УДК 004.434

ФАБРИКИ ПРИКЛАДНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ, УПРАВЛЯЕМЫЕ МОДЕЛЯМИ ПРЕДМЕТНЫХ ОБЛАСТЕЙ

Н. Д. Андреев,
начальник отдела разработки программного обеспечения
ООО «Джи Джи Эй Софтвэр Сервисес», г. Санкт-Петербург
Ф. А. Новиков,
доктор техн. наук, профессор
Санкт-Петербургский государственный политехнический университет

Обсуждаются методы повышения продуктивности разработки прикладного программного обеспечения на основе определения и использования моделей предметных областей и языков предметной области. Предлагаются принципы применения разработки, управляемой моделью предметной области, для создания фабрик прикладного программного обеспечения. Приводится пример разработанного языка предметной области и указываются преимущества, которые дает его использование.

Ключевые слова *— разработка программного обеспечения, фабрики программного обеспечения, управляемая моделью разработка, предметно-ориентированные языки.*

Введение

В последние годы продолжается совершенствование различных подходов и инструментов для повышения эффективности и предсказуемости разработки прикладного программного обеспечения (ППО). В основе этих подходов лежат методы более эффективного повторного использования накопленных знаний и артефактов, а также повышение уровня абстракции, на котором ведется разработка. Обычно это достигается обобщением решаемых задач, их стандартизацией и последующей автоматизацией. Кроме того, оптимизируется и стандартизируется процесс разработки ППО. Таким образом, создаются фабрики по разработке однородного ППО [1].

Для повышения уровня абстракции обычно предлагается использовать проблемно-ориентированные и предметно-ориентированные языки [2]. Проблемно-ориентированные языки нужны для решения конкретной проблемы или ряда сходных проблем в различных предметных областях. Предметно-ориентированные языки имеют специфику конкретной предметной области [3]. Граница между проблемно- и предметно-ориентированными языками достаточно условна, и в большинстве случаев можно использовать обобщающий термин: язык предметной области [4].

Кроме того, для многих проектов разработки ППО разумно повышать уровень абстракции за счет ведения части или всей разработки на уровне модели создаваемого ППО. В этом случае говорят, что разработка управляется моделью. При этом оказывается, что многие модели проще и удобнее конструировать и поддерживать визуально [5]. Общепринятым средством визуального моделирования в настоящее время является унифицированный язык моделирования UML [6].

Этот же метод разработки лежит в основе инициативы MDA (*Model Driven Architecture* — архитектура, управляемая моделью) [7]. Основная идея этого подхода — создание платформенно независимой модели и ее автоматическое преобразование в целевые платформы.

В данной работе предлагаются подходы к созданию фабрик ППО, управляемых моделями. В качестве примера рассматривается фабрика по созданию интеграционных веб-сервисов.

Подходы к построению фабрик ППО, управляемых моделями предметных областей

Центральным понятием в нашем подходе является модель предметной области (МПО). В целом модель предметной области применительно к разработке ППО — это абстрактное представле-

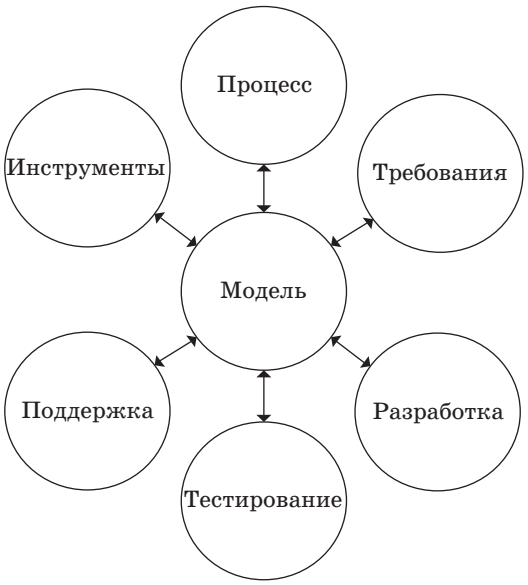
ние знаний, которые определяют множество программно решаемых задач и способов их решения в данной предметной области. В нашем подходе МПО состоит из статического и динамического представлений. Статическое представление является конечным множеством сущностей и множеством отношений между сущностями, что удобно описывается (ориентированным) графом, в котором узлами являются сущности предметной области, а дугами (при необходимости, мультидугами и гипердугами) — отношения между сущностями. Динамическое представление МПО является множеством операций над сущностями и композиций этих операций, т. е. алгоритмов.

Наш многолетний опыт [8] показывает, что моделирование предметной области занимает центральное место в разработке ППО, причем качество МПО существенно влияет на продуктивность разработки. При этом статическое представление модели имеет большее значение, чем динамическое представление.

Моделирование предметной области при разработке ППО является необходимым по существу: невозможно разработать программное обеспечение случайно, не понимая, что и как программное обеспечение должно делать. Но на практике иногда бывает, что моделирование предметной области остается неявной частью процесса разработки, т. е. МПО не разрабатывается в явном отчуждаемом виде, а остается в головах у разработчиков. Это влечет за собой дублирование и одновременно рассогласование моделей в различных артефактах. Например, неявная модель в коде может отличаться от неявно заданной модели в описании требований к ППО, что приводит к несоответствию требованиям и снижению качества ППО.

Основа нашего подхода — явное выделение моделирования предметной области как центральной части разработки ППО. Взаимосвязи МПО с основными составляющими разработки ППО показаны на рис. 1.

1. Требования. Модель предметной области, несомненно, является основой требований к ППО, а конструирование модели — первым шагом в процессе сбора требований и их анализа. Если создается продукт с некоторым шаблонным и стандартным поведением, то, по существу, МПО и является требованиями к ППО. Кроме того, модель является основным связующим звеном между пользователями и командой разработки ППО. Действительно, хорошая модель достаточно абстрактна, не содержит технических деталей и поэтому понятна пользователям. В то же время модель обладает ясной семантикой, а потому является важным источником информации для команды разработки.



■ ***Рис. 1.*** Разработка ППО, управляемая моделью предметной области

2. Разработка. Модель предметной области является основным словарем для разработчиков. Она определяет не только программный код для операций над сущностями и способы хранения сущностей, но может определять, например, структуру и поведение пользовательского интерфейса, интеграцию с другими системами и подсистему верификации данных.

3. Тестирование. Основу тестирования составляют входные и выходные данные системы — разумный перебор входных и верификация выходных данных. МПО определяет свойства данных и, таким образом, во многом определяет набор тестовых примеров.

4. Поддержка. Под поддержкой ППО обычно подразумеваются исправление ошибок после выпуска системы и внесение локальных исправлений внутри системы. Обычно такими исправлениями являются небольшие изменения бизнес-логики или МПО. Например, в системе документооборота может потребоваться дополнительное свойство документа, что повлечет за собой изменение модели.

5. Инструменты. Модель предметной области обычно создают, используя один из инструментов моделирования. Учитывая взаимосвязь модели со всеми шагами разработки ППО, необходимо интегрировать инструмент моделирования с другими используемыми инструментами. Например, в некоторых случаях, исходя из МПО, можно автоматически создавать задачи разработки ППО в системе управления проекта, т. е. интегрировать эту систему с инструментом моделирования. Или, например, автоматически генериро-

вать программный код по модели для последующего редактирования в системе разработки.

6. Процесс. Какой процесс разработки ППО является наиболее эффективным для конкретного проекта, зависит от множества факторов. Тем не менее, по нашему мнению, решающим фактором является стоимость внесения изменений в систему на различных этапах разработки. Применение подхода разработки, управляемой моделью, позволяет минимизировать дублирование и рассогласование знаний, что в свою очередь позволяет минимизировать стоимость внесения изменений в систему.

Важно отметить, что, несмотря на тесную взаимосвязь МПО с различными этапами разработки, в большинстве случаев МПО пока не может полностью определять все артефакты разработки. К сожалению, иногда идеологи разработки, управляемой моделью, настаивают на том, что система всегда должна полностью определяться моделью. Мы считаем, что главной целью должна оставаться эффективность разработки ППО в целом.

Фабрики программного обеспечения позволяют создавать ряд однородных продуктов одинаковым образом и таким образом повышать повторное использование идей, концепций, знаний и артефактов. Процесс создания и использования фабрики ППО, управляемой МПО, представлен на рис. 2.

На первом шаге при создании фабрики программного обеспечения необходимо определить границы обобщения, т. е. тип продуктов, для которого фабрика будет предназначена. По сути это задает круг задач, которые фабрика позволяет решать. При этом можно выбрать два ортогональных направления обобщения:

— проблемный — например, фабрика предназначена для создания сервисов;

— предметный — например, фабрика создается для приложений в области банковского бизнеса.

Далее необходимо определить стандарты и выделить повторяющиеся задачи в рамках фабрики. Все это определяет требования к языку моделирования, проектируемому и реализуемому на следующем шаге. Если мы создаем фабрику проблемного уровня, то создаваемый язык будет тоже проблемно-ориентированным. Если мы создаем фабрику предметного уровня, то создаваемый язык будет тоже предметно-ориентированный.

Если не повышать уровень абстракции за счет создания языка и при этом следовать принятым стандартам и выполнять повторяющиеся задачи вручную, то в рамках фабрики будет расти дублирование знаний и кода. С помощью созданного языка можно определять МПО продуктов с семантикой, которая позволит минимизировать дублирование в рамках фабрики, т. е. создавать часть артефактов разработки автоматически.

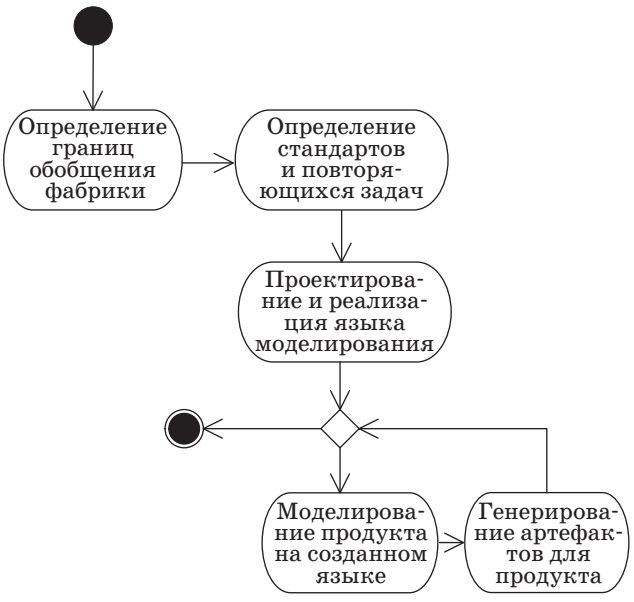
Важно построить эффективный процесс по созданию и развитию фабрик ППО. Мы считаем, что тут необходим инкрементальный процесс [3]. В этом случае создаются одна или несколько фабрик, поддерживающих разработку на проблемных или предметных уровнях. Если при этом использовать подход разработки ППО, управляемой моделью, то следует создать проблемно- или предметно-ориентированные языки моделирования.

Мы предлагаем следующие принципы совместного использования фабрики ППО и разработки, управляемой МПО.

• Явное выделение моделирования предметной области как центральной части разработки ППО. При построении МПО различаются статическое и динамическое представления, причем статическое представление описывается графом, задающим отношения между сущностями. В большинстве случаев граф МПО разумно представлять и изменять визуально.

• Создание проблемно-ориентированного языка моделирования предметной области. Синтаксис и метамодель языка моделирования основываются на языке UML [9]. Созданный язык определяет стандарты и уровень обобщения в рамках фабрики. Семантика созданного языка позволяет избежать дублирования концепций и кода через создание генераторов.

• Модель продукта на созданном проблемно-ориентированном языке предопределяет характеристические свойства продукта, но не довлеет над деталями реализации. Часть артефактов может быть создана традиционными способами.



■ ***Рис. 2.*** Создание и использование фабрики ППО, управляемой МПО

Возможно продолжение разработки на традиционном языке программирования в любой момент.

• Создание языка и последующее его использование осуществляются в специализированном инструменте, который позволяет производить метамоделирование, задавать свойства графического синтаксиса языка и правила, накладываемые на метамодель. Правила задаются на языке OCL [10]. В процессе моделирования на созданном языке инструмент верифицирует модели на соответствие заданным правилам.

Проблемно-ориентированный язык SOALang для описания сервисов

Особенностью программных систем уровня предприятия является долгое время жизни и интеграция систем между собой. Любое изменение программных интерфейсов одной системы необходимо проводить вместе с изменениями зависимых систем. При большом количестве систем и связей между ними подобные изменения являются рискованными и дорогостоящими. Сервис-ориентированная архитектура (*Service-Oriented Architecture* — SOA) [11] призвана решить эти проблемы посредством интеграции систем через слабо связанные между собой сервисы. Такие сервисы не имеют прямых вызовов друг друга, и системы интегрируются не напрямую, а через сервисную шину предприятия [12]. При этом программные интерфейсы и МПО, выставляемые потребителю интеграционных сервисов, должны быть каноническими, т. е. не зависеть от используемых систем и деталей реализации сервисов.

Таким образом, при изменении или замене используемых систем интеграционные сервисы скрывают эти изменения от других систем. Подобные сервисы должны создаваться в соответствии с разработанными и принятыми стандартами, чтобы разработчикам зависимых систем было привычно и удобно пользоваться их интерфейсами и чтобы снизить стоимость поддержки как самих сервисов, так и использующих их систем. Особенно важно следовать этим подходам в крупных компаниях, где количество интегрируемых между собой систем может исчисляться тысячами. При этом разработку интеграционных сервисов разумно обобщить и выполнить в рамках фабрики. Это даст возможность автоматизировать повторяющиеся действия и построить эффективный процесс разработки. Все принятые стандарты необходимо автоматически верифицировать, если соответствующие действия не могут быть автоматизированы и выполняются вручную. Одним из самых важных этапов при разработке интеграционных сервисов является разработка канонической МПО [13].

Для решения вышеописанных задач в компании «Джи Джи Эй» [14] был разработан проблемно-ориентированный язык SOALang, который позволяет задавать модель сущностей, их свойства и связи. Таким образом, язык описывает явно только статическую составляющую интеграционных сервисов.

Язык SOALang основан на подмножестве графического синтаксиса, метамодели и семантики диаграммы классов языка UML. Упрощенная метамодель языка SOALang представлена на рис. 3. При этом введены дополнительные правила и более строгие ограничения, в частности:

1) имя сущности должно быть уникальным существительным в единственном числе;

2) именование сущности должно быть в соответствии со стандартом UpperCamelCase [15];

3) поле сущности должно иметь уникальное имя в рамках сущности и разрешимый тип;

4) именование полей сущности должно быть в соответствии со стандартом lowerCamelCase [15];

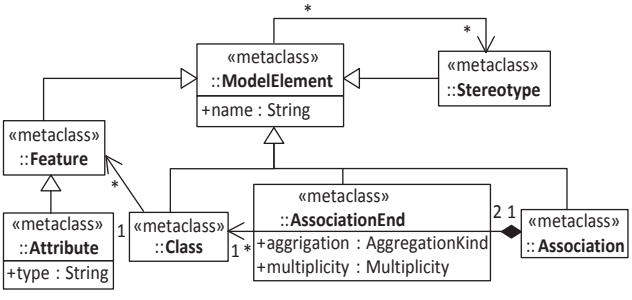
5) имя роли ассоциации должно быть задано, если роль не является агрегацией, композицией или направленной; должно быть во множественном или в единственном числе — в зависимости от свойства множественности роли ассоциации;

6) для каждой сущности должен быть задан один уникальный неизменяемый ключ — задается как стереотип «*unique immutable*» для одного из полей;

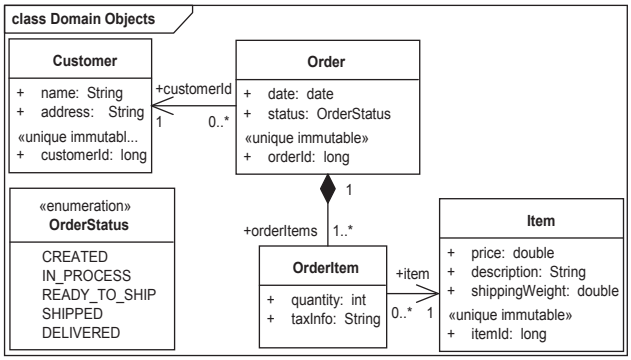
7) у сущности типа Перечисление (стереотип «*enumeration*») все значения должны состоять из прописных букв.

Многие из дополнительных правил задают стандарты именования сущностей и полей. Это необходимо, чтобы выводимые из моделей артефакты были согласованы и похожи между собой в рамках фабрики. Принятые единые стандарты и согласованность интерфейсов в рамках фабрики упрощают использование сервисов в целом.

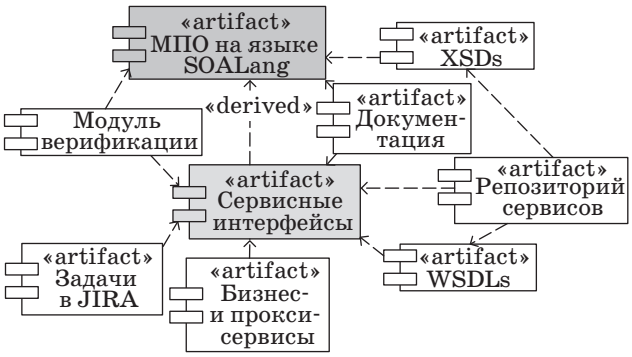
Пример модели, созданной с помощью языка SOALang, показан на рис. 4. В этой модели описывается предметная область ведения базы заказов. В данной МПО выделяются сущности: заказ (*Order*), заказчик (*Customer*), позиция заказа



■ ***Рис. 3***. Упрощенная метамодель языка SOALang



■ ***Рис. 4.*** Пример МПО на языке SOALang



■ ***Рис. 5.*** Язык SOALang и производные артефакты

(*OrderItem*) и товар (*Item*). Сущности имеют атрибуты и связаны отношениями, семантика которых определяется семантикой использованных отношений UML.

В качестве инструмента моделирования используется Enterprise Architect (EA) [16]. Для него можно разрабатывать расширения, в коде которых можно осуществлять доступ к открытым в инструменте моделям — создавать новые или изменять существующие модели. Были разработаны нижеследующие расширения для поддержки разработанного языка SOALang и верификации модели на соответствие правилам, определенным в языке. На рис. 5 показана диаграмма связи модели, артефактов и модуля верификации. Первичным артефактом является МПО на языке SOALang, разработанная бизнес-аналитиком. В следующем разделе описываются выводимые из МПО артефакты и модуль верификации.

Преимущества использования языка SOALang

Под сервисными интерфейсами мы понимаем набор операций над объектами предметной области, предоставляемых сервисом. При этом операции определенным образом группируются в интерфейсы.

В рамках фабрики сервисов определены стандарты и правила, которым необходимо следовать при разработке сервисных интерфейсов. В частности, для каждой сущности создается отдельный сервисный интерфейс. Например, для сущности с именем Entity создается интерфейс с именем EntityService. Исключение составляют:

— перечисления;

— сущности, которые связаны с другими сущностями ассоциацией композиции, т. е. являются частью другой сущности, например OrderItem (см. рис. 4).

При этом в каждом интерфейсе присутствует стандартный набор методов со следующими правилами именования для сущности с именем Entity:

— получение сущности по ключу (атрибут со стереотипом «*unique immutable*») — findEntityByEntityId;

— получение сущностей по списку ключей — findEntitiesByEntityIds;

— получение всех сущностей — findAllEntities.

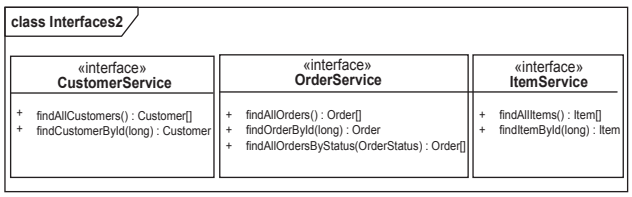
Например, для сущности Order:

— findOrderByOrderId(long orderId) : Order
— findOrdersByOrderIds(long[] orderIds) : Order[]
— findAllOrders() : Order[]

Мы определили эти правила как часть семантики языка SOALang, и это позволило реализовать генератор сервисных интерфейсов по МПО. Таким образом, сервисные интерфейсы и их операции по умолчанию генерируются автоматически как еще одна модель. Перед запуском процесса можно выбрать опцию генерации методов, которые изменяют данные: добавление, изменение и удаление сущности. По умолчанию генерируются только методы чтения, так как обычно интеграционные сервисы только возвращают информацию из различных источников данных, но не записывают ее.

Получившиеся после генерации интерфейсы можно отредактировать вручную. Если будут сделаны изменения модели и опять запущена генерация сервисов, то будут добавлены новые или изменены существующие методы. Добавленные пользователем методы изменены не будут.

Пример сгенерированной модели сервисных интерфейсов из МПО представлен на рис. 6. Стоит отметить, что для сущности OrderItem интер-



■ ***Рис. 6.*** Пример модели сервисных интерфейсов

фейс сгенерирован не был в соответствии с вышеописанными правилами — действительно, сущности Order и OrderItem связаны отношением композиции.

Верификация модели — это проверка ее соответствия требованиям и правилам, определенным для языка. Верификация позволяет определить проблемы моделирования на ранних стадиях. Был реализован модуль верификации МПО и сервисных интерфейсов на соответствие правилам, определенным в языке SOALang. Верификация может быть запущена вручную при редактировании модели и автоматически запускается при генерации артефактов из модели. Мы разделили потенциальные проблемы на две группы:

— ошибка (error) — при таких проблемах дальнейшая генерация невозможна. Например, если имя роли ассоциации не задано, то это ошибка;

— предупреждение (warning) — при таких проблемах будет выдано предупреждение, но работа может быть продолжена. Например, если для какой-то сущности, которая связана отношением композиции с другой сущностью (является частью другой сущности), не будет задан ключ, то будет выдано предупреждение.

В процессе верификации сервисных интерфейсов проверяются правила именования и существование типов, на которые есть ссылки из сигнатур операций.

В используемом инструменте Enterprise Architect верификация правил стандарта UML реализована не полностью. Разработанный нами модуль верификации покрывает пропущенные, но являющиеся частью языка SOALang правила UML, и дополнительные правила языка SOALang. Для верификации модели правила группируют по тем элементам метамодели, к которым правила применимы. Кроме того, для эффективного процесса верификации формируются логические цепочки зависимостей правил [17]. Разработанный модуль верификации делает обход модели и применяет соответствующие правила к каждому элементу.

Файлы стандартов WSDL [18] и XSD [19] описывают интерфейс сервисов для потенциальных потребителей. Для обеспечения совместимости и согласованности были разработаны обязательные стандарты и правила создания WSDL и XSD. Встроенные возможности инструмента Enterprise Architect по генерации этих артефактов не отвечают нашим стандартам. Поэтому мы разработали собственный генератор WSDL и XSD. Например, тип агрегирования в отношении между сущностями влияет на XSD. Если отношение является композицией, то для зависимой сущности будет сгенерирован вложенный элемент. Если отношение является агрегацией, то будет сгенерирована ссылка на зависимую сущность. Ниже представлен фрагмент XSD, сгенерированного по модели:

```
<xs:complexType name="Order">
 <xs:sequence>
  <xs:element name="date" type="xs:date"
              minOccurs="0" maxOccurs="1"/>
  <xs:element name="status" type="tns:OrderStatus"
              minOccurs="0" maxOccurs="1"/>
  <xs:element name="orderId" type="xs:long"
              minOccurs="0" maxOccurs="1"/>
  <xs:element name="orderItems" type="tns:OrderItem"
              minOccurs="1" maxOccurs="unbounded"/>
  <xs:element name="customerId" type="xs:long"
              minOccurs="1" maxOccurs="1"/>
 </xs:sequence>
</xs:complexType>
```

Файлы XSD создаются по МПО, а по сервисным интерфейсам генерируются файлы WSDL. При этом в WSDL добавляются требуемые стандартом параметры безопасности и пространства имен. Для создания и отслеживания статуса задач и ошибок используется система JIRA [20]. Для разрабатываемых сервисов готовятся и проводятся автоматические функциональные и нагрузочные тесты. Таким образом, для каждого метода сервиса существуют следующие стандартные задачи:

1) разработка метода;

2) подготовка функциональных тестов с помощью SOAP UI [21];

3) подготовка тестов производительности/нагрузки с помощью JMeter [22].

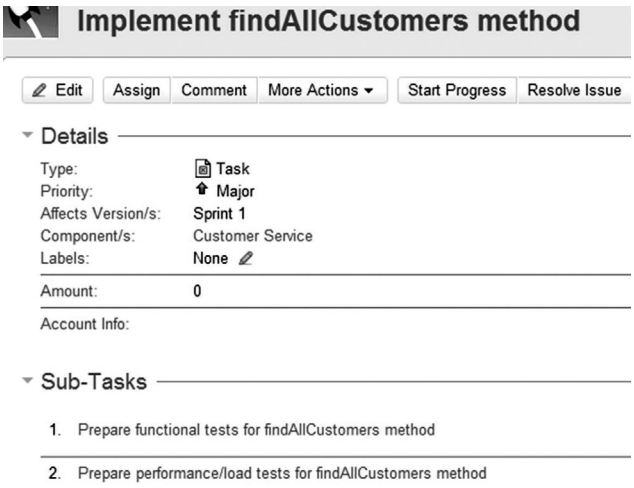
Система JIRA поддерживает удобный программный интерфейс, который, в частности, позволяет добавлять новые задачи. Была реализована подсистема автоматического создания вышеописанных задач в JIRA по сервисной модели. В качестве компоненты указывается соответствующий сервис, а задачи типа 2, 3 (подготовка автоматических тестов) создаются как подзадачи соответствующей задачи на разработку метода сервиса. На рис. 7 показан пример автоматически полученной задачи в JIRA.

Обычно при разработке интеграционных сервисов реализуют следующие дополнительные сервисные прослойки на сервисной шине предприятия:

— бизнес-сервисы — для балансировки нагрузки, кэширования результатов, обеспечения гарантированной доставки;

— прокси-сервисы — для маршрутизации запросов и ответов, трансформации данных, управления транзакциями.

При полноценном использовании сервис-ориентированной архитектуры создание бизнес- и прокси-сервисов является обязательным. При этом на этих уровнях подключаются некоторые стандартные механизмы, например безопасность. Часто получается, что эти прослойки не выпол-



■ *Рис. 7.* Пример автоматически созданной задачи в JIRA

няют никакой дополнительной функции и реализуются стандартным образом. Был реализован генератор XML-файлов, описывающих конфигурацию бизнес- и прокси-сервисов из МПО и сервисных интерфейсов. Ниже приводится фрагмент конфигурации XML для бизнес-сервиса:

```
<ser:binding type="SOAP" isSoap12="false"
  xsi:type="con:SoapBindingType"
  xmlns:con="http://www.bea.com/wli/sb/services/bindings/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <con:wsdl ref="OrderingServices/Resources/
    CustomerInteraction_1_0/CustomerInteraction"/>
  <con:port>
    <con:name>CustomerResponderPort</con:name>
    <con:namespace>
      urn:namespace:services:Customer:1.0
    </con:namespace>
  </con:port>
</ser:binding>
```

Таким образом, мы уменьшаем дублирование и гарантируем использование принятых стандартов.

Частью процесса разработки интеграционных сервисов является подготовка документации. Был реализован генератор документов по МПО и сервисных интерфейсов. Получаемая документация не является полной, и ее необходимо дополнять вручную. Тем не менее, генератор существенно упрощает создание документации и помогает следовать принятым шаблонам.

Для эффективного использования интеграционных сервисов в крупных компаниях необходим репозиторий сервисов, где собирается информация о свойствах, версиях и возможностях сервисов, взаимосвязях между собой и зависимостях от внешних систем. Была реализована интеграция с репозиторием SOA Lifecycle Manager [23] для автоматического добавления туда информации о сервисах по модели, включая описания WSDL.

Заключение

В работе рассмотрен пример фабрики по созданию интеграционных веб-сервисов, управляемой МПО. На примере показано, что если в рамках фабрики определить стандарты и правила для получаемых артефактов, то многие процессы можно оптимизировать. При этом центральным элементом становится МПО. На основе обобщения этого и других примеров предложены принципы построения фабрик ППО, управляемых моделями.

Дальнейшее развитие работы мы видим в определении подходов к декларативному описанию операционной семантики. Например, указывать дополнительные свойства на МПО, которые позволят генерировать часть программного кода. При этом важно сохранить платформенно-независимую модель. Для рассмотренного примера интеграционных сервисов это даст возможность генерировать часть кода сервисов, например на языке BPEL или Java.

Литература

1. **Greenfield J., Short K., Cook S., Kent S.** Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. — Wiley, 2004. — 500 p.
2. **Dmitriev S.** Language Oriented Programming: The Next Programming Paradigm. http://www.onboard.jetbrains.com/articles/04/10/lop/mps.pdf (дата обращения: 08.03.2012).
3. **Андреев Н. Д., Новиков Ф. А.** Инкрементальный предметно-ориентированный процесс разработки прикладного программного обеспечения // Информационно-управляющие системы. 2012. № 1. С. 60–66.
4. **Новиков Ф. А., Тихонова У. Н.** Автоматный метод определения проблемно-ориентированных языков // Информационно-управляющие системы. 2009. № 6. С. 34–40; 2010. № 2. С. 31–37; № 3. С. 29–37.
5. **Новиков Ф. А.** Визуальное конструирование программ // Информационно-управляющие системы. 2005. № 6. С. 9–22.
6. **Новиков Ф. А., Иванов Д. Ю.** Моделирование на UML. Теория, практика, видеокурс. — СПб.: Наука и Техника, 2010. — 640 с.

7. **Booch G.** et al. An MDA Manifesto // The Mda J.: Model Driven Architecture Straight From The Masters. Meghan Kiffer Press, Dec. 2004. Chap. 11. P. 2–9.
8. **Новиков Ф. А.** Методы алгоритмизации предметных областей: дис. ... д-ра техн. наук. СПб.: СПбГУ ИТМО, 2011. http://www.dissercat.com/content/metody-algoritmizatsii-predmetnykh-oblastei (дата обращения: 08.03.2012).
9. **Андреев Н. Д.** Предметно-ориентированный язык моделирования, основанный на UML // Формирование технической политики инновационных наукоемких технологий: материалы конф. и школы-семинара / СПбГПУ. СПб., 2004. С. 75–82.
10. **Язык** OCL. http://en.wikipedia.org/wiki/Object_Constraint_Language (дата обращения: 08.03.2012).
11. **Service-Oriented** Architecture. http://en.wikipedia.org/wiki/Service-oriented_architecture (дата обращения: 08.03.2012).
12. **Enterprise** Service Bus. http://en.wikipedia.org/wiki/Enterprise_service_bus (дата обращения: 08.03.2012).
13. **Hohpe G., Woolf B.** Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. — Addison-Wesley, 2003. — 736 p.
14. **http:**//www.ggasoftware.com/ (дата обращения: 08.03.2012).
15. **Camel** case. http://ru.wikipedia.org/wiki/CamelCase (дата обращения: 08.03.2012).
16. **Enterprise** Architect. http://www.sparxsystems.com/ (дата обращения: 08.03.2012).
17. **Андреев Н. Д.** Автоматическая верификация модели UML // IV Междунар. молодежная школа-семинар «БИКАМП-2003»: тр. конф. / ГУАП. СПб., 2003. С. 145–149.
18. **WDL.** http://www.w3.org/TR/wsdl (дата обращения: 08.03.2012).
19. **XML** Schema. http://www.w3.org/XML/Schema.html (дата обращения: 08.03.2012).
20. **Инструмент** JIRA. http://www.atlassian.com/software/jira/overview (дата обращения: 08.03.2012).
21. **Инструмент** SOAP UI. http://www.soapui.org (дата обращения: 08.03.2012).
22. **Инструмент** JMeter. http://jmeter.apache.org/ (дата обращения: 08.03.2012).
23. **Продукт** Lifecycle Manager. http://www.soa.com/products/lifecycle_manager (дата обращения: 08.03.2012).

ПАМЯТКА ДЛЯ АВТОРОВ

Поступающие в редакцию статьи проходят обязательное рецензирование.

При наличии положительной рецензии статья рассматривается редакционной коллегией. Принятая в печать статья направляется автору для согласования редакторских правок. После согласования автор представляет в редакцию окончательный вариант текста статьи.

Процедуры согласования текста статьи могут осуществляться как непосредственно в редакции, так и по e-mail (80x@mail.ru).

При отклонении статьи редакция представляет автору мотивированное заключение и рецензию, при необходимости доработать статью — рецензию. Рукописи не возвращаются.

Редакция журнала напоминает, что ответственность за достоверность и точность рекламных материалов несут рекламодатели.