JSR-299: Contexts and Dependency Injection

Общее описание

Спецификация JSR-299: Contexts and Dependency Injection for the Java EE Platform («Контексты и инъекция зависимостей для платформы Java EE», далее — CDI) была принята в декабре 2009 г. Многие идеи были почерпнуты из каркаса разработки веб-приложений Seam, который первый позволял прозрачно вызывать сессионные EJB-компоненты из JSP-страниц и фэйслетов. Эталонная реализация спецификации CDI носит название Weld и входит в состав сервера приложений Glassfish V3.

В спецификации CDI определены требования к новой службе, обязательной в рамках платформы Java EE 6. Данная служба призвана обеспечить 1) «прозрачное» управление жизненным циклом Java EE-компонентов и 2) их гибкое связывание, причем независимо от архитектурного слоя, в котором находятся эти компоненты.

Первое достигается путем универсализации понятия контекста в Java EE-приложении. Раньше контексты запроса, сессии и приложения, обеспечивающие хранение данных в течении соответствующего времени, существовали только в веб-приложениях. В CDI контексты могут содержать любые Java EE-компоненты, создаваемые и удаляемые службой CDI (иначе говоря, CDI-контейнером) согласно времени жизни контекста. Также появляется возможность создавать новые типы контекстов с особым временем жизни.

Гибкое связывание компонентов в рамках CDI обеспечивается едиными механизмами инъекции зависимостей и обработки событий, независимыми от типа компонентов.

По сути, служба CDI разрушает барьеры между EJB- и веб-компонентами, максимально упрощая использование первых в веб-приложениях. Тип компонента, реализующего бизнес-логику (EJB-компонент или управляемый бин JSF), более не влияет на его использование в приложении с точки зрения разработчика — создание (либо обнаружение) и удаление используемых компонентов выполняет служба CDI. Выбор типа компонента (EJB либо JSF) в настоящее время зависит от того, требуется ли для заключенной в компонент бизнес-логики декларативные управление транзакциями, синхронизация обращений и безопасность, либо нет. CDI позволяет обойтись без промежуточных управляемых бинов (JSF), если их единственная задача заключалась в делегировании вызовов от веб-страниц и контроллеров к EJB-компонентам.

Роль службы CDI становится еще более значимой, если учесть, что в Java EE 6 стала возможной упаковка EJB-компонентов в модуль веб-приложения (WAR-файл). Размещение в одном модуле компонентов, построенных на разных технологиях, упрощает разработку, но не должно обманывать: универсального контейнера пока не существует (веб-компоненты обрабатываются веб-контейнером, а EJB-компоненты — EJB-контейнером, соответственно), и именно служба CDI «маскирует» работу с несколькими контейнерами.

Основные концепции

Java EE-компонент является *CDI-компонентом* (bean), если контейнер может управлять ЖЦ его экземпляров в соответствии с моделью ЖЦ *контекстов*, определенной в спецификации CDI. *Экземпляры CDI-компонента* – это объекты, относящиеся к определенному контексту и определяющие состояние и логику приложения. Контейнер создает и удаляет экземпляры CDI-компонентов и связывает их с подходящим контекстом. Экземпляры CDI-компонентов можно инъецировать в другие объекты, выполняющиеся в том же контексте, и использовать в EL-выражениях, вычисляемых в том же контексте.

С CDI-компонентом могут быть связаны метаданные, определяющие его ЖЦ и взаимодействие с другими компонентами. Эти метаданные задаются в виде аннотаций либо в дескрипторе

beans.xml. Наличие дескриптора beans.xml обязательно, если Java EE-модуль использует службу CDI, но если все метаданные задаются с помощью аннотаций, то этот файл может содержать только корневой элемент.

CDI-компонентами могут быть управляемые бины JSF и сессионные EJB-компоненты.

Характеристики CDI-компонента:

1. реализация компонента — как правило, некоторый Java-класс, за исключением ресурсов из окружения Java EE-компонента (для них реализация предоставляется контейнером);

2. типы компонента — множество допустимых типов языка Java, по которым происходит получение зависимостей. У CDI-компонента может быть несколько типов компонента. Например, для сессионного компонента доступны следующие типы компонента: 1) все локальные интерфейсы и их суперинтерфейсы, 2) в случае представления «без интерфейса» класс компонента и все его суперклассы, 3) java.lang.Object;

3. квалификаторы — множество аннотаций, применимых к типу компонента и позволяющих выбрать одну из нескольких альтернативных реализаций либо один из альтернативных способов получения экземпляра CDI-компонента. Для использования в качестве квалификаторов подходят аннотационные типы, помеченные аннотацией @Qualifier. Квалификаторы разрабатываются под специфику конкретного приложения (или предметной области) и соответствуют точкам вариации (variation points) в приложении. Также квалификаторы используются для различения ресурсов одного вида, но с разными параметрами (например, источников данных — ресурсов типа DataSource, — указывающих на разные базы данных);

4. контекст, к которому относятся экземпляры компонента;

5. привязки к перехватчикам вызова методов (interceptors);

6. имя для использования в EL-выражениях (необязательное) — задается аннотацией @Named("beanName").

В CDI есть несколько способов получить нужную зависимость:

1. Инъекция через конструктор, помеченный аннотацией @Inject, при этом контейнер передает в каждый параметр конструктора инъецируемую ссылку. К параметрам конструктора можно применять аннотации-квалификаторы для определения зависимостей с конкретными характеристиками.

2. Инъекция в поле, помеченное аннотацией @Inject и (необязательно) аннотациями-квалификаторами.

3. Инъекция через метод инициализации (initialize method) — любой метод, помеченный аннотацией @Inject, при этом контейнер передает в каждый параметр метода инъецируемую ссылку. К параметрам метода инициализации можно применять аннотации-квалификаторы для определения зависимостей с конкретными характеристиками.

Для инъекции сессионных EJB-компонентов можно использовать как средства CDI (@Inject), так и специфичную для EJB аннотацию @EJB. Для SFSB @Inject обеспечивает удаление экземпляра компонента при удалении соответствующего контекста. Например, CDI-компонент располагается в контексте диалога (является @ConversationScoped) и в него инъецируется ссылка на SFSB, тогда по завершении диалога соответствующий экземпляр SFSB будет удален. При применении @EJB удаление связанного экземпляра SFSB должен выполнить разработчик CDI-компонента либо этот

экземпляр будет удален контейнером по таймауту. (Для всех видов сессионных компонентов контейнер при инъекции @Inject создает клиентский прокси-объект, который позволяет выполнять пассивацию CDI-компонента, содержащего зависимость.)

Экземпляры CDI-компонентов в большинстве случаев создаются контейнером путем вызова конструктора (по умолчанию либо помеченного аннотацией @Inject) класса компонента, помеченного необходимыми квалификаторами. Существует альтернативный способ создания инъецируемых объектов — определение в CDI-компоненте метода (или поля) с аннотацией @Produces (и квалификаторами, при необходимости). Это используется в следующих случаях: 1) создаваемый объект не является экземпляром CDI-компонента (например, источник данных); 2) конкретный тип инъецируемого объекта определяется во время выполнения приложения; 3) необходимо выполнить дополнительную настройку экземпляра компонента сверх вызова конструктора.

При удалении любого экземпляра Java EE-компонента, поддерживающего инъекцию зависимостей, контейнер удаляет все зависимые от него объекты (контекст @Dependent, см. далее).

Контексты

Контекст определяет ЖЦ и доступность всех находящихся в нем экземпляров CDI-компонентов, в т.ч. 1) когда создается новый экземпляр компонента в рамках контекста, 2) когда удаляется существующий в рамках контекста экземпляр компонента, 3) какие инъецированные ссылки указывают на каждый экземпляр компонента в данном контексте.

Для указания, к какому контексту будут относиться экземпляры определенного CDI-компонента, используются аннотации на классе компонента.

В CDI различают нормальные контексты и псевдо-контексты. В нормальном контексте каждому типу компонента (с учетом квалификаторов) соответствует не более одного экземпляра CDI-компонента (то есть контекст представляет собой хранилище «ключ-значение», где ключом является сочетание типа компонента и квалификаторов, а значением – экземпляр компонента). Именно поэтому инъецируемые ссылки на экземпляры компонентов из нормальных контекстов ведут не на сами эти экземпляры, а на клиентские прокси-объекты (client proxy). В псевдо-контексте приведенное выше требование не выполняется.

В спецификации CDI определено четыре нормальных контекста (встроенные контексты): 1) запрос (@RequestScoped), 2) сессия (@SessionScoped), 3) приложение (@ApplicationScoped), 4) диалог (@ConversationScoped), а также единственный псевдо-контекст — @Dependent.

Контексты с пассивным состоянием

Временное перемещение состояния неиспользуемого объекта из оперативной памяти во внешнее хранилище называется пассивацией, а обратный процесс — активацией. Основным требованием к компонентам, поддерживающим пассивацию, является возможность сериализации их состояния для сохранения во внешнем хранилище. Среди CDI-компонентов пассивацию поддерживают сессионные компоненты с состоянием (EJB) и сериализуемые управляемые бины (JSF).

Для каждого компонента, относящегося к поддерживающему пассивацию контексту, контейнер при установке приложения проверяет, что он действительно способен перейти в пассивное состояние (по сути эти требования аналогичны требованиям к сессионным компонентам с состоянием) и, более того, все его зависимости также поддерживают пассивацию. К такого рода зависимостям относятся следующие: 1) все сессионные компоненты, 2) все компоненты, относящиеся к нормальным контекстам, 3) все компоненты из псевдо-контекста @Dependent, поддерживающие пассивацию, 4) все ресурсы, 5) ряд встроенных CDI-компонентов (например,

Event для отправки событий). Поддержка пассивации зависимостей 2го типа обусловлена использованием не прямых ссылок на указанные компоненты, а клиентских прокси-объектов.

Среди нормальных контекстов пассивацию поддерживают контексты сессии и диалога.

Обратите особое внимание на допустимые контексты для различных типов сессионых компонентов. Сессионный компонент без состояния должен принадлежать к псевдо-контексту @Dependent. Сессионный компонент с состоянием может принадлежать к контексту любого типа. Синглтон должен принадлежать к контексту приложения либо псевдо-контексту @Dependent.

Контекст диалога

Контекст диалога активен на всех этапах обработки запросов в веб-приложении, использующем каркас JSF, и предоставляет доступ к состоянию, связанным с определенным диалогом между пользователем и веб-приложением. В данном случае диалог — это некая завершенная по смыслу последовательность действий пользователя с веб-приложением, например, выбор товаров в Интернет-магазине и оформление заказа, покупка авиабилета, подбор конфигурации атомобиля и т.п. Диалог занимает промежуточную позицию по времени жизни между запросом и сессией (сеансом работы с приложением). В вырожденном случае границы диалога совпадают с обработкой единственного запроса, например, при редактировании данных из таблиц в административной панели веб-приложения. При этом в рамках одной сессии может одновременно происходить несколько диалогов, когда пользователь в разных вкладках браузера параллельно выполняет несколько операций с веб-приложением (например, выбирает билеты по различным направлениям).

При обработке каждого запроса в JSF-приложении на этапе восстановления представления (1ый этап обработки запроса) определяется конкретный диалог, к которому относится данный запрос.

Диалог находится в одном из двух состояний: временном (transient) или длительном (long-running). Упрощенная диаграмма состояний (без учета пассивного состояния) приведена на рис. Временное состояние является состоянием по умолчанию. Для перехода в длительное состояние нужно вызвать метод Conversation.begin(), а для возврата во временное — метод Conversation.end(). У каждого длительного диалога есть уникальный строковый идентификатор, который устанавливается самим приложением, либо генерируется контейнером.

Если при завершении обработки запроса в JSF-приложении связанный с этим запросом диалог находится во временном состоянии, то он удаляется вместе с контекстом диалога. Диалог в длительном состоянии сохраняется между последовательными запросами, для этого идентификатор диалога передается в POST-запросах через скрытое поле формы (<h:fom>), а в GET-запросах — с помощью параметра запроса cid.

Если с запросом к JSF-приложению не связан идентификатор диалога, то с этим запросом связывается новый диалог во временном состоянии.

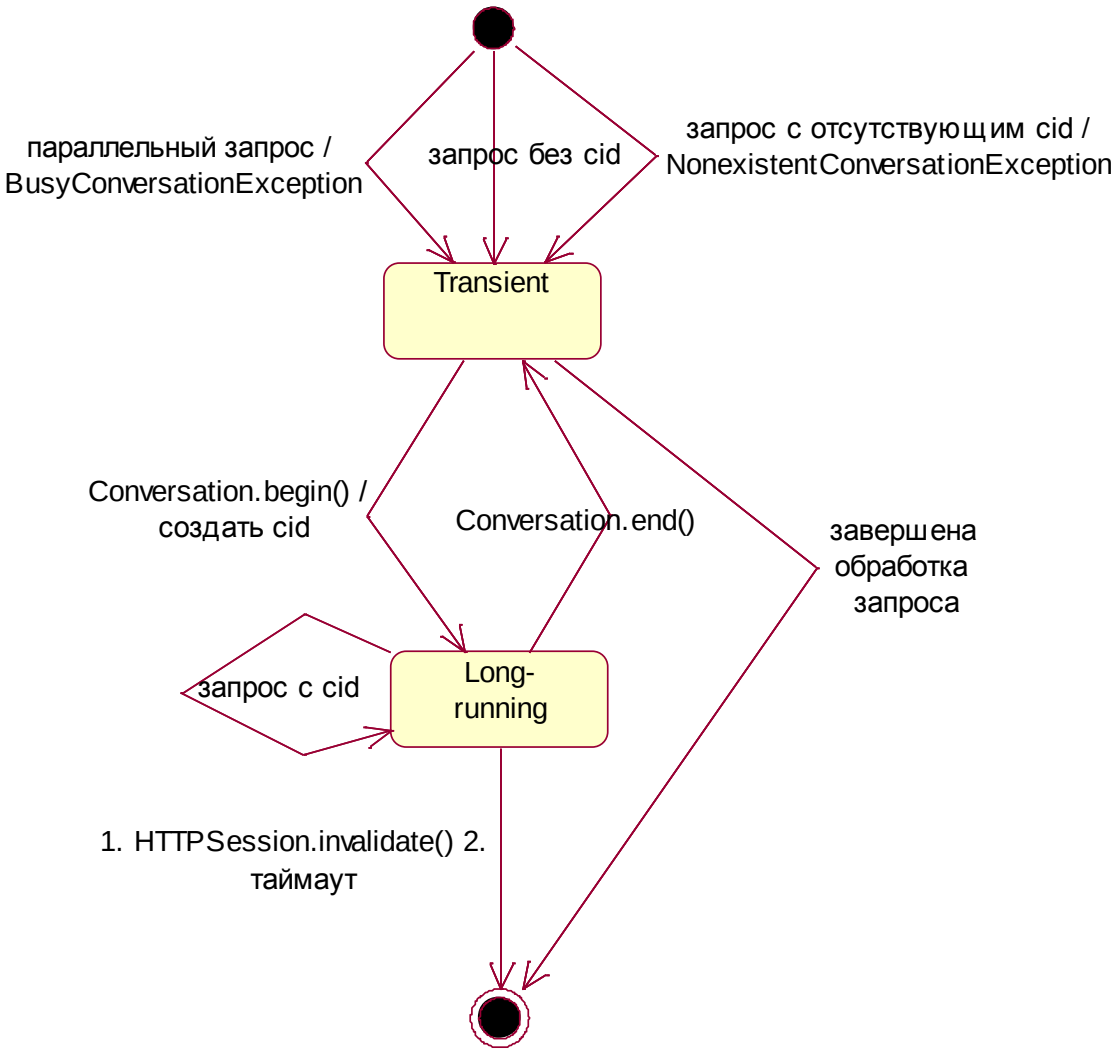
Диалог в долговременном состоянии удаляется контейнером в следующих случаях:

1) при завершении HTTP-сессии, к которой он относится (после обработки метода service() сервлета, то есть можно пользоваться объектами из контекста диалога после вызова метода HTTPSession.invalidate() вплоть до завершения обработки запроса);

2) при необходимости освободить ресурсы (память), а также ориентируясь на таймаут, установленный для диалога.

Если в запросе указывается идентификатор удаленного диалога, то контейнер связывает с таким запросом новый временный диалог и выбрасывает исключение NonexistentConversationException.

Контейнер гарантирует, что диалог в длительном состоянии в любой момент времени связан с обработкой не более одного запроса, а параллельные запросы, относящиеся к этому же диалогу, блокируются или отклоняются. В последнем случае контейнер связывает с отклоняемым запросом



новый временный диалог и выбрасывает исключение BusyConversationException.

Преимущества диалога перед сессией заключаются в следующем: 1) Состояние сессии нужно «очищать» после завершения одной последовательности действий перед началом следующей. 2) С помощью сессии затруднительно реализовать параллельное выполнение одной и той же последовательности действий, но с разными исходными данными.

События

CDI-компоненты могут инициировать и обрабатывать события, что позволяет им взаимодействовать по принципу гибкого связывания, без явного определения зависимостей на этапе компиляции приложения. Более того, события позволяют синхронизировать состояние компонентов, находящихся в различных архитектурных слоях приложения (в веб- и EJB-слоях).

Событие состоит из Java-объекта, передающего данные от источника к обработчикам, и (возможно пустого) множества квалификаторов, более конкретно определяющих тип события и сужающих

число его обработчиков. Таким образом, прежде всего необходим Java-класс события, содержащий поля / свойства для передачи данных.

Для инициирования событий предназначен инъецируемый интерфейс Event с параметром типа события, например:

@Inject Event<GuessEvent> event;

Также для инъецируемого интерфейса Event можно указать квалификаторы, которые будут связаны с инициируемыми событиями. Например, в приложении можно определить два квалификатора: @Success — для события успешного угадывания числа и @Failure — для события превышения числа попыток. Тогда для оповещения об успехах нужно определить следующее поле:

@Inject @Success Event<GuessEvent> event;

Метод Event.fire() отправляет переданный в качестве параметра объект события на обработку (добавляя к нему квалификатору, указанные при инъекции объекта типа Event), например:

event.fire(new GuessEvent());

Метод обработки события — это неабстрактный метод класса управляемого бина (JSF) либо либо класса сессионного компонента (EJB), который обрабатывает только события определенного типа с конкретным множеством квалификаторов. В случае сессионного компонента метод обработки должен быть либо бизнес-методом компонента, либо статическим методом класса компонента. В приложении (и классе компонента) может быть произвольное количество методов обработки событий с одинаковым типом и квалификаторами события.

У метода обработки события должен быть ровно один параметр типа события, помеченный аннотацией @Observes и (необязательно) квалификаторами, во все остальные (необязательные) параметры выполняется инъекция зависимостей при вызове метода. Таким образом, метод обработки события отличается от остальных методов аннотацией @Observes у одного из своих параметров (как правило, у первого).

Пример объявления метода обработки события:

public void processGuess(@Observes @Success GuessEvent event) { … }

Порядок вызова методов обработки события в спецификации не определен, поэтому полагаться на него не следует.

По умолчанию событие обрабатывается непосредственно после его инициирования в источнике события, но есть возможность выполнять обработку позже, при подготовке к фиксации текущей транзакции либо после ее завершения (с любым исходом, с фиксацией, либо с откатом). Это определяется элементом during аннотации @Observes, подобные методы обработки событий называются транзакционными.

Метод обработки события не должен напрямую начинать и завершать (подтверждением либо откатом) JTA-транзакции. При необходимости и только до завершения транзакции метод обработки события может пометить текущую транзакцию к откату с помощью метода setRollbackOnly().

Методы обработки событий могут выбрасывать исключения. Если метод обработки событий является транзакционным, то любое исключение ловится и журналируется контейнером. В противном случае исключение прерывает обработку текущего события, оставшиеся методы

обработки для него не вызываются, а метод Event.fire() повторно выбрасывает исключение, при этом контролируемое исключение «оборачивается» в неконтролируемое ObserverException.

Пример использования

Рассмотрим пример простого веб-приложения, в котором пользователь угадывает загаданное компьютером число от 0 до 9.

Проект приложения имеет следующую структуру:

- index.html — фэйслет, представляющий собой весь интерфейс пользователя приложения;
- GuessNumber — класс управляемого бина (JSF) контекста диалога, который обрабатывает действия пользователя;
- SuccessCounterBean – класс сессионного компонента (EJB) типа синглтон контекста приложения, который реализует счетчик количества угаданных чисел;
- GuessEvent – класс события, оповещающего об угадывании числа;
- Success – квалификатор, обозначающий событие успешного угадывания числа.

1. Рассмотрение примера начнем с файла index.html:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:c="http://java.sun.com/jsp/jstl/core">
  <h:body>
    <f:view>
      <h:form>
        <div align="right">
          <small>Total wins: #{successCounter.totalSuccesses}</small>
        </div>
        <c:choose>
          <c:when test="#{guessNumber.hasGuessed}">
            <h1>You Win!!!!</h1>
            <h:commandLink value="Once again" />
          </c:when>
          <c:otherwise>
            Попытка №#{guessNumber.triesCount}<br/>
            Угадайте число от 0 до 9:
            <h:inputText value="#{guessNumber.userGuess}"/>
            <h:commandButton value="Проверить"
                             action="#{guessNumber.checkGuess()}"/>
          </c:otherwise>
        </c:choose>
      </h:form>
    </f:view>
  </h:body>
</html>
```

В теле фэйслета есть отложенные EL-выражения, содержащие обращения к компонентам с именами guessNumber и successCounter. Контейнер создаст по одному экземпляру этих компонентов при первом обращении к данному фэйслету и разместит их в надлежащих контекстах.

2. В файле GuessNumber.java определен класс управляемого бина GuessNumber:

```
@Named("guessNumber")
@ConversationScoped
```

```
public class GuessNumberBean implements Serializable {
    @Inject Conversation conversation;
    @Inject @Success Event<GameOverEvent> guessEvent;
    private int number;
    private int userGuess;
    private int triesCount;
    private boolean hasGuessed;

    @PostConstruct
    private void postConstruct() {
        conversation.begin();
        number = new Random().nextInt(10);
        triesCount = 1;
    }

    public boolean isHasGuessed() { return hasGuessed; }

    public int getTriesCount() { return triesCount; }

    public int getUserGuess() { return userGuess; }

    public void setUserGuess(int userGuess) { this.userGuess = userGuess; }

    public void checkGuess() {
        if (userGuess == number) {
            hasGuessed = true;
            guessEvent.fire(new GuessEvent());
            conversation.end();
        } else
            triesCount++;
    }

}
```

Экземпляры данного компонента размещаются в контексте диалога (@ConversationScoped) и доступны в EL-выражениях по имени guessNumber (@Named). В полях хранится признак отгаданного числа, само загаданное число, текущая догадка пользователя и счетчик числа попыток.

Длительный диалог создается и завершается из данного компонента, поэтому выполняется инъекция объекта Conversation.

Когда пользователь отгадывает число, компонент инициирует соответствующее событие, для чего выполняется инъекция объекта Event.

Длительный диалог начинается в обработчике события создания экземпляра компонента (метод, помеченный аннотацией @PostConstruct, там же происходит реальная инициализация полей), а завершается в методе проверки догадки пользователя checkGuess().

Обратите внимание, что в случае угадывания числа устанавливается признак hasGuessed, который используется в фэйслете index.html, но при этом завершается диалог, что должно привести к удалению экземпляра компонента. Здесь нет противоречия: фэйслет обрабатывается на последнем этапе обработки запроса, а контекст завершенного диалога (и с ним экземпляр компонента) удаляется после завершения обработки запроса. Это легко отследить, если определить обработчик события удаления экземпляра компонента (метод, помеченный аннотацией @PreDestroy). Таким образом, после угадывания числа фэйслет отображает

поздравление пользователю, а если пользователь желает еще раз угадать число, то он обращается к тому же фэйслету index.html, в котором обрабатывается EL-выражение, приводящее к созданию нового экземпляра компонента guessNumber и началу нового длительного диалога.

3. Файл SuccessCounterBean.java содержит определение EJB-компонента, но размещается в проекте веб-приложения, так как в Java EE 6 разрешено размещать EJB-компоненты в WAR-архиве:

```
@ApplicationScoped
@Named("successCounter")
@Singleton
@LocalBean
public class SuccessCounterBean {
    private int totalSuccesses;

    public int getTotalSuccesses() { return totalSuccesses; }

    @Lock(LockType.WRITE)
    public void addSuccess(@Observes GuessEvent event) {
        totalSuccesses++;
    }
}
```

Класс SuccessCounterBean определяет компонент типа синглтон (@Singleton) с представлением без интерфейса (@LocalBean), экземпляры данного компонента размещаются в контексте приложения (@ApplicationScoped) и доступны в EL-выражениях по имени successCounter (@Named). В поле хранится количество угадываний.

Метод addSuccess() обрабатывает (@Observes) события типа GuessEvent, причем количество угадываний увеличивается независимо от исхода игры, так как для события не указан квалификатор @Success. Для данного метода указана аннотация автоматической синхронизации доступа @Lock, значение которой (LockType.WRITE) говорит о том, что метод изменяет состояние экземпляра компонента и в каждый момент времени его можно вызывать не более чем из одного потока.

4. В файле GuessEvent.java определен класс события:

```
public class GuessEvent implements Serializable {}
```

5. В файле Success.java определен аннотационный тип Success:

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Success {
}
```