Математические структуры и моделирование
2014. № 4(32). С. 180–184

УДК 004.052

ПРИМЕНЕНИЕ ЭКСТРЕМАЛЬНОГО ПРОГРАММИРОВАНИЯ ПРИ РАЗРАБОТКЕ НАУЧНЫХ ПРИЛОЖЕНИЙ

В.С. Веденеев
ведущий спец-т по информационной безопасности (ОАО «ЧЦЗ»),
e-mail: ingafen@gmail.com
И.В. Бычков
д.ф.-м.н., профессор, e-mail: bychkov@csu.ru

Челябинский государственный университет

Аннотация. В работе рассмотрены особенности применения методологии экстремального программирования при разработке научного программного обеспечения.

Ключевые слова: экстремальное программирование, рефакторинг.

1. Экстремальное программирование

Экстремальное программирование — методология разработки программного обеспечения (далее — ПО), придуманная группой разработчиков (Кент Бек, Уорд Каннингем, Мартин Фаулер и др.). Применение этой методологии даёт возможность быстро разрабатывать бизнес-приложения, а также даёт большую гибкость при разработке в случае изменения условий заказчика. Экстремальное программирование ориентировано на работу в команде. При этом существует возможность разработки с применением этой методологии одним разработчиком.

Применение этой методологии даёт большое преимущество при разработке бизнес-приложений. Первое — применение разработки через тестирование (англ. Test Driven Development, далее — TDD) даёт не только некоторый уровень гарантии в качестве программного продукта (т.е. соответствия изначальным требованиям заказчика и полного соответствия замыслу разработчиков), но и даёт уверенность разработчиков в том, что новый функционал не будет вносить коррективы в базис разрабатываемой системы. Кроме того, при выявлении ошибок в разработке или неучтённых ситуаций TDD даёт уверенность в том, что ошибка не повторится вновь. Впоследствии, при длительной разработке больших систем применение TDD даёт большое преимущество в скорости разработки перед традиционными методологиями.

Кроме TDD в экстремальное программирование содержит ряд других практик [1]:

- Короткий цикл обратной связи (Fine scale feedback)

Игра в планирование (Planning game)

Заказчик всегда рядом (Whole team, Onsite customer)

Парное программирование (Pair programming)

- Непрерывный, а не пакетный процесс

  Непрерывная интеграция (Continuous Integration)

  Рефакторинг (Design Improvement, Refactor)

  Частые небольшие релизы (Small Releases)

- Понимание, разделяемое всеми

  Простота (Simple design)

  Метафора системы (System metaphor)

  Коллективное владение кодом (Collective code ownership) или выбранными шаблонами проектирования (Collective patterns ownership)

  Стандарт кодирования (Coding standard or Coding conventions)

- Социальная защищенность программиста (Programmer welfare)

  40-часовая рабочая неделя (Sustainable pace, Forty hour week)

При разработке командой из одного человека многие практики сливаются во едино. Решение о применении или неприменении той или иной практики остаётся на усмотрение разработчика.

При разработке одним человеком этот человек становится и разработчиком, и руководителем проекта. В случае разработки научных приложений он является ещё и заказчиком. Поэтому многие практики могут «мутировать» и изменяться.

По мнению автора, наиболее удобным набором практик в таком случае являются:

1. TDD и рефакторинг
2. Непрерывная интеграция
3. Простота
4. Метафора системы

Расшифруем каждый из терминов.

Рефакторинг — оптимизация кода программы с целью достижения простоты и читаемости кода и др. без изменения поведения программы [2]. Признаками кода, которые свидетельствуют о необходимости рефакторинга (т.н. «код с запашком»), являются следующие явления:

- дублирование кода;
- длинный метод;

- большой класс;
- длинный список параметров;
- «завистливые» функции — это метод, который чрезмерно обращается к данным другого объекта;
- избыточные временные переменные;
- классы данных;
- не сгруппированные данные

Каждый из вышеуказанных признаков свидетельствует о том, что код является неэффективным. Без модификации этот код может повлечь за собой ещё большее количество неоптимального кода, что в конечном итоге приводит к труднопонимаемому коду, который будет содержать трудно выявляемые ошибки. Методы рефакторинга позволяют оптимизировать программный код и исключать накопление ошибок.

Цикл операций при разработке с использованием TDD выглядит следующим образом:

1. Написание теста (описание будущего функционала)
2. Написание кода — «заглушки» для компиляции программы
3. Написание полного кода для того, чтобы выполнялись все тесты.
4. Рефакторинг.

Рефакторинг является неотъемлемой частью TDD. Одним из способов рефакторинга является рефакторинг с использованием шаблонов (паттернов). Шаблоны проектирования — это типовые приёмы программирования, позволяющие их использовать в большинстве операций при проектировании и разработке программного обеспечения [3].

2. S.O.L.I.D.

Также в приложениях могут быть использованы принципы объектно-ориентированного проектирования, называемые S.O.L.I.D., по первым буквам названий методов на английском языке. [4]

Single responsibility principles — принцип единственности ответственности подразумевает, что каждый класс должен выполнять только одну задачу (должна лежать только одна ответственность).

Open/closed principles — принцип закрытости/открытости подразумевает, что сущности должны быть открыты для расширения, но закрыты для изменений.

Liskov substitution principle — принцип Лисков означает, что сущность может быть заменена своими потомками (в рамках наследования) без изменения свойств программы.

Interface segregation principle — принцип разделения интерфейса означает, что различные частные интерфейсы являются более предпочтительными, чем один большой и всеобъемлющий интерфейс.

Dependency inversion principle — принцип инверсии зависимостей подразумевает следующее:

1. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракции.

2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

3. Ограничения методологий при разработке научного программного обеспечения

Несмотря на все преимущества перечисленных выше методологий при разработке научно-исследовательского ПО существует ряд условий, ограничивающих применение указанных выше методик.

Одним из условий применения TDD является точный результат, который должен быть получен в итоге. В случае бизнес-систем результат будет предсказуемым: бизнес-процессы максимально детерминированы для исключения рисковых ситуаций. При выполнении научных вычислений результат заранее неизвестен. Поэтому при использовании TDD часть интеграционных тестов состоит в том, чтобы проверить, сходится ли алгоритм: ограничены ли каким-нибудь конечным значением его результаты.

Вторым способом проверки является проверка на отсутствие запрещённых состояний (к примеру, результаты работы алгоритма должны быть неотрицательными и др.).

Третьим способом проверки является наличие «типовых значений», т.е. конкретные решения либо заранее известные результаты работы алгоритма при определённых входных значениях.

В случае, если алгоритм разбивается на блоки, то с помощью TDD можно проверить каждую из частей алгоритма. Это позволяет выявить «арифметические» ошибки при написании кода. Подобные тесты помогут выявить правильный порядок арифметических выражений, выявить, к примеру, использование сложения вместо вычитания, что возможно при написании больших участков кода, над которыми невозможно провести рефакторинг.

Тесты помогают проверить инфраструктуру, окружающую исследуемый алгоритм: корректность подготовки входных данных, обработки полученных результатов, операции импорта и экспорта данных.

В связи с этим использование TDD, несмотря на кажущуюся избыточность, позволяет максимально исключить неопределённость при исследовании какого-

либо явления или алгоритма и целиком сосредоточиться на исследовании проблемы. Применение методологий гибкой разработки также содержит кажущуюся, на первый взгляд, избыточность, однако, на этапе исследований позволяют быстро переключать акценты и направления исследований без переработки больших участков кода.

Зачастую пробы изменения входных данных, коэффициентов, отдельных блоков обработки данных для получения определённых результатов и есть важнейшая составляющая научной работы. В случае неадекватных результатов система контроля версий позволит быстро вернуться к предыдущему этапу разработки.

4. Заключение

Использование методов экстремального программирования при разработке научного ПО позволяет быстро пробовать и применять новые идеи и методы. Это позволяет в достаточно сжатые сроки проверять научные предположения, возникающие при изучении той или иной проблемы. Несмотря на кажущуюся избыточность, методы экстремального программирования позволяют сохранить гибкость системы при её росте, что впоследствии позволяет избежать коллапса и необходимости разработки системы заново.

Литература

1. Бек К. Экстремальное программирование. Питер, 2002. С. 79–90.
2. Фаулер М. Рефакторинг: улучшение существующего кода. СПб: Символ-Плюс, 2003.
3. Кириевски Д. Рефакторинг с использованием шаблонов. М.: ООО «И.Д. Вильямс», 2006. с. 400.
4. Мартин Р., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#. СПб: Символ-Плюс, 2011. с. 768.

THE USE OF EXTREME PROGRAMMING IN THE DEVELOPMENT OF SCIENTIFIC APPLICATIONS

V.S. Vedeneev
Postgraduate, e-mail: ingafen@gmail.com
I.V. Byichkov
Doctor of Physico-Mathematical Science, Prof., e-mail: bychkov@csu.ru

Chelyabinsk State University

Abstract. The usage of extreme programming in development of scientific software is described in this work.

Keywords: extreme programming, refactoring.