электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.4'22

ПРАКТИЧЕСКИЕ АСПЕКТЫ ЦЕЛЕОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

***Богачёв А.Ю.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

***Данелян Е.К.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

***Солопов А.И.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»
art-solopov@yandex.ru

Введение

В настоящее время существует множество методологий разработки ПО. Некоторые из них (такие, как Model-Driven Architecture или Rational Unified Process) разделяют процесс непосредственно проектирования и кодирования. В подобных методологиях на этапе проектирования вопросы специфики кодирования не обсуждаются вообще.

Другие методологии (например, методологии класса Agile) интегрируют процесс итеративного проектирования и кодирования: заказчик устно объясняет, чего он хочет, а команда специалистов сразу переводит его требования в код.

Первые методологии часто страдают от недостатков перепроектирования, когда проектирование ПО становится самоцелью. В результате вместо простой программы, которая должна считать сумму двух чисел, команда может спроектировать нагромождение Стратегий Декораторов, Команд, Коллекций, Фабрик, Фабрик для создания этих Фабрик и Фабрику-Одиночку для создания Фабрик, создающих Фабрики. Этот монстр сможет считать любую функцию от любого количества чисел, но пользователь никогда не сможет разобраться в настройках, требуемых для подсчёта суммы двух чисел.

Методологии второго типа часто критикуют за недостаток проектирования, подход к программированию в стиле «абы как, но чтобы работало» и отсутствие управления требованиями. Результатом подобного подхода может стать наполненная «костылями» программа, которая будет очень негибкой и сложной в поддержке.

Встаёт вопрос: как совместить управление требованиями классических методологий, гибкость и скорость Agile, при этом получить программу, простую в

поддержке? И, возможно, улучшить существующие концепции, возложив часть труда рутинного программирования на машину?

Для начала обратимся к процессу проектирования ПО и управления требованиями. Авторы полагают, что проектирование программы неотделимо от целеполагания и должно опираться именно на него. То есть, за основной принцип разработки возьмём целеориентированное проектирование.

Целеориентированное проектирование (Goal-driven development) — это методология проектирования ПО, при котором основной упор изначально ставится на цели разработки ПО и требования к ПО. Обычно при этом программа проектируется по методу схождения подходов сверху вниз и снизу вверх.

Подход сверху вниз заключается в выделении одной главной цели и разбиении её на подцели. Каждой цели и подцели ставится в соответствие несколько вопросов. Вопросы определяют тестируемые моменты в системе. Цель считается достигнутой, когда на все её вопросы получен положительный ответ и достигнуты все её подцели.

Подход снизу вверх отвечает на три основных вопроса: осуществима ли цель в принципе, как её осуществить и какие потребуются для этого вложения.

Для адекватного проектирования заказчики и разработчики должны плотно сотрудничать в процессе определения цели, поддерживая заказчиков, которые идут сверху вниз. Причина такой необходимости в том, что пользователи зачастую не знают реальных цен разработки ПО: так внедрение ещё одного экрана в программу может обойтись довольно дорого при том, что разработка сложных статистических расчётов, необходимых для бизнес-логики, могут обойтись почти бесплатно, если воспользоваться подходящими библиотеками в СУБД или сервере приложений.

Результатом разработки становится дерево целей, идущее от верхней, неспециализированной, общей, неформальной цели к более конкретным и лучше формализуемым целям нижнего уровня. Листовым целям (то есть, целям, которые далее не декомпозируются) присваиваются артефакты: документы, спецификации, файлы с исходным кодом и т. д. [1].

Постановка задачи

Итак, в итоге мы имеем замечательную методологию разработки, разумную (поскольку она идёт из целей, из того, ради чего, собственно, и создаётся ПО) и гибкую (поскольку артефакты чётко не заданы и каждую отдельную подцель можно разрабатывать отдельно по разным методологиям). Остаётся вопрос: как применить её на практике?

Прежде всего, вполне можно применять её просто как верхний уровень разработки ПО: для целеполагания и управления целями. После выделения всех подцелей можно

использовать другую методологию для проектирования, после чего приступить к кодированию. Однако у подобного подхода есть несколько существенных недостатков:

1. Сложность. Целеориентированная методология сама по себе непроста, её применение требует квалифицированных разработчиков. Введение дополнительного слоя проектирования только усложняет систему.

2. Отделение кода от целей. Промежуточные слои проектирования могут пасть жертвой перепроектирования, в результате чего программист может не знать, какой именно цели служит его код.

Авторы считают, что лучшее практическое применение методологии может быть достигнуто путём её интегрирования в процесс проектирования и кодирования, то есть, создания единой методологии (и CASE-системы на её основе), опирающейся на цели.

Однако для проектирования, напрямую опирающегося на цели (не использующего другие методологии для разработки конкретной цели) необходимо решить ещё одну проблему. Цели задают только поведение системы, упуская из виду структуру информационных сущностей, которыми оперирует система. Для создания и упорядочения информационных сущностей можно использовать онтологии.

Онтология — это вычислительный артефакт, предназначенный для формализации структуры системы, её сущностей и связей, релевантных и полезных нам. Онтология строится на принципах таксономии: введения отношений специализации и обобщения [2].

Решение задачи

Для формулировки концепции CASE-системы будем исходить из следующих основных постулатов:

1. Изначальная цель (корень дерева целей) общая, расплывчатая и неконкретная. То есть, попытка сделать универсальное CASE-средство, которое бы строило дерево целей и генерировало код в полностью автоматическом режиме, если не обречена на провал, то по меньшей мере неосуществима в сколько-нибудь разумные сроки.

2. Листовые цели, напротив, вполне конкретны и часто не просто хорошо формализуемы, но вообще шаблонны. Примеры целей нижнего уровня — сохранить некую сущность в базе данных, получить из базы данных сущности, удовлетворяющие некоторому условию, записать некую информацию в журнал, отобразить результаты действий пользователю и т. д. Код для подобных шаблонных целей также будет шаблонным.

3. Написание шаблонного кода скучно, а значит, такой код будет содержать ошибки по невнимательности.

4. Не все цели можно целиком разложить на шаблоны. В системе могут остаться листовые цели, не соответствующие шаблонам и не декомпозируемые в дальнейшем.

5. Оставшаяся нешаблонной подцель будет, как и исходная цель, плохо формализуемой. Значит, для её алгоритмизации нужно задействовать программиста. Поскольку код нешаблонный, вероятность ошибок по невнимательности снижается.

6. Код должен быть неотделим от цели. Любой оператор языка должен быть написан с какой-то целью, которая должна «висеть» над программистом подобно Дамоклову мечу, дабы тот ни на секунду не мог забыть, для чего он пишет код.

7. Цели меняются.

8. Цели редко меняются глобально. Скорее всего, из всего дерева целей поменяется относительно небольшое поддерево.

Распишем принцип действия системы подробнее.

В отличие от классических IDE, в которых разработчику предлагается сразу начать писать код, данное CASE-средство должно предложить разработчикам удобный визуальный редактор построения целей, подцелей и их связей. Например, это может быть «поярусный» редактор, когда в каждый конкретный момент на экране разработчики видят подмножество яруса дерева целей (ярус — множество вершин дерева с одинаковым уровнем [3]). В этом редакторе можно будет:

1. Вводить шаблонные и нешаблонные цели и задавать их свойства.

2. Декомпозировать нешаблонные цели или объявлять их листовыми (то есть, которые должны быть непосредственно закодированы).

3. Присваивать нешаблонным целям вопросы.

4. Задавать связи между целями.

5. Переходить по дереву целей вверх и вниз.

Параллельно с процессом задания целей разработчики создают онтологию сущностей системы в виде некоторой концептуальной схемы.

После задания целей начинается самая интересная часть. На основе шаблонных целей и онтологии система генерирует агенты, которые эти шаблонные цели будет выполнять. Для нешаблонных целей генерируются «скелеты» агентов, на основе которых будут разрабатывать программисты. На основе вопросов генерируются тестовые сценарии для нешаблонных целей.

Как мы помним, цели в системе будут изменяться, хотя и не слишком глобально. Что же произойдёт в случае изменения дерева целей?

1. Система вычислит разницу между деревьями (то есть, найдёт поддеревья,

которые требуется удалить и поддеревья, которые требуется ввести);

2. Система удалит соответствующие поддеревья и артефакты, не обеспеченные целями;

3. Система введёт новые поддеревья;

4. Для шаблонных целей поддеревьев система сгенерирует шаблонных агентов и привяжет их к системе;

5. Если в системе остались нешаблонные цели, система даст сигнал программистам и сгенерирует для них скелеты агентов.

6. После реализации всех агентов система заново активирует проект.

Выводы

Преимущества CASE-системы состоят в том, что:

1. Больше внимания и труда переносится с процесса кодирования на процесс проектирования. Как известно, именно ошибки в проектировании обходятся дороже всего.

2. Уменьшается стоимость разработки за счёт автоматической генерации шаблонного кода.

3. Упрощается отладка за счёт внедрения вопросов, на которые отвечает система в процессе тестирования.

4. Улучшается гибкость за счёт возможности перестройки дерева целей и простой замены одних шаблонных агентов на другие.

5. Увеличивается скорость разработки.

6. Агенты могут писаться на самых разных языках программирования с применением самых разных парадигм.

Недостатки системы:

1. Разработка проекта исходя из целей — удовольствие долгое и дорогое. Само по себе проектирование целей требует высококлассного специалиста, который сможет помочь пользователю разложить цель на подцели и определить возможность реализации цели.

2. Взаимодействие агентов увеличит требования системы к ресурсам.

3. Трудоёмкость создания инфраструктуры самой CASE-системы. Создание хорошего кода для шаблонных целей на многих языках программирования — неподъёмная задача для одной компании.

Вопросы, на которые ещё требуется найти ответы:

1. Вопросы реализации системы в целом. Что именно должна генерировать система? Модули одной программы? Мультиагентную систему как набор автономных, взаимодействующих между собой программ?

2. Вопросы реализации интерфейса: как должны выглядеть экраны проектирования целей и онтологии?

3. Вопрос проектирования тестов: автоматическое или ручное?

4. Вопрос открытости кода: closed source или open source?

5. Вопрос изменения онтологии: как должна меняться система при изменении онтологии системы?

6. Вопросы кодогенерации: осуществлять кодогенерацию используя существующие языки программирования, или создавать специальный язык для этих целей?

Список литературы

1. Ingo Schnabel, Markus Pizka «Goal-Driven Software Development» URL http://www.itestra.de/uploads/media/06_itestra_goal_driven_sw_development.pdf (дата обращения: 25.09.2013г.).
2. Oberle, D., Guarino, N., & Staab, S. (2009) What is an ontology? . In: "Handbook on Ontologies". Springer, 2nd edition, 2009.