONS requests
- doTrace for handling HTTP TRACE requests (p 457)

Сервлеты

- Example
- ex2

JSP

- Технология Java Server Pages (JSP) обеспечивает разделение динамическойи статической частей страницы, результатом чего является возможность изменения дизайна страницы, не затрагивая динамическое содержание. Это свойство используется при разработке и поддержке страниц, так как дизайне рам нет необходимости знать, как работать с динамическими данными. JSP-код состоит из специальных тегов и выражений, которые указывают контейнеру соответствие между тегами и java-кодом для генерации сервлета
- или его части. Таким образом поддерживается документ, который одновремен но содержит и статическую страницу, и теги Java, управляющие страницей. Статические части HTML-страниц посылаются в виде строк в метод **write()**. Динамические части включаются прямо в код сервлета. С момента формирования ответа сервера страница ведет себя как обычная HTML-страница с ассоциированным сервлетом.
- Чтобы создать простейшую JSP, достаточно взять HTML-страницу и заменить расширение **html** на **jsp**. Только в этом случае для запуска страницы необходим специальный application server с контейнером сервлетов и особое размещение самой страницы.

JSP

- Страницы JSP и сервлеты никогда не следует использовать в информационных системах друг без друга. Причиной являются принципиально различные роли, ко-
- торые играют данные компоненты в приложении. Страница JSP ответственна за формирование пользовательского интерфейса и отображение информации, переданной с сервера. Сервлет выполняет роль контроллера запросов и ответов, то есть принимает запросы от всех связанных с ним JSP-страниц, вызывает соответствующую бизнес-логику для их (запросов) обработки и в зависимости от результата выполнения решает, какую JSP поставить этому результату в соответствие. При необходимости расширения функциональности системы *не следует* создавать дополнительные сервлеты. Сервлет в приложении должен быть *один*. При создании нового учебного приложения во избежание путаницы всегда следует создавать новый проект. Для демонстрации взаимодействия JSP-страниц и сервлета будет решена задача определения времени между загрузкой страницы в браузер и нажатием кнопки на этой же странице. Страница JSP с последующим вызовом другой JSP-страницы, отображающей результаты выполнения запроса.

JSP

- Example ex3

Google Web Toolkit (GWT)

- **Google Web Toolkit** (**GWT**, ˈgwit) — свободный Java-фреймворк, который позволяет веб-разработчикам создавать Ajax-приложения. Его особенность - это компилятор Java -> JavaScript, позволяющий почти всю разработку клиента и сервера реализовать на основе Java и лишь на последнем этапе создать соответствующие JavaScript, HTML и CSS. Выпускается под лицензией Apache версии 2.0. GWT делает акцент на повторное использование и кросс-браузерную совместимость.

GWT

- Используя GWT, разработчики могут быстро писать и отлаживать AJAX приложения на языке Java, используя инструментарий отладки Java. Компилятор GWT переведёт код Java приложения соответствующему браузеру JavaScript,HTML и CSS.
- Кроме того, GWT оснащен XML парсером, поддержкой интернационализации, интерфейсом для удаленного вызова процедур, интеграцией JUnit и небольшим пакетом виджетов для разработки элементов графического интерфейса пользователя (GUI). Они могут быть созданы аналогично тому, как это делается с помощью пакета Swing.
- Отладка GWT-приложения разделена на две части: отладка серверной части приложения осуществляется как отладка обычного Java web-приложения, для отладки клиентской части понадобится gwt dev-plugin для браузера.
- Утилита командной строки webAppCreator, поставляемая вместе с GWT, автоматически создает все файлы, необходимые для нового GWT-проекта. Она также позволяет создавать файлы проекта Eclipse.
- Существует подключаемый модуль Google Plugin для IDE Eclipse (начиная с версии 3.3), позволяющий упростить процессы создания GWT-проекта и размещения готовых приложений на сервисе Google App Engine.

- Избегается повторная реализация одних и тех же графических интерфейсов для локальных и для веб приложений с помощью различных технологий, например - RCP для первых и JSF - для вторых. Обычная схема - "stateless клиент / stateful сервер" может быть заменена схемой "stateful клиент / stateless сервер". Это позволяет больше реакций пользователя обрабатывать непосредственно в клиенте. Простой механизм удаленного вызова процедур. В сочетании с предыдущим качеством это позволяет серверу передавать в ответ данные, а не HTML. Сервер при этом может быть любой - не обязательно тот, который передал первую картинку. Для передачи сложных данных может использоваться как стандартный RCP - передаче подлежат ериализируемые Java-объекты, так и тексты на XML или JSON. Динамические и многоразовые компоненты пользовательского интерфейса(виджеты): Программисты могут использовать заранее разработанные классы для реализации трудоемких элементов динамического поведения, таких, как drag-and-drop, или сложных визуальных структур. Управление историей браузера Поддержка полнофункциональной Java отладки
- GWT устраняет некоторые кросс-браузерные проблемы разработки.
- JUnit-интеграция
- Поддержка интернационализации и локализации.
- Поддержка HTML CANVAS(с учётом изменений API)
- Разработчики могут вставлять готовые JavaScript-фрагменты в исходный Java-текст, применяя JavaScript Native Interface (JSNI).
- Поддержка использования Google API GEARS в приложениях GWT
- Программное обеспечение с открытым исходным кодом
- Разработчики могут проектировать и разрабатывать приложения в чистом объектно-ориентированном стиле, так как они используют Java (вместо JavaScript). Обычные JavaScript-ошибки, такие, как опечатки и несоотвтетствие типов, обрабатываются во время компиляции.
- JavaScript, генерируемый GWT-компилятором, может быть разделен на фрагменты. Это не только дает возможность лучше понимать его, но и экономит время загрузки приложения - первый фрагмент может начать работать, не дожидаясь пока весь JavaScript текст будет загружен
- Ряд библиотек доступны для GWT, от Google или третьих лиц. Это расширяет функциональные возможности GWT.

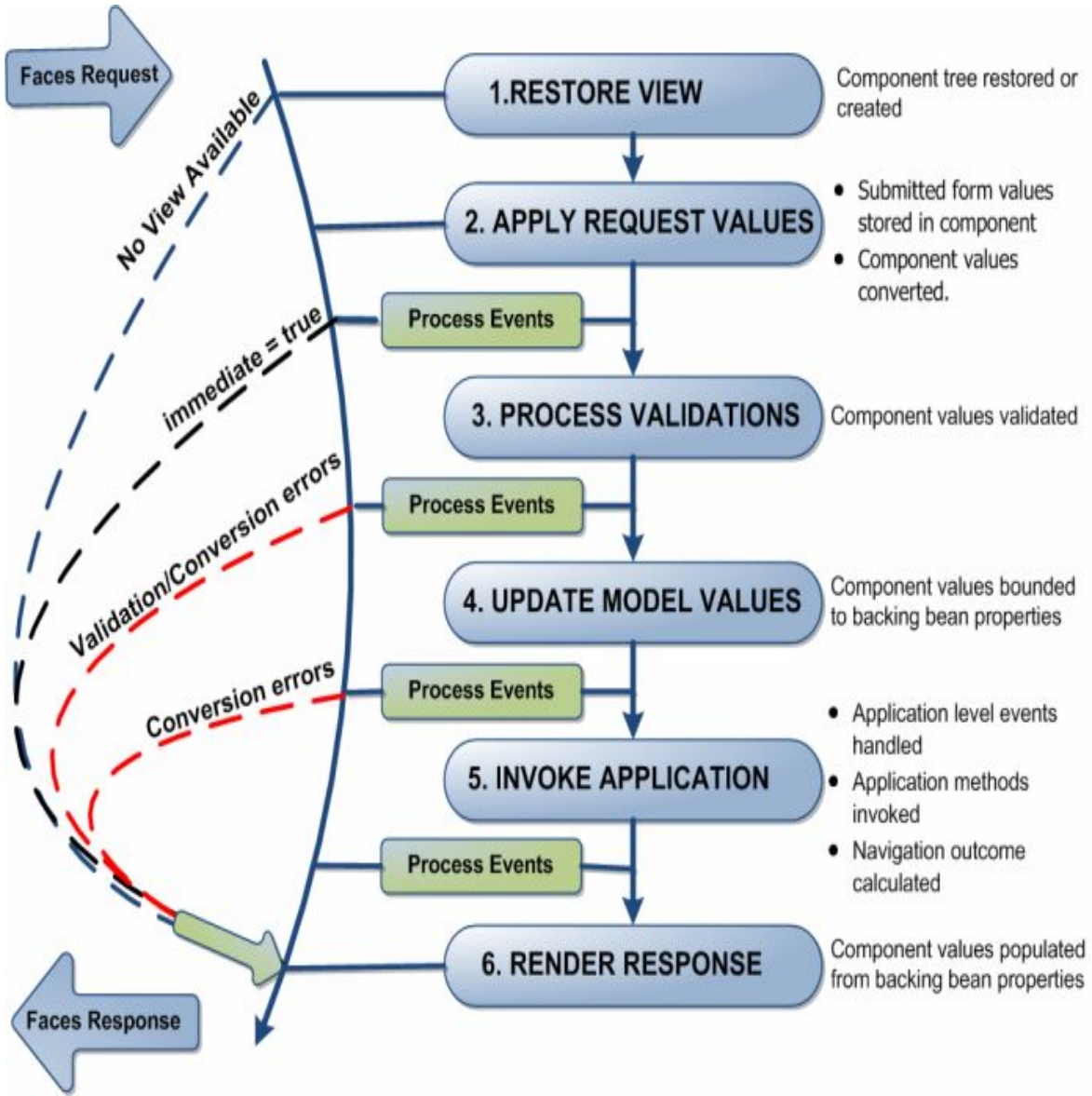
JSF

- JavaServer Faces (JSF) - это платформа разработки интерфейса пользователя для веб-приложений Java. Она призвана значительно упростить процесс создания и поддержки приложений, работающих на сервере приложений Java и визуализирующих свои пользовательские интерфейсы на целевом клиенте. JSF обеспечивает простоту использования благодаря целому ряду факторов:
- упрощает формирование пользовательского интерфейса из набора повторно используемых компонентов пользовательского интерфейса;
- упрощает перенос данных приложения в пользовательский интерфейс и из него;
- помогает управлять состоянием пользовательского интерфейса при запросах к серверу;
- предоставляет простую модель установления связи между созданными клиентом событиями и кодом приложения на стороне сервера;
- упрощает сборку и повторное использование компонентов пользовательского интерфейса.

JSF

- Технология **JavaServer Faces** включает:
- Набор API для представления компонент пользовательского интерфейса (UI) и управления их состоянием, обработкой событий и валидацией вводимой информации, определения навигации, а также поддержку интернационализации (i18n) и доступности (accessibility).
- Специальная библиотека JSP тегов для выражения интерфейса JSF на JSP странице. В JSF 2.0 в качестве обработчика представления используется технология Facelets которая пришла на замену JSP.

JSF Lifecycle



Серверы приложений

- **Сервер приложений** (application server) — это программная платформа (framework), предназначенная для эффективного исполнения процедур (программ, скриптов), на которых построены приложения. Сервер приложений действует как набор компонентов, доступных разработчику программного обеспечения через API (интерфейс прикладного программирования), определённый самой платформой.
- Для веб-приложений основная задача компонентов сервера — обеспечивать создание динамических страниц. Однако современные серверы приложений включают в себя и поддержку кластеризации, повышенную отказоустойчивость,балансировку нагрузки, позволяя таким образом разработчикам сфокусироваться только на реализации бизнес-логики.
- В случае сервера приложений Java, сервер приложений ведёт себя как расширенная виртуальная машина для запуска приложений, прозрачно управляя соединениями с базой данных с одной стороны и соединениями с веб-клиентом с другой.

Серверы приложений

- **GlassFish** — Сервер приложений с открытым исходным кодом, реализующий спецификации Java EE, изначально разработанный Sun Microsystems. В настоящее время спонсируется корпорацией Oracle. Актуальная версия платформы называется Oracle GlassFish Server.
- В основу GlassFish легли части кода Java System Application Server компании Sun и ORM TopLink (решение для хранения Java объектов в реляционных БД, предоставленное Oracle). В качестве сервлет-контейнера в нём используется модифицированныйApache Tomcate, дополненный компонентом Grizzly, использующим технологию Java NIO.

Серверы приложений

- **Tomcat** (в старых версиях — **Catalina**) —контейнер сервлетов с открытым исходным кодом, разрабатываемый Apache Software Foundation. Реализует спецификацию сервлетов и спецификацию (JSP) и (JSF). Написан на языке Java.
- **Tomcat** позволяет запускать веб-прилодения , содержит ряд программ для самоконфигурирования.
- Tomcat используется в качестве самостоятельного веб-сервера, в качестве сервера контента в сочетании с веб-сервером Apache HTTP Server, а также в качестве контейнера сервлетов в серверах приложений JBoss и GlassFish.

Серверы приложений

- **WebSphere**

- WebSphere is a set of Java-based tools from IBM that allows customers to create and manage sophisticated business Web sites. The central WebSphere tool is the WebSphere Application Server (WAS), an application server that a customer can use to connect Web site users with Java applications or servlets. Servlets are Java programs that run on the server rather than on the user's computer as Java applets do. Servlets can be developed to replace traditional common gateway interface (cgi) scripts, usually written in C or Practical Extraction and Reporting Language, and run much faster because all user requests run in the same process space.

Серверы приложений

- **WebLogic** — семейство продуктов от одноимённой компании, поглощённой в 1998 году BEA Systems, а с 2008 года принадлежащей корпорации Oracle. В платформу *WebLogic Suite* входят сервер приложенийJ2EE (*Weblogic Server*), портал, интеграционные продукты, средства для разработки приложений и JRockit — собственная JVM компании.
- Последняя финальная версияплатформы WebLogic — 12c, работает на большинстве распространённых операционных систем, включая UNIX, Linuxи Microsoft Windows. Поддерживает следующие стандарты и технологии:

ПОТОКИ ВЫПОЛНЕНИЯ

- К большинству современных распределенных приложений (Rich Client)
- и веб-приложений (Thin Client) выдвигаются требования одновременной поддержки многих пользователей, каждому из которых выделяется отдельный поток, а также разделения и параллельной обработки информационных ресурсов.
- Потоки — средство, которое помогает организовать одновременное выполнение нескольких задач, каждой в независимом потоке. Потоки представляют собой экземпляры классов, каждый из которых запускается и функционирует
- самостоятельно, автономно (или относительно автономно) от главного потока
- выполнения программы. Существует два способа создания и запуска потока:
- на основе расширения класса **Thread** или реализации интерфейса **Runnable**.

ПОТОКИ ВЫПОЛНЕНИЯ

```
public class TalkThread extends Thread {
@Override
public void run() {
for (int i = 0; i < 10; i++) {
System.out.println("Talking");
try {
Thread.sleep(7); // остановка на 7 миллисекунд
} catch (InterruptedException e) {
System.err.print(e);
}
}
}
}
```

ПОТОКИ ВЫПОЛНЕНИЯ

- Example
- ex4

Жизненный цикл потока

- Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления **Thread.State**:
- **NEW** — поток создан, но еще не запущен;
- **RUNNABLE** — поток выполняется;
- **BLOCKED** — поток блокирован;
- **WAITING** — поток ждет окончания работы другого потока;
- **TIMED_WAITING** — поток некоторое время ждет окончания другого потока;
- **TERMINATED** — поток завершен.
- Получить текущее значение состояния потока можно вызовом метода
- **getState().**

Жизненный цикл потока

- Поток переходит в состояние «неработоспособный» в режиме ожидания (**WAITING**) вызовом методов **join(), wait(),** suspend() (deprecated-метод) или методов ввода/вывода, которые предполагают задержку. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания по времени (**TIMED_WAITING**) с помощью методов **yield(), sleep(long millis), join(long timeout)** и **wait(long timeout),** при выполнении которых может генерироваться прерывание **InterruptedException**. Вернуть потоку работоспособность после вызова метода suspend() можно методом resume() (deprecated-метод), а после вызова метода **wait()** — методами **notify()** или **notifyAll().** Поток переходит в «пассивное» состояние (**TERMINATED**), если вызваны методы **interrupt(),** stop() (deprecated-метод) или метод **run()** завершил выполнение, и запустить его повторно уже невозможно. После этого, чтобы запустить поток, необходимо создать новый объект потока. Метод **interrupt()** успешно завершает поток, если он находится в состоянии «работоспособный». Если же поток неработоспособен, например, находится в состоянии **TIMED_WAITING**, то метод инициирует исключение **InterruptedException**. Чтобы это не происходило, следует предварительно вызвать метод **isInterrupted(),** который проверит возможность завершения работы потока. При разработке не следует использовать методы принудительной остановки потока, так как возможны проблемы с закрытием ресурсов и другими внешними объектами.
- Методы suspend(), resume() и stop() являются deprecated-методами и запрещены
- к использованию, так как они не являются в полной мере «потокобезопасными».

Управление приоритетами и группы потоков

- Потоку можно назначить приоритет от **1** (константа **MIN_PRIORITY**)
- до **10** (**MAX_PRIORITY**) с помощью метода **setPriority(int prior).** Получить
- значение приоритета потока можно с помощью метода **getPriority().**

Управление приоритетами и группы потоков

- Example
- ex5

Управление потоками

- Приостановить (задержать) выполнение потока можно с помощью метода **sleep(int millis)** класса **Thread**. Менее надежный альтернативный способ состоит в вызове метода **yield(),** который может сделать некоторую паузу и позволяет другим потокам начать выполнение своей задачи. Метод **join()** блокирует работу потока, в котором он вызван, до тех пор, пока не будет закончено выполнение вызывающего метод потока или не истечет время ожидания при обращении к методу **join(long timeout).**

Управление потоками

- Example
- ex6

Потоки–демоны

- Потоки-демоны используются для работы в фоновом режиме вместе с программой, но не являются неотъемлемой частью логики программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон. С помощью метода **setDaemon(boolean value),** вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод **boolean isDaemon()** позволяет определить, является ли указанный поток демоном или нет.

Потоки–демоны

- Example
- ex7

Потоки и исключения

- В процессе функционирования потоки являются в общем случае независимыми друг от друга. Прямым следствием такой независимости будет корректное продолжение работы потока **main** после аварийной остановки запущенного из него потока после генерации исключения.

Потоки и исключения

- **public class** ExceptThread **extends** Thread {
- **public void** run() {
- **boolean** flag = **true**;
- **if** (flag) {
- **throw new** RuntimeException();
- }
- System.*out*.println("end of ExceptThread");
- }
- }
- //
- **package** by.bsu.thread;
- **public class** ExceptionThreadDemo {
- **public static void** main(String[ ] args) **throws** InterruptedException {
- **new** ExceptThread().start();
- Thread.sleep(1000);
- System.*out*.println("end of main");
- }
- }

Методы synchronized

- Нередко возникает ситуация, когда несколько потоков имеют доступ к некоторому объекту, проще говоря, пытаются использовать общий ресурс и начинают мешать друг другу. Более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в файл/объект/поток. Для контролирования процесса записи может использоваться разделение ресурса с применением ключевого слова **synchronized**. В качестве примера будет рассмотрен процесс записи информации в файл двумя конкурирующими потоками. В методе **main()** класса **SynchroRun** создаются два потока. В этом же методе создается экземпляр класса **Resource**, содержащий поле типа **FileWriter**, связанное с файлом на диске. Экземпляр **Resource** передается в качестве параметра обоим потокам. Первый поток записывает строку методом **writing()** в экземпляр класса **Resource**. Второй поток также пытается сделать запись строки в тот же самый объект **Resource**. Во избежание одновременной записи такие методы объявляются как **synchronized**.
- Синхронизированный метод изолирует объект, после чего он становится недоступным для других потоков. Изоляция снимается, когда поток

Шаблоны и антишаблоны проектирования

- В задачах проектирования информационных систем, классов, их составляющих, при распределении обязанностей и способов взаимодействия объектов этих классов перед программистом возникает серьезная проблема. Неоптимальный выбор может сделать системы и их отдельные компоненты непригодными для
- поддержки, восприятия, повторного использования и расширения. Систематизация приемов программирования и принципов организации классов получила название шаблона (паттерна).

шаблоны GoF и GRASP,

- GoF и GRASP шаблоны представляют собой две различные точки зрения на организацию классов. GRASP представляют обобщенный взгляд на организацию самих классов и их взаимодействия вне зависимости от целевой задачи, решаемой этими классами. GoF представляют собой рецепты решения конкретных, и при этом достаточно узких, проблем. Тем не менее, если их использовать совместно, то качество и удобочитаемость кода повысятся.

Шаблоны GRASP

- Наиболее общие принципы объектно-ориентированного проектирования,
- применяемого при создании диаграммы классов и распределения обязанностей между ними, систематизированы в шаблонах GRASP (General Responsibility Assignment Software Patterns). Ниже будут сформулированы некоторые базовые способы и некоторые стандартные решения, придерживаясь которых можно создавать хорошо структурированный и понятный код. Прежде чем рассмотреть шаблоны GRASP, следует привести список основных шаблонов, составляющих базовые принципы ООП как парадигмы.

Шаблон Expert

- Все классы делятся на две большие группы: классы-носители информации,
- классы, производящие действия. Классов-носителей значительно меньше, но они
- играют роли отображения сущностей реального мира в системе анимирования
- «неживых» сущностей и, как правило, должны быть представителями Java Beans.
- Классы-носители информации, в общем, не должны выполнять действий по манипулированию значениями своих полей, не считая установки, извлечения и поддержки функциональности по контракту, навязанной классом **Object**.
- При проектировании классов на первом этапе необходимо определить об-
- щий принцип распределения обязанностей между классами проекта, а именно:
- в первую очередь определить кандидатов в информационные *эксперты* —

Шаблон Expert

- Ex 8
- Класс **CurrentStateTest** достаточно серьезно отличается от класса **Test** и откровенно просто воспринимается с первого взгляда. Этот класс может быть объявлен как внутренний класс класса **Test** и при небольших изменениях соответствовать шаблону State из группы шаблонов GoF.
- Преимущества следования шаблону Expert:
- ——сохранение инкапсуляции информации при назначении ответственности классам, которые уже обладают необходимой информацией для обеспече ния своей функциональности;
- ——уклонение от новых зависимостей способствует обеспечению низкой степени связанности между классами (Low Coupling);
- ——добавление соответствующего метода или внутреннего класса способствует
- высокому зацеплению (Highly Cohesive) классов, если класс уже обладает информацией для обеспечения необходимой функциональности.

Шаблон Creator

- Существует большая вероятность того, что класс станет проще, если он будет большую часть своего жизненного цикла ссылаться на создаваемые объекты. После определения информационных экспертов следует определить классы, ответственные за создание нового экземпляра некоторого класса. Следует назначить классу **B** обязанность создавать экземпляры класса **A**, если выполняется одно из следующих условий:
- ——класс **B** содержит или получает данные инициализации (has the initializing data), которые будут передаваться объектам класса **A** при его создании;
- ——класс **B** записывает или активно использует (records or closely uses) экземпляры объектов **A**;
- ——класс **B** агрегирует (aggregate) объекты **A**;
- ——класс **B** содержит (contains) объекты **A**;
- ——классы **B** и **A** относятся к одному и тому же типу, и их экземпляры составляют, агрегируют, содержат или напрямую используют другие экземпляры того же класса.
- Если выполняется одно из указанных условий, то класс **B** – создатель (creator) объектов **A**.
- Инициализация объектов — стандартный процесс. Грамотное распределение обязанностей при проектировании позволяет создать слабо связанные независимые простые классы и компоненты.

Шаблон Low Coupling

- Степень связанности классов определяет, насколько класс связан с другими
- классами и какой информацией о других классах он обладает. При проектировании отношений между классами следует распределить обязанности таким
- образом, чтобы степень связанности оставалась низкой.
- Наличие классов с высокой степенью связанности нежелательно, так как:
- ——изменения в связанных классах приводят к локальным изменениям в дан-
- ном классе;
- ——затрудняется понимание каждого класса в отдельности;
- ——усложняется повторное использование, поскольку для этого требуется до-

Шаблон Low Coupling

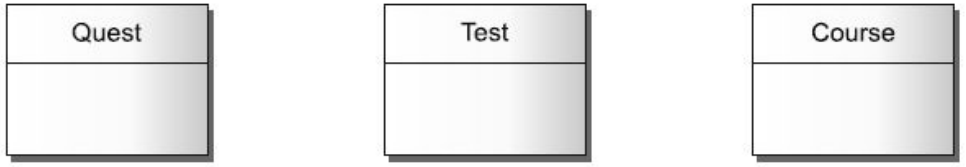


Рис. 20.1. *Классы, которые необходимо связать*

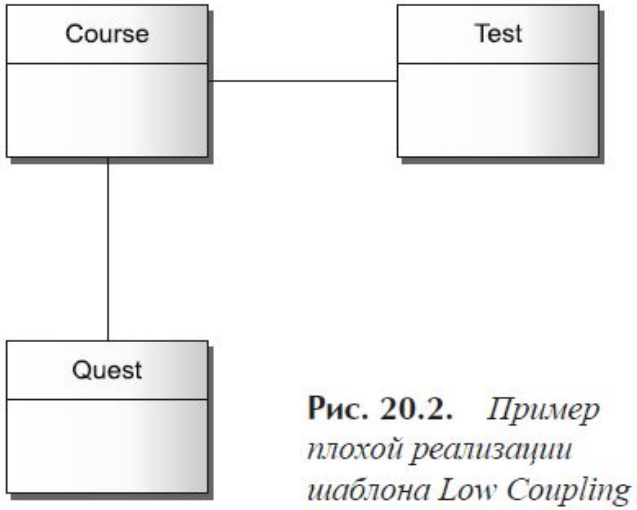


Рис. 20.2. *Пример плохой реализации шаблона Low Coupling*

Шаблон Low Coupling

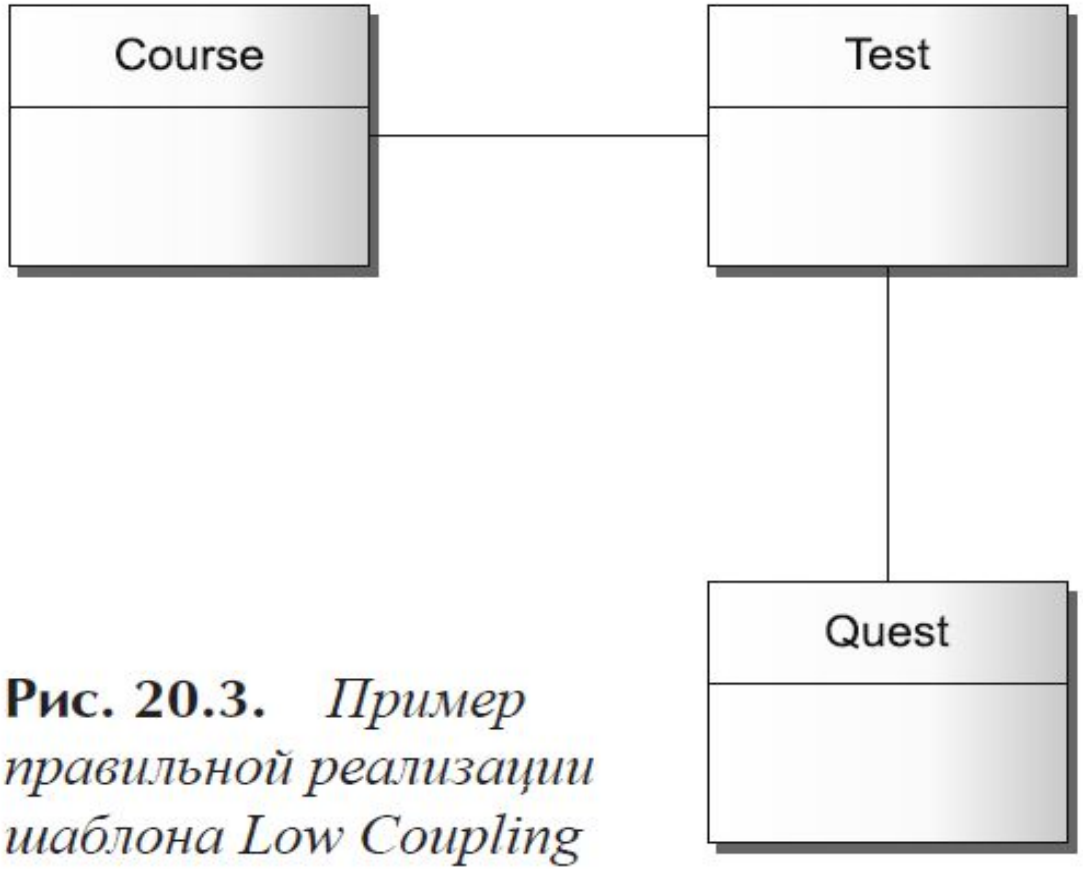


Рис. 20.3. *Пример правильной реализации шаблона Low Coupling*

Шаблон Low Coupling

- Ex10

Шаблон High Cohesion

- С помощью этого шаблона можно обеспечить возможность управления сложностью, распределив обязанности, поддерживая высокую степень зацепления. Зацепление — мера специализированности класса на своих обязанностях.
- При высоком зацеплении обязанности класса тесно связаны между собой,и класс не выполняет работ непомерных объемов. Класс с низкой степенью зацепления выполняет много разнородных действий или не связанных между собой обязанностей.
- Возникают проблемы, связанные с тем, что класс:
- ——труден в понимании, так как необходимо уделять внимание несвязным (не-
- родственным) идеям;
- ——сложен в поддержке и повторном использовании из-за того, что он должен
- быть использован вместе с зависимыми классами;
- ——ненадежен, постоянно подвержен изменениям.
- Классы со слабым зацеплением выполняют обязанности, которые можно
- легко распределить между другими классами.

Шаблон High Cohesion

- ex11

Шаблон Controller

- Одной из базовых задач при проектировании информационных систем является определение класса, отвечающего за обработку системных событий.При необходимости посылки внешнего события прямо объекту приложения, которое обрабатывает это событие, как минимум один из объектов должен содержать ссылку на другой объект, что может послужить причиной очень негибкого дизайна, если обработчик событий зависит от типа источника событий или источник событий зависит от типа обработчика событий. В простейшем случае зависимость между внешним источником событий и внутренним обработчиком событий заключается исключительно в передаче событий. Довольно просто обеспечить необходимую степень независимости между источником событий и обработчиком событий, используя интерфейсы. Интерфейсов может оказаться недостаточно для обеспечения поведенческой независимости между источником и обработчиком событий, когда отношения между этими источником и обработчиком достаточно сложны.

Шаблон Controller

- Согласно шаблону Controller, производится делегирование обязанностей по обработке системных сообщений классу, если он:
- ——представляет всю организацию или всю систему в целом (внешний контроллер);
- ——представляет активный объект из реального мира, который может участво-
- вать в решении задачи (контроллер роли);
- ——представляет искусственный обработчик всех системных событий прецедента и называется *Прецедент***Handler** (контроллер прецедента) .
- Для всех системных событий в рамках одного прецедента используется
- один и тот же контролер.
- Controller — это класс, не относящийся к интерфейсу пользователя и отвечающий за обработку системных событий. Использование объекта-контроллера обеспечивает независимость между внешними источниками событий
- и внутренними обработчиками событий, их типом и поведением. Выбор
- определяется зацеплением и связыванием.

Шаблон Controller