УДК 004.438

ПРИЧИНЫ И ДВИЖУЩИЕ СИЛЫ ПОЭТАПНОЙ СМЕНЫ ПОДХОДОВ ВНУТРИОБЪЕКТНО-ОРИЕНТИРОВАННОЙ ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

***Галина Михайловна Рудакова**, к.ф.-м.н., профессор*
Тел.: +7 983 150 7529, email:gmrfait@gmail.com
Сибирский государственный технологический университет,
кафедра информационных технологий
***Дмитрий Олегович Кожевников**, аспирант*
Тел.: +7 953 594 2082, email: d.o.kozhevnikov@gmail.com
Сибирский государственный технологический университет
кафедра информационных технологий
http://www.kit-sibstu.ru/

В работе представлена характеристика важнейших когнитивных событий истории парадигмы, предпосылки, авторство и значение её основных научных концепций. Предложена следующая периодизация эволюции объектно-ориентированной парадигмы (ООП) в теории разработки программного обеспечения (ПО): академическая ООП; ранняя ООП; зрелая ООП; современная ООП. Развитие обусловлено стремлением задействовать человеческий потенциал в разработке эволюционирующих программных комплексов.

Ключевые слова:объектно-ориентированная парадигма, программное обеспечение, концепции паттернов проектирования,рефакторинг

Объектно-ориентированная парадигма, являясь на данный момент доминирующей, как система теорий, подходов и методов вызывает много споров в профессиональном и научном сообществе. Горячие дискуссии вызывает не только современное её состояние и перспективы будущего развития, но и истоки, предпосылки зарождения и причины перехода к тем или иным нынешним формам. При этом вся история объектно-ориентированных подходов занимает всего 50 лет.

Г.М. Рудакова

1 Краткое введение в разработку программного обеспечения

Д.О. Кожевников

Программное обеспечение (ПО), технологии его разработки и соответствующая профильная научная дисциплина, как раздел информатики, явились частью третьей научно-технической революции по Тоффлеру – информационной революции, которая привела к появлению постиндустриального общества. Разработка ПОв общем смысле - это область человеческой деятельности по созданию программных средств, комплексов и систем. Разработка ПО неотделима от таких видов деятельности как проектирование, анализ требований, внедрение, сопровождение, поддержание качества ПО.

Одной из ключевых тем теории разработки ПО является изучение разработки как процесса взаимодействия разработчиков между собой, программой и компьютером. При этом конечной целью является выработка более успешных моделей взаимодействия, которые позволили бы разрабатывать более сложные программные комплексы быстрее и качественнее чем раньше. Основой любой модели взаимодействия является язык программирования. Важно и то, что он является средством взаимодействия разра-

ботчиков между собой, выполняя порой функцию lingua franca для международного сообщества разработчиков.

Существует огромное множество языков и их классификаций, однако в контексте данной работы необходимо отметить следующие важнейшие характеристики языка программирования: моделирующая способность и выразительная сила.

Однако, хотя язык программирования является основой взаимодействия в процессе разработки ПО, существуют и другие важные аспекты этого взаимодействия. Его характер менялся во времени с появлением новых задач, которые вставали перед разработчиками, в процессе информатизации общества. Программное обеспечение быстро шло по пути усложнения. Наряду с разработкой новых языков программирования, требовались более продвинутые подходы по организации программного кода, управлению программными проектами, тестированию, управлению командами разработчиков, коммуникации между разработчиками.

Для того чтобы понять предпосылки становления нынешнего состояния теории разработки ПО и доминирования объектно-ориентированной парадигмы, необходимо кратко рассмотреть эволюцию подходов и научных взглядов, предшествующих современному этапу развития.

2. Структурная парадигма программирования

Первые программы не были программным обеспечением в современном смысле этих слов и представляли собой двоичный или машинный код в том или ином виде, то есть последовательно нулей и единиц. Машинный код легко интерпретируется компьютером и совершенно не приспособлен для понимания человеком. Написание программ стало видом интеллектуальной деятельности, связанной с особым, несколько чуждым человеку типом мышления – компьютерной логикой.

Поэтому основной движущей силой развития ранних языков программирования была необходимость приблизить язык взаимодействия с компьютером к естественному человеческому языку. Так появились языки, основанные на командах и директивах. Компьютеры так и не научились понимать что-либо кроме двоичного кода, однако выход был найден в виде специализированных служебных программ, которые транслировали команды в двоичные коды. Таким образом, разработчики получили возможность работать с более понятными конструкциями и текстом программы, который походит на структурированный текст на естественном языке. Примерами таких языков можно назвать Fortran, Cobol, Algol, C.

Разработка ПО шагнула из научных центров и лабораторий во многие сферы человеческой деятельности, например, бизнес, образование и государственное управление. Тогда же сформировалось несколько конкурирующих парадигм разработки, доминирующие положение среди которых заняло структурное программирование.

Становление структурной парадигмы программирования принято связывать с именами Эдсгера Дейкстры и Никлауса Вирта и относить к 70-м годам двадцатого века. Методология структурного программирования возникла как результат борьбы с увеличивающейся сложностью разработки ПО. Новые языки программирования, более мощные и простые в использовании позволили создавать большие программные комплексы для самых разных задач. Оборотной стороной этого стала сложность, как самих этих программ, так и процесса их разработки.

Программный код представляет собой текст, составленный на языке программирования согласно его правилам. При этом составитель текста программы, то есть разработчик, волен в определённых пределах придавать тексту различную структуру, форматирование и стиль, по разному связывать части программы отношениями зависимости. Оказалось, что эти характеристики во многом определяют способность разработчика и команды в целом работать с программой. Подобно тому, как хорошо структурированный конспект легко читаем, хорошо структурированная программа сама по себе даёт разработчикам больше возможностей для дальнейшего развития и модификации.

Сложность разработки программного обеспечения быстро выделилась как отдельное разностороннее явление, которое необходимо минимизировать. Именно сложность самих программ и процесса разработки, а не языков программирования стали главной головной болью разработчиков. Структурная парадигма предложила простую методологию, сочетающую практики по управлению текстом программы с приёмами структурного анализа проблемы, на решение которой направлена программа. Текст программы разбивается на строгую иерархию небольших подпрограмм, при этом набор базовых конструкций строго ограничивается, соответственно декомпозируется проблема. Работа над подпрограммами ведётся последовательно сверху вниз, от общих к более частным. На «вершине» находится головная программа. Таким образом, в каждый момент времени разработчик имеет дело с небольшим текстом одной подпрограммы.

Структурная парадигма программирования направляет разработчиков формировать программу как иерархию подпрограмм, решая проблему сложности ПОза счёт уменьшения размеров фрагмента программы, который необходимо держать в голове для продолжения работы. Множество практик структурного программирования направлены на работу с текстом и имеют целью повысить его читаемость.

3. Зарождение объектно-ориентированной парадигмы

Несмотря на ряд неоспоримых успехов структурной парадигмы, не все аспекты проблемы сложности разработки ПО решались ею одинаково успешно. Так иерархия подпрограмм хорошо подходит для описания работы алгоритма или линейного процесса, однако плохо справляется с описанием предметной области программы в целом. Так как область применения программного обеспечения постоянно расширялась, усложнялись и те данные и процессы, управление которыми оно должно автоматизировать. Однако возможности структурной парадигмы исчерпывались работой с текстом обширной программы.

Недостаточно мощным было и представление данных в структурной парадигме. Объектами работы здесь являются такие единицы как строка, число, символ, дата и время. В реальном мире данные редко существуют отдельно, чаще всего они сгруппированы и ассоциированы с некоторой самостоятельной сущностью. Например, персональные данные человека представляют собой неделимое целое.

Начиная с середины 70 годов двадцатого века, проблема моделирования предметной области программного обеспечения заявляет о себе все отчётливей. Завоёвывает популярность идея ПО как модели автоматизируемого процесса. К этому времени относится появление первых объектно-ориентированных языков общего назначения.

Зарождение объектно-ориентированных представлений в теории разработки ПО относится к 60 годам и связано с именами Ивана Сазерленда и Алана Кея, крестного отца объектно-ориентированного программирования. В основе этих представлений лежала идея сделать язык программирования инструментом моделирования реальных процессов и явлений. Для этого в языки было предложено добавить возможность определять собственные типы данных и наделять их не только способностью хранить связанную информацию, но и иметь собственное поведение.

Собственные абстрактные типы данных - классы (фреймы, шаблоны) – позволяли определять модели реальных объектов, описывать их поведения и отношения гораздо нагляднее и проще, чем это было возможно в программах структурной парадигмы. Класс объединяет – инкапсулирует - данные и поведение некоторой категории объектов и предоставляет остальной программе возможность работать со своими объектами – объектами класса – как с самостоятельными единицами данных. Объекты обладают внутренним состоянием, внешним поведением и способны взаимодействовать с другими объектами, образуя сложные системы и реализуя требуемые варианты поведения. Первым объектно-ориентированным языком получившим широкое распространение стал Smalltalk-80 Алана Кея, а мировое признание снискал язык C++, представленный Бьёрном Страуструпом в 1983 году.

Объектно-ориентированная революция изменила разработку ПО и сами программы до неузнаваемости. Отныне программы представляли собой систему взаимодействующих объектов и старались моделировать существенные объекты и взаимосвязи своих предметных областей. Так в системах банковских расчётов обязательными стали классы Счёт, Транзакция и Платёж. Весь программный код теперь заключался внутрь классов, а иерархии подпрограмм пришла на смену сложная композиция объектов.

Так как программы продолжают разрабатываться как тексты, практики структурного программирования не потеряли свою актуальность. Однако область их применения сузилась и ограничивается кодом поведения внутри класса.

Объектно-ориентированная парадигма предложила разработчикам набор философских принципов правильного проектирования, конструирования и использования классов и объектов:

1) Абстрагирование. Всякий класс должен представлять собой хорошо согласованную абстракцию, то есть отражать существенные для решения задачи характеристики некого цельного реального или виртуального объекта.

2) Инкапсуляция. Всякий класс объединяется данные и логику работающую с этими данными и тем самым концентрирует связанные аспекты программы в одном месте, внутри одного объекта.

3) Сокрытие информации. Всякий объект класса является чёрным ящиков, вещью в себе для внешних объектов и взаимодействует с ними через строго очерченный интерфейс, скрывая свои данные как внутреннее состояние. Тем самым ответственность за определённые данные остаётся только у него.

4) Наследование. Класс может унаследовать характеристики – данные и поведение – у другого класса, тем самым повторно используя уже написанный код, добавляя собственную специфику к характеристикам класса предка.

5) Полиморфизм. Один интерфейс – множество реализаций. В рамках иерархии наследования классы могут изменять внутреннюю логику, оставляя внешнее поведение неизменным, таким образом, скрывая от внешних объектов некоторую условную логику изменения поведения.

Само по себе добавление объектно-ориентированных механизмов в языки программирования не даёт разработчику никаких преимуществ. Наоборот, этот тонкий и в тоже время мощный инструмент требует большой теоретической и философской подготовки, а при неумелом использовании способен порождать отчаянно запутанные и сложные программные системы. По сути, теоретики объектно-ориентированной парадигмы, такие как Гради Буч и Бертран Мейер, потребовали от разработчиков полной перестройки мышления. Результатом стали описанные выше принципы, которые скорее относятся к методологии моделирования и анализа, чем к написанию программного кода. Глубокое осмысление принципов, которое приходит только с опытом, стало неотъемлемой частью профессии разработчика и частью его мышления.

Моделирующая мощь объектно-ориентированной парадигмы уже к середине 80-х годов сделала её доминирующей парадигмой в теории разработки ПО, а множество научных работ теперь были направлены на совершенствование объектно-ориентированных подходов. Главным достижением этих ранних подходов стало появление инструмента для моделирования и анализа предметной области.

4. Зрелость парадигмы

Последнее десятилетие XX века в теории разработки программного обеспечения – это период бурного развития объектно-ориентированных подходов, на фоне беспрецедентного роста рынков программных решений для всех областей человеческой деятельности. С появлением Интернета ПО действительно стало проникать повсюду и работать для всех. Для объектно-ориентированной парадигмы это также время обобщения накопленного опыта и новой эволюции подходов.

К предпосылкам нового скачка в развитии парадигмы можно отнести, как дальнейшее усложнение решаемых задач, так и становление сообщества разработчиков, применяющих теорию ООП практике. Завоевав популярность в 80-е, объектно-ориентированная парадигма в 90-е обзавелась интернациональным сетевым сообществом приверженцев, в силу своей профессии работающих со средствами массовой коммуникации, склонных к коммуникации и командной работе над сложными инженерными и научными проблемами. Новые идеи и подходы, зарождающиеся внутри сообщества, быстро распространяются, проходят отбор и пополняют совокупный опыт сообщества.

Совершенствование языков и технологий программирования хоть и не остановилось, но уже перестало быть основным направлением развития теории разработки ПО. Они стали восприниматься в качестве инструментов разработчика подходящих для решения тех или иных задачи, сменяющихся гораздо быстрее, чем происходят сдвиги в общих подходах к разработке. Ядром набора профессиональных компетенций разработчика становятся освоенные и просто осознаваемые ими теоретические основы и подходы, соотносящиеся с текущим уровнем сообщества.

Во многих случаях для обозначения базовых теоретических концепций, структуры глубинных взаимосвязей и практик правильного использования некоторой технологии стало применяться понятие философия. Всякая технология, например объектно-ориентированный язык программирования, является обоюдоострым мечом, который в неумелых руках способен натворить бед. Авторы и разработчики языка не могут запретить неправильное использование его возможностей, однако всегда закладывают в своё творение систему идей и правил, которые ведут к элегантным и эффективным решениям. Технологии и языки без выраженной и заявленной философии практически бесполезны, так как не предлагают системного подхода к решению поставленных задач, то есть не согласуются положениями теории разработки ПО.

Таким образом, новые технологии всегда ориентированны на господствующие в глобальном сообществе концепции разработки, а для разработчика очень важно разделять уровень сообщества, для того чтобы быстро входить в философию вновь появляющихся технологий. Сетевой характер, творческая энергия и постоянно растущий совокупный опыт сообщества позволяют ему играть определяющую роль в эволюции, как теории, так и технологий разработки.

Систематизация опыта, накопленного сообществом к началу 90 годов, привела появлению и быстрому распространению концепции паттернов проектирования[1], которая стала новым языком объектно-ориентированной парадигмы. Паттерн проектирования – это формально описанный, именованный, зарекомендовавший себя подход или приём решения определённой категории задач, который можно использовать повторно. Паттерн состоит из четырёх основных элементов:

1) Имя. Точное и запоминающееся наименование, ссылка на которое в литературе, документации или совещании сразу определяет решаемую задачу, предлагаемы подход её решения и последствия для остальной системы;

2) Задача. Проектная ситуация, контекст или условия, в которой можно применить паттерн для получения более прозрачной и понятной структуры программы или избежания дублирования кода;

3) Решение. Обобщённое описание элементов и взаимосвязей предлагаемого решения, то есть структуры и динамики классов и объектов, на которые будет возложено решение задачи;

4) Результаты Последствия применения паттерна для остальной программы и соответствующие компромиссы, на которые придётся пойти, для того, чтобы решение вписалось в существующую структуру.

Идея паттернов проектирования была выдвинута архитектором Кристофером Александером ещё 1970-х. Он предложил свой «язык шаблонов» для решения часто

возникающих проблем проектирования архитектурных сооружений. В конце 80-х такие программисты как Кент Бек, Уорд Каннингэм, Джеймс Коплин и Эрих Гамма исследуют возможность применения концепции паттернов в разработке ПО. В 1994 году выходит книга «Design Patterns — Elements of Reusable Object-Oriented Software»(«Приёмы объектно-ориентированного проектирования») четырёх авторов - Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса, в которой представлен первый каталог паттернов проектирования. Все паттерны описаны в понятиях объектно-ориентированной парадигмы. Заодно в теорию разработки ПО проник термин «архитектура», который стал обозначать верхнеуровневое устройство программного комплекса в целом.

Паттерны проектирования обобщают опыт многих разработчиков и предлагают готовые, хорошо себя зарекомендовавшие, но далеко не очевидные решения постоянно возникающих задач моделирования и проектирования. При этом паттерны проектирования универсальны для объектно-ориентированной парадигмы и реализуются на любом языке в ней. При описании паттерна теоретики стараются сделать философскую выжимку и представить её в формальном виде.

Паттерны проектирования добавили новый уровень абстракции в объектно-ориентированную парадигму, и подобно классам и объектам в своё время сформировали новый тип мышления разработчиков. Теперь процесс моделирования предметной области и проектирования программного продукта ведётся на языке паттернов, который доступен каждому разработчику в сообществе, хотя его освоение требует серьёзных усилий и определённой опыта.

Несмотря на довольно высокий порог входа, концепция паттернов проектирования завоевала популярность в сообществе и позволила выработать более эффективный базис проектирования крупных программных комплексов. Существенной частью жизни сообщества стала выработка новых паттернов проектирования, дифференциация и каталогизация паттернов по областям применения, а также описание наилучших практик применения, как самих паттернов, так и других подходов с применением паттернов. Отныне именно на языке паттернов формулируются новые подходы и теории.

Также получила широкое распространение оригинальная альтернативная модель обобщения опыта – антипаттерны проектирования. Это примеры ситуаций, приёмов или подходов, которые зарекомендовали себя как однозначно ущербные, ведущие к затруднению понимания программы или другим негативным последствиям. Работа над новыми паттернами и осмыслением уже существующих никогда не прекращается. Нередки случаи, когда паттерн переквалифицируется в антипаттерн.

В конечном итоге формирование неписанного, но общего для всего сообщества каталога паттернов проектирования способствовало не только распространению опыта и облегчению коммуникации. Популярные паттерны часто ложатся в основу философии новой технологии, более того именно такие технологии имеют лучшие перспективы на рынке, так как реализуют теорию, проверенную опытом. Технологии, закладывающие поддержку того или иного паттерна по умолчанию, не только способствуют росту его популярности, но росту конструктивной критики в его адрес, повторному анализу и модификации. То есть, в конечном счёте, обогащают теорию разработки программного обеспечения.

Взаимозависимость и взаимовлияние теории и технологий во многом обусловили современный идеал успешного разработчика. Несмотря на то, что разработчики попрежнему работают с конкретными технологиями, имеют специализацию и разделяются на множество групп по предпочтениям в используемых инструментах, для каждого из них жизненно важно понимать философию разработки в целом, а также находиться в курсе современных теорий, которые через год-два уже выйдут на рынок в виде новой версии языка.

Среди теоретиков, труды которых сильно повлияли на технологии, появившиеся в XXI веке, можно назвать Мартина Фаулера, Эндрю Ханта, Дэйва Томаса, Роберта Мартина.

5. Современное состояние парадигмы

Как уже отмечалось со времён до появления структурной парадигмы программирования разработчики сталкиваются с проблемой сложности разрабатываемого ПО. Не только процесс разработки и проектирования, но и сами получившиеся приложения сложны для понимания. Природа программных продуктов такова, что в большинстве случаев их развитие, то есть расширение функциональности, реинжиниринг, выпуск новых версий, прекращаются только тогда, когда продукт перестаёт пользоваться спросом у пользователей.

Некоторые проекты не доживают до первой стабильной версии, а другие сменяют несколько десятков. Однако в течение своей жизни программный код приложения постоянно изменяется и совершенствуется. Это связано как с динамикой рынка и требований пользователей, так и с природой программ. Исходный код программы представляет собой текст на языке программирования, то есть теоретически это очень гибкая система, ведь формализованный текст можно переработать и усовершенствовать. Однако на практике внесение изменений программный код является очень сложной задачей.

Наибольшие трудности связаны с тем, чтобы при внедрении новой функциональности не нарушить согласованную работу уже написанного кода. При этом решающую роль играет понятность изменяемого кода, его качество, степень, в которой разработчики с самого начала ориентировались на последующие изменения. Большинство практик объектно-ориентированного проектирования направлены на то, чтобы сделать программный код более гибким и легко расширяемым. Также немалую роль играет интуитивное предугадывание направления эволюции функциональности проекта. Не подлежит сомнению то, что написание гибкого кода требует от разработчика более серьёзной подготовки и в первую очередь осознания философских основ разработки.

Однако большинство теоретиков сходится в том, что невозможно безошибочно предугадать все изменения и заложить в проект только необходимую долю гибкости не под силу даже самому опытному разработчику. А излишняя, невостребованная гибкость самых совершенных паттернов только умножает сложность.

Проблема сложности программного обеспечения связана также с явлением деградации программного кода во времени. Разработчики всякого крупного продукта сталкиваются с тем, что со временем их программа теряет согласованность и стройность, в ней накапливаются ошибки и несообразности. Это связано с тем, что человек не способен удерживать в памяти всю программу одновременно и учитывать все последствия при добавлении нового кода. Ситуация усугубляется в случае командной работы. Деградацию невозможно предотвратить, умелое применение практик объектно-ориентированного программирования может только замедлить её.

Не решают проблему ни тщательное, вплоть до мелких деталей проектирование, ни обширная документация, ни строгие договорные обязательства между заказчиком и разработчиками. К началу 2000-х такие практики зарекомендовали себя как негибкие, а основой новых методологий стала концепция рефакторинга, которую развивали такие теоретики как Бек, Каннингэм, Джонсон и Фаулер. Последний внёс большой вклад в популяризацию рефакторинга в сообществе.

По Фаулеру рефакторинг – это «изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения»[3]. Другими словами, это вид деятельности разработчика направленный на устранение последствий деградации ПОпутём реорганизации кода и приведения его в порядок без добавления новой функциональности.

Революционность данного подхода заключалась в том, что разработчик, выполняя работу по созданию полезного работающего кода, мог меньше думать о будущих изменениях и делать только необходимое. А во время рефакторинга он уделял внимание согласованности программы в целом, прозрачности и понятности тех или иных решений и фрагментов кода. Разумеется, для этого разработка и рефакторинг должны ритмично чередоваться.

Рефакторинг ни в коем случае не отменяет проектной работы и применения навыков объектно-ориентированного моделирования, наоборот он на них основан, однако он позволяет отложить принятие некоторых решений на тот момент времени, когда реальное развитие программы прояснит ситуацию. В этом состоит идейное основание рефакторинга, однако на этом он не заканчивается.

В 1999 году выходит фундаментальный труд «Refactoring: ImprovingtheDesignofExistingCode» («Рефакторинг: улучшение существующего кода») за авторством Мартина Фаулера, Кента Бека, Джона Бранта, Уильяма Апдайка и Дона Робертса. В этой книге были впервые каталогизированы паттерны и систематизированы основные идеи рефакторинга. А в 2004 увидела свет книга Джошуа Кериевски «RefactoringToPatterns» («Рефакторинг с использованием шаблонов»), в которой он указал связь рефакторинга с классическими паттернами проектирования.

Применение рефакторинга при разработке крупного программного продукта даёт целый ряд преимуществ:

1) Борьба с деградацией программного обеспечения. Регулярные системные мероприятия по наведению порядка способствуют итеративному формированию программной архитектуры обладающей релевантным уровнем гибкости и абстрактности;

2) Повышение понятности программного обеспечения. Рефакторинг позволяет избавиться от не оправдавших себя усложнений, устранить лишние уровни абстракции и дублирование кода, тем самым облегчая работу по дальнейшему развитию проекта;

3) Уменьшение количества ошибок. Рефакторинг подразумевает более глубокое осмысление написанного программного кода, поэтому не только автоматически устраняет мелкие ошибки, но и подталкивает к выявлению идейных, глубоко залегающих упущений в модели предметной области;

4) Ускорение процесса разработки. Совокупность положительных эффектов от рефакторинга даёт существенное ускорение работы команды разработчиков даже в среднесрочной перспективе.

Таким образом, программный продукт развивается вокруг оберегаемой целостности идей и механизмов, которую также принято называть философией. Изначально закладываемые в основу философии продукта проектные решения эволюционируют и упорядочиваются, а новые идеи со временем вписываются в общую схему наиболее прозрачным образом. Без рефакторинга даже самые философски цельные программы со временем теряют свою внутреннюю согласованность и становятся всё более запутанными и неповоротливыми. Развивать такие продукты слишком сложно и дорого.

Системное применение такой продвинутой техники как рефакторинг требует большой самоотдачи и мотивации, не говоря уже о глубоком осмыслении философских основ разработки ПО. Рефакторинг по своей сути это работа на завтрашний день в ущерб интересам дня сегодняшнего, поэтому дисциплинированность и сознательность разработчика играет в ней подчас решающую роль.

Кент Бек в своих работах[2] посвящённых методологии экстремального программирования идёт дальше и предлагает вписать рефакторинг в оригинальную новаторскую технику разработки через тестирование(TestDrivenDevelopment, TDD). Тестированием в теории разработки ПО называется деятельность по проверке правильности работы программы, в том числе с помощью написания других программ – тестов. Единственное назначение тестов в том, чтобы проверять реальный программных про-

дукт. Традиционно этап тестирования следует за этапом разработки рабочей функциональности.

Разработка через тестирование выводит работу с одним тестом в качестве такта процесса разработки. Такт начинается с написания теста – небольшой, простой программы, которая проверяет правильность работы некоторого аспекта требуемой функциональности. Тест не срабатывает, так как никакого полезного кода ещё не написано. После этого разработчик реализует требуемую функциональность самым простым и понятным образом, какой может выдумать. Тест срабатывает. Такт завершается сеансом рефакторинга, в течение которого разработчик вписывает новый код в общую архитектуру, обобщает и упрощает структуру программы в целом.

Работая в таком ритме, разработчик всегда имеет один нереализованный тест и очень небольшой участок программы, который необходимо держать в голове. Кроме того, в его распоряжении набор уже реализованных тестов. С его помощью можно быстро проверить правильность программы в любой момент. Средства автоматизированного тестирования выполняют сотни тестов на секунды.

Таким образом, разработчику не нужно думать о том, когда проводить рефакторинг и сколько времени ему уделять, его рабочий процесс подчиняется ритму разработки через тестирование. Кент Бек указывает на то, что применение методологии TDD позволяет перераспределить проектную работу по множеству сеансов рефакторинга и вовсе бросить попытки заранее предсказать будущие изменения. Все изменения вписываются в программу в своё время, а проектные решения принимается на основании реальной необходимости. Философия продукта вырастает и развивается как бы сама собой.

При этом продукт всегда нацелен на реальные задачи, так как всё его развитие направляют тесты. Философия TDD органично и чётко вводит рефакторинг как обязательный элемент разработки и в очередной раз перестраивает мышление разработчика. Соблюдение постоянного ритма, осознание ограниченности своих возможностей по контролированию больших объёмов кода, понимание принципиальной непредсказуемости последствий новых изменений становятся частью мышления. Вместе с тем тесты до некоторой степени гарантируют безопасность, контролируемость и результативность работы.

В начале 2000-х идеи рефакторинга и гибкой разработки произвели революцию в теории разработки ПО и породили семейство Agile (гибких) методологий разработки, которые сочетают философские, инженерные и управленческие подходы в создании и развитии современных программных продуктов.

В 2001 году группа из 17 авторитетных разработчиков подписала Agile манифест[4], содержащий основные философские принципы гибких методологий разработки:

1) Люди и взаимодействия важнее, чем процессы и инструменты
2) Работающее программное обеспечение важнее, чем полная документация;
3) Сотрудничество с заказчиком важнее, чем контрактные обязательства;
4) Реакция на изменения важнее, чем следование плану.

Авторы манифеста: Кент Бек, Майк Бидл, Эйри Ван Баннекум, Алистер Кокбёрн, Уорд Каннингэм, Джеймс Греннинг, Стивен Меллор, Мартин Фаулер, Джим Хайсмит, Эндрю Хант, Рон Джеффрис, Джон Кёрн, Брайан Мэрик, Роберт Мартин, Кен Швабер, Джефф Сазерленд, Дэйв Томас. Развивая различные методологии авторы манифеста декларируют, что разделяют общие ценности и принципы.

Agile методологии больше ориентируются на отдельных людей, команды, коммуникации и стараются использовать, как особенности мышления и поведения разработчиков, так и природу эволюции программных продуктов. Среди методологий внёсших серьёзный вклад в теорию разработки ПО можно выделить Scrum и XP(экстремальное программирование).

В целом Agile методологии направлены на гуманизацию процессов спецификации требований, проектирования и разработки с отходом от традиционных инженерных практик. Agile провозглашает ценностью человеческие отношения не только внутри команды, но между командой и заказчиком. Ни одна гибкая методология не принесёт результатов в разобщённой команде с враждебным настроем по отношению к пользователям или менеджменту.

Многие практики Agile методологий направлены на возбуждение интереса и азарта в среде разработчиков, другие на обмен опытом и вовлечение в принятие решений, третьи на развитие лидерских и коммуникативных навыков. В Agile принято считать, что разработчик должен получать удовольствие от своей работы.

Развитие гибких методологий разработки в противовес механистическим и перегруженным инженерным методологиям определило современный облик мира разработки программного обеспечения. На сегодняшний день они задают тон и тематику основной массы исследований. Так одними из самых перспективных считаются идейные наследники разработки через тестирование BeheviorDrivenDevelopment(BDD) и AcceptanceTestDrivenDevelopment(ATDD), выводящие тесты, которые направляют разработку, на новый уровень.

В области проектирования в объектно-ориентированной парадигме, например в работах Эрика Эванса, громко заявляет о себе методология DomainDrivenDesign[5], DDD, развивающая продвинутые подходы к моделированию предметной области приложений. DDD основана на каталоге паттернов проектирования специально разработанных для выявления существенных закономерностей запутанных и непрозрачных предметных областей. Кроме того, применяется итеративный подход и эволюцию модели предметной области вместе с ростом взаимопонимания разработчиков и пользователей. DDD всегда в ожидании прорыва, скачкообразного обновления модели по мере углубления осмысления предметной области.

6. Выводы

По результатам проведённого исследования [6,7] предлагается следующая периодизация эволюции объектно-ориентированной парадигмы (ООП) в теории разработки программного обеспечения (ПО):

1) Академическая ООП, с начала 1960-х до конца 1970-х. Зарождение объектно-ориентированных подходов в работах и экспериментальных языках Алана Кея и Ивана Сазерленда;

2) Ранняя ООП, 1980-е. Становление классических философских принципов объектно-ориентированной парадигмы. Развитие обусловлено потребностью моделирования сложных предметных областей;

3) Зрелая ООП, 1990-е. Популяризация концепции паттернов проектирования. Развитие обусловлено стремление систематизировать опыт сообщества для решения сложных программных комплексов;

4) Современная ООП, с начала 2000-х. Гуманизация теорий разработки ПО, замена детального проектирования рефакторингом. Развитие обусловлено стремлением задействовать человеческий потенциал в разработке эволюционирующих программных комплексов.

Литература

1. *Гамма Э.* Приёмы объектно-ориентированного проектирования. Паттерны проектирования/ Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – СПб.: Питер, 2012. – 368 с.

2. *Бек К.* Экстремальное программирование. Библиотека программиста / К. Бек. – СПб.: Питер, 2003. – 212 с.

3. Фаулер М. Рефакторинг. Улучшение существующего кода / М. Фаулер. – СПб.: Символ-плюс, 2010. – 432 с.

4. *Мартин Р.С.* Принципы, паттерны и методики гибкой разработки на языке C#. / Р.С. Мартин. – СПб.: Символ, 2012. – 768 с.

5. *Эванс Э.* Предметно-ориентированное проектирование. Структуризация сложных программных систем/ Э. Эванс. – М.: ООО «И.Д. Вильямс», 2011.– 448 с.

6. *Kozhevnikov D.O., Rudakova G.M.* History of structured programming and origin of objective-oriented paradigm. //2nd International Academic Conference on Applied and Fundamental Studies, March 8-10, 2013., St. Louis, USA. Publishing House «Science & Innovation Center», 2013. P. 174-180.

7. *Kozhevnikov D.O., Rudakova G.M*. Evolution of the object-oriented paradigm in software development. // 2nd International Academic Conference on Applied and Fundamental Studies, March 8-10, 2013. – St. Louis, USA. Publishing House «Science & Innovation Center», 2013. P. 180-186.

The reasons and driving forces of stage-by-stage change of approaches in an object-oriented paradigm of programming

Galina Mikhailovna Rudakova, PhD, professor
Siberian State Technological University, Chair of Information Technologies.

Dmitry Olegivich Kozhevnikov, postgraduate
Siberian State Technological University, Chair of Information Technologies.
http://www.kit-sibstu.ru/

In work the characteristic of the major cognitive events of history of a paradigm, the prerequisite, authorship and importance of its main scientific concepts are presented. The following periodization of evolution of the object-oriented paradigm (OOP) in the theory of software development is offered: academic OOP; early OOP; mature OOP; modern OOP. The progress is caused by aspiration to engage human potential in development of evolving program complexes.

Key words: object-oriented paradigm, software, concepts of design patterns, refactoring.

УДК 519.6

ИСПОЛЬЗОВАНИЕ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ МЕТРИК ДЛЯ АНАЛИЗА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

***Ольга Алексеевна Самойликова**, магистр*
Тел.: 8 (902) 992-54-57, e-mail: olga.samoylikova@yandex.ru
***Галина Михайловна Рудакова**, к.ф.-м.н., профессор,*
зав .кафедрой информационных технологий
Тел.: +7 (983) 150 7529, e-mail: gmrfait@gmail.com
Сибирский государственный технологический университет

В работе рассматривается использование объектно-ориентированных метрик для анализа и количественной оценки программного обеспечения. Количественная оценка сложности информационной системы производится при помощи метрик, которые принято подразделять на семь классов. Наиболее подробно описывается класс объектно-ориентированных метрик. Особое внимание уделяется набору метрик Чидамбера и Кемерера.

Ключевые слова: метрики, информационная система, проект, класс, метод, инкапсуляция, наследование, экземплярная переменная.набор метрик Чидамбера и Кемерера.