Добрынин А.С., Койнов Р.С., Кулаков С.М., Пургина М.В. О практическом применении гибких механизмов разработки программного обеспечения // Программные системы и вычислительные методы. — 2018. - № 3. - С.45-53. DOI: 10.7256/2454-0714.2018.3.26877. URL: http://e-notabene.ru/ppsvm/article_26877.html

О практическом применении гибких механизмов разработки программного обеспечения

Добрынин Алексей Сергеевич

старший преподаватель, кафедра автоматизации и информационных систем, Сибирский государственный индустриальный университет

654007, Россия, Кемеровская область, г. Новокузнецк, ул. Кирова, 42

Dobrynin Aleksey Sergeevich

Senior Lecturer, Department of Automation and Information Systems, Siberian State Industrial University

654007, Russia, Kemerovskaya oblast', g. Novokuznetsk, ul. Kirova, 42

serpentfly@mail.ru

Другие публикации этого автора

Койнов Роман Сергеевич

ведущий специалист по информатизации, Сибирский государственный индустриальный университет

654041, Россия, Кемеровская область, г. Новокузнецк, ул. Кирова, 42

Koynov Roman Sergeevich

Leading Specialist for Informatization, Siberian State Industrial University

654041, Russia, Kemerovskaya oblast', g. Novokuznetsk, ul. Kirova, 42

koynov_rs@mail.ru

Другие публикации этого автора

Кулаков Станислав Матвеевич

доктор технических наук

профессор, кафедра автоматизации и информационных систем, Сибирский государственный индустриальный университет

654007, Россия, Кемеровская область, г. Новокузнеце, ул. Кирова, 42

Kulakov Stanislav Matveevich

Doctor of Technical Science

Professor, Department of Automation and Information Systems, Siberian State Industrial University

654007, Russia, Kemerovskaya oblast', g. Novokuznetse, ul. Kirova, 42

kulakov-ais@mail.ru

Другие публикации этого автора

Пургина Марина Владимировна

кандидат технических наук

доцент, кафедра информационной безопасности, Сибирский государственный индустриальный университет

630099, Россия, Новосибирская область, г. Новосибирск, ул. Каменская, 56

Purgina Marina Vladimirovna

PhD in Technical Science

Associate Professor, Department of Information Security, Novosibirsk State University of Economics and Management

654041, Russia, Kemerovskaya oblast', g. Novokuznetsk, ul. Kirova, 42

pur-11@yandex.ru

Аннотация.

Предметом исследования являются модели жизненного цикла и подходы к разработке программного обеспечения в условиях значительных временных и финансовых ограничений. Рассматривается рабочий процесс разработки программного обеспечения с использованием гибкой методологии и итерационной модели жизненного цикла. Объектами исследования являются современные подходы к разработке программного обеспечения и повышения эффективности труда за счёт предложенных механизмов стимулирования, основанных на учете требований заказчика. Авторы также уделяют внимание вопросам организации труда при разработке программного обеспечения. В работе использовались методы системного анализа, подходы к принятию коллективных решений в условиях неопределенности, включая методы опроса и экспертных оценок. Авторы предлагают механизм стимулирования для разработчиков программного обеспечения, работающих в рамках итерационной модели неполного жизненного цикла. Важным результатом проведенного исследование является вывод об эффективности подхода к стимулированию, основанного на реализованной функциональности в рамках текущей итерации разработки. Представленный подход позволяет объективно оценить вклад каждого разработчика в решение отдельных задач проекта.
Ключевые слова: гибкая разработка, жизненный цикл, программное обеспечение, разработка, механизм стимулирования, тестирование, итерационная модель, модульное тестирование, управление разработкой, стимулирование разработчика

DOI:

10.7256/2454-0714.2018.3.26877

Дата направления в редакцию:

15-07-2018

Дата рецензирования:

20-07-2018

Дата публикации:

11-10-2018

Keywords:

agile development, life cycle, software, development, incentive mechanism, testing, iterative model, unit testing, development management, stimulating the developer

Введение

В последнее время получили широкое распространение гибкие (agile) методологии разработки программного обеспечения. Это высокоскоростные подходы к созданию качественного программного обеспечения, они ориентированы, в первую очередь, на получение быстрого результата при минимальных затратах, в условиях постоянно изменяющихся требований заказчика. В гибких методологиях основные усилия направлены на получение качественного конечного результата, создание ориентированной на изменения архитектуры программного обеспечения и написание модульных тестов (Unit Tests). Небольшая команда программистов из 4-10 человек работает короткими итерациями (обычно 2-4 недели), целью которых является прирост функций программного продукта и согласование его с заказчиком, при сохранении общего назначения проекта в условиях увеличивающегося риска появления существенных изменений в будущем. В конце каждой итерации осуществляется глобальный рефакторинг кода, что позволяет максимально приспособить текущую программную архитектуру проекта к любым изменениям на следующих итерациях разработки. В отличие от традиционных команд программистов с четким распределением ролей, использующих полноценные модели жизненного цикла продукта, включая каскадную и спиральную модели, у гибких команд часто не остается времени и сил на создание различных дополнительных артефактов в ущерб программированию, включая диаграммы, модели и документацию, оценку рисков. В идеале,

любой участник гибкой команды должен быть способен вносить критические изменения в проект, при необходимости. Статья рассматривает рабочий процесс и механизм стимулирования, для программистов, работающих в условиях гибкой и экстремальной парадигмы разработки программного обеспечения.

Гибкий механизм разработки программного обеспечения

Конечной целью гибкой разработки программного обеспечения является создание ориентированного на изменения эффективного механизма для скоростной разработки качественного программного обеспечения, с учетом современных требований. На первом месте здесь выступают: взаимодействие с заказчиком, следование изменениям, ускоренное написание кода и повышение качества разработки, причем без каких-либо компромиссов по отношению к любым другим аспектам создания программного обеспечения. Это отличает гибкий механизм разработки от других механизмов и общепризнанных классических моделей т.н. «полного» жизненного цикла. Долгое время считалось, что правильная организация «формальных процессов» и исчерпывающее документирование проекта способствует преодолению многих проблем, однако с позиций гибкой разработки этот путь в итоге создает больше проблем, чем решает. Никто не знает, что в итоге понадобится разработчику программного обеспечения, кроме самого разработчика. Часто встречается ситуация, когда излишняя изменчивость продукта приводит к неизбежному падению скорости работы команды программистов и вносит противоречия в постоянно изменяемый проект. Гибкие команды ориентированы на скорость, качество и поддержку изменений, это базовые принципы разработки, которыми нельзя жертвовать при любых обстоятельствах.

Гибкая разработка подразумевает освобождение от любой деятельности, которая затрудняет внесение изменений в проект, приводит к появлению противоречий и снижению скорости написания кода или потере его качества, включая предварительное планирование и написание документации. Некоторые специалисты в области быстрой разработки и экстремального программирования, обоснованно считают, что единственной адекватной разрабатываемой системе инженерной документацией является ее исходный код. По своей сути, большинство общеизвестных моделей разработки программного обеспечения, включая каскадную и спиральную модель жизненного цикла [1], представляют собой неповоротливый, «тяжеловесный» инструмент, классическую парадигму разработки программного обеспечения, которая управляется на основе документации. То есть, сначала создаются артефакты проектной деятельности (документация), а потом непосредственно проект (код). Классические подходы, по причине своей неуклюжести, просто не способны моментально отреагировать на резкие изменения требований заказчика. Требуется длительное время на внесение поправок в проектную и нормативную документацию, затем – в исходный код системы.

Отличие гибких подходов заключается в уходе от классической парадигмы, когда гибкая разработка программного обеспечения должна управляться изменениями и тестированием (Test Driven Development – TDD) [2], при этом документация не играет первостепенной роли, как в классических подходах. Определенные типы документации скорее вредны, чем полезны, поскольку запутывают разработчиков, отвлекают от цели, поскольку заказчик все равно внесет трудно прогнозируемые изменения в проект, уже на следующей итерации разработки. В любом случае, один «гибкий программист» пишет простое приложение быстрее, чем штат из нескольких сотрудников в классической парадигме, поскольку последние постоянно вынуждены устранять накапливающиеся противоречия. Классическая парадигма накладывает множество формальных ограничений, на процесс разработки ПО, когда сначала штат программистов будет писать детальный анализ требований, большинство из которых резко изменятся на следующей итерации разработки, потом техническое задание. На этапе разработки проекта первоначальная документация и постоянно изменяющийся по требованию заказчика функционал и код в итоге сплетутся в полный неразрешимых противоречий гордиев узел, «разрубить» который можно только одним способом – полностью убрать либо документацию, либо код из проекта.

Широко применяемой на практике, неполной моделью жизненного цикла [3], используемой в условиях рыночной экономики, значительных ограничений на время и ресурсы является итерационная модель жизненного цикла – одна из самых бескомпромиссных и жестких схем создания программного обеспечения в короткие сроки на сегодняшний день. Большинство гибких методологий, включая экстремальное программирование – XP [4], Scrum [5], TDD использует итерационную модель и т.н. “спринты” (в Scrum) для ускорения разработки ПО. Преимущества такого подхода заключаются в сокращении времени на разработку в 2-3 раза, по сравнению с более традиционными подходами, когда тратится лишнее время на создание артефактов проектной деятельности, помимо кода и программного обеспечения. Внутри фрейма текущего спринта (Scrum) может строиться диаграмма «выгорания» работ, которая показывает, сколько уже задач сделано и сколько ещё остаётся сделать до закрытия очередной итерации разработки. Диаграмма показывает нерешенные задачи и трудозатраты, необходимые для их закрытия за одну итерацию. Пример диаграммы выгорания задач в Scrum показан на рисунке 1.

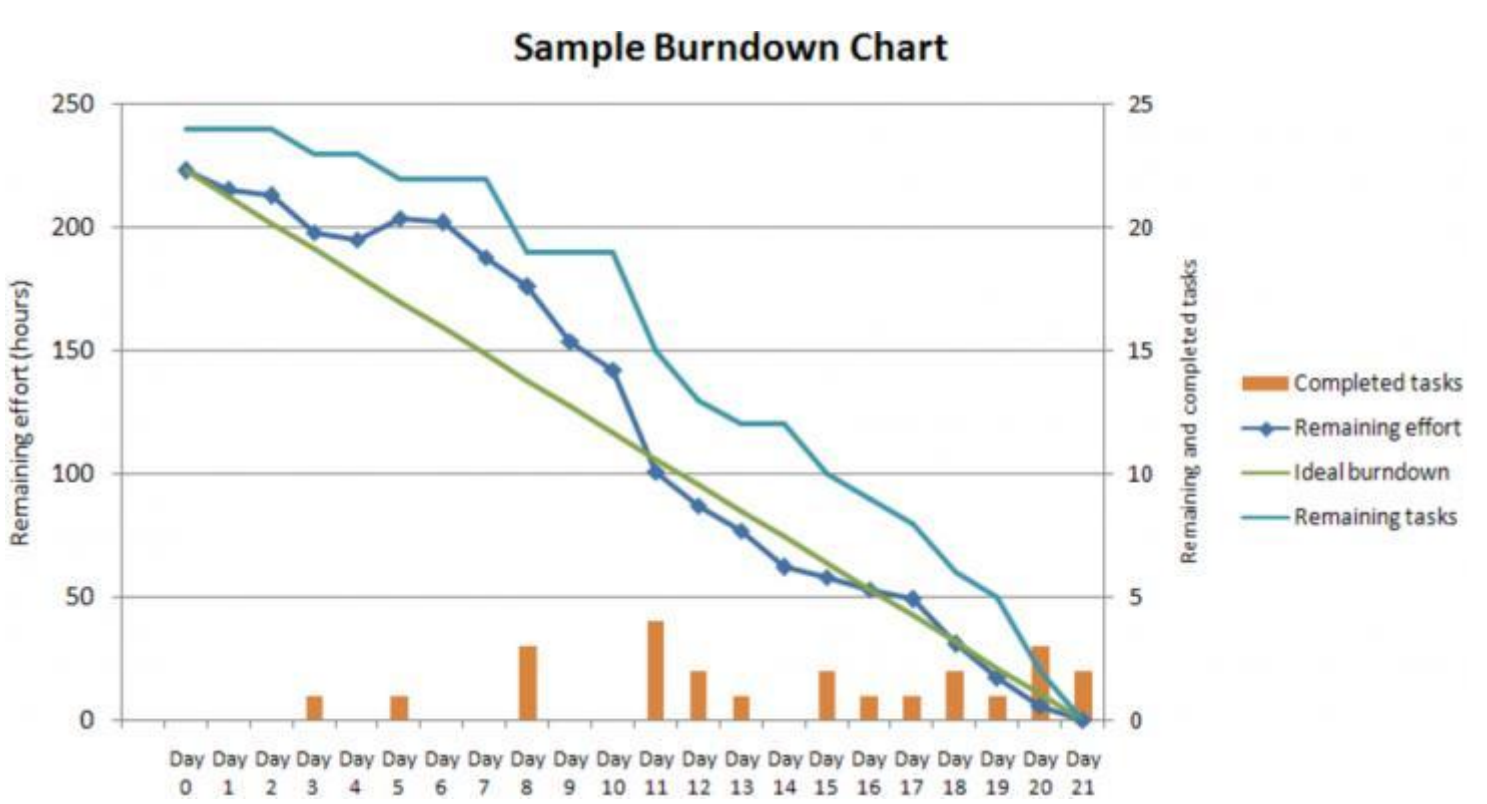

Рисунок 1. Диаграмма «выгорания» задач в спринте Scrum.

В отличие от традиционных команд, которые боятся любых изменений и тщательно их документируют, особенно на этапе ввода в эксплуатацию, гибкая команда должна приветствовать их, поскольку любые изменения в итоге позволяют получить более качественный продукт и уникальный опыт. Традиционная команда заинтересована в первую очередь в скорейшей сдаче проекта и выполнении всех формальных договоренностей, в то время как гибкие разработчики ориентированы на пользовательские истории, повышении качества продукта любой ценой и многолетнее плодотворное сотрудничество с заказчиком. Команда осуществляет поиск правильного пути движения проекта на каждой итерации. Каждая попытка в итоге оценивается и фиксируется заказчиком, который может пересмотреть свое видение итерации в любой момент, увидеть новые возможности, о которых он ранее не догадывался. Такой подход отражает совместные попытки заказчика и разработчиков двигаться в правильном направлении на каждой итерации, а также позволяет снизить риски попадания проекта в тупик в условиях, когда сам заказчик не имеет четкого представления о необходимом ему конечном результате.

Непрерывные изменения проекта в процессе работы над ним в итоге приводят к неконтролируемому, лавинообразному росту рисков, которые обычно в условиях нехватки времени никто не оценивает. Всегда существуют риски того, что ранее написанный и корректно работающий функционал, зафиксированный в текущей итерации, придется полностью или частично переписывать уже на следующей итерации разработки, поскольку заказчик изменяет свое видение конечного продукта и информирует об этом команду. Это стандартная, штатная ситуация, которую необходимо уметь предвидеть, постоянно ориентируясь на изменения и желая их. Гибкая команда разработчиков обязана уделять большое внимание вопросам построения рациональных архитектур программного обеспечения и полноценному рефакторингу кода, после каждой итерации своей работы. Под рациональной архитектурой подразумевается в первую очередь “слабо связанная” архитектура, создание и управление которой входит в компетенцию системного архитектора, работающего в русле гибких парадигм.

Представим себе ситуацию, когда некий достигаемый или уже достигнутый в определенной итерации функционал (например, запись в реляционную базу данных), подключается к системе в 10-15 различных точках, что является нормальной практикой для классической разработки ПО. Это означает, что для модификации или полного удаления функционала при наличии изменений, придется не забывать модифицировать систему в 10-12 различных местах, что приведет к усугублению проблемы на конечных этапах разработки, при существенном росте функционала и точек его подключения. Задача построения слабо связанной архитектуры при экстремальном программировании означает явное выделение отдельного пространства для будущего внедрения функционала в систему, на практике обычно используется фабрика классов (class factory) в общей структуре проекта. В идеальной, слабо связанной архитектуре программного продукта, любой достигаемый функционал должен подключатся к системе ровно один раз (что не всегда возможно), в одном и том же месте, таким образом, снижаются риски влияния резких изменений на разработку проекта. Системный архитектор определяет «точку внедрения зависимостей» (от функционала или реализации интерфейсов) в разрабатываемой системе, выбирает стратегию, а также конкретные прикладные

контейнеры разрешения зависимостей (CoDI, IoC), создает классы, фабрики классов и встраивает их в разрабатываемый продукт, которые впоследствии должны использовать все участники гибкой команды.

Практическое применение слабо связанных архитектур при разработке программного обеспечения позволяет снизить риски и сократить влияние изменений, подключая их единственный раз к определенной точке для привязки инфраструктуры. На практике, можно рекомендовать качественный инструментарий по интеграции и управлению зависимостями от различных поставщиков, включая Autofac, Unity, Castle Windsor, StructureMap, MEF и Ninject . Пример создания фабрики по разрешению зависимостей на основе паттерна MVC и распространенного DI-контейнера Ninject, с единственной точкой подключения используемой и достигаемой функциональности (метод AddBindings) приведен на рисунке 2.

```
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Ninject;
using System.Web.Routing;
using AgileTrade.Models;

namespace AgileTrade.Infrastructure
{
    public class NinjectControllerFactory : DefaultControllerFactory
    {
        private IKernel ninjectKernel;
        public NinjectControllerFactory()
        {
            ninjectKernel = new StandardKernel();      this.AddBindings();
        }

        protected override IController GetControllerInstance(RequestContext
requestContext, Type controllerType)
        {
            return controllerType == null ? null :
(IController)ninjectKernel.Get(controllerType);
        }

        public void AddBindings()
        {
            ninjectKernel.Bind<IProductViewData>().To<DbProductsRead>();
            ninjectKernel.Bind<IArticleViewData>().To<DbArticlesRead>();
            ninjectKernel.Bind<IOrderDetailData>().To<DbOrderDetailData>();
        }
    }
}
```

Рисунок 2. Пример создания точки интеграции зависимостей посредством Ninject.

Организация рабочего процесса гибких команд

Рассмотрим организацию рабочего процесса для гибкой команды. Как уже сказано выше, разработка продукта осуществляется фиксированными короткими итерациями времени (2-4 недели), под контролем представителя заказчика. Для достижения успеха, в команде должны состоять междисциплинарные специалисты и представлены, как минимум, следующие роли:

1) Бизнес-аналик (архитектор). Осуществляет общее руководство процессом разработки, представляет заказчика (бизнес) и является специалистом в области информационных технологий. Имеет четкое представление о конечных требованиях к продукту и технологиях, используемых командой для их выполнения. Осуществляет жесткий личный контроль функциональности на каждой итерации разработки по мере движения к конечному продукту. Формирует требования к проекту для системных аналитиков, а также сроки, количество и бюджет итераций. Роль изменяется от проекта к проекту.

2) Системный аналитик (архитектор). Ведущий специалист в области гибких технологий разработки ПО и вопросах модульного тестирования (Unit Tests). Взаимодействует с бизнес-аналитиком. Осуществляет координацию работы команды и глобальный рефакторинг исходного кода. Транслирует бизнес-требования и идеи в архитектуру и

программный код проекта. Разрабатывает модульные тесты для покрытия всей необходимой функциональности, которой необходимо достигнуть за одну итерацию в соответствии с бизнес-требованиями.

3) Разработчик ПО. Реализует работы по наращиванию функциональности продукта, в соответствии с задачами и требованиями, представленными системным архитектором. Обеспечивает прохождение модульных тестов, написанных системным архитектором, если используется TDD.

Предусматриваются различные способы взаимодействия в команде гибкой разработки ПО. В дополнение к основным функциям, руководство проекта выполняет построение диаграмм IDEF, UML, с целью прояснения отдельных аспектов функционирования системы. Все диаграммы создаются исключительно для разработчиков системы, поскольку другие варианты их представления вне команды утрачивают смысл. Наиболее полезными с точки зрения практики являются следующие UML диаграммы:

1) Диаграмма прецедентов (Use case). Описывает только самые общие случаи пользовательских историй. В детализации вообще нет смысла, поскольку функциональность изменится. Опытный разработчик со знанием технологий всегда имеет представление, как ему следует поступить в конкретном проекте, а типовые решения не подходят на все случаи жизни.

2) Диаграмма классов и диаграмма объектов (статический снимок проекта).

3) Диаграмма последовательности (динамический снимок передачи сообщений в группе объектов).

4) Диаграмма состояний (конечного автомата). Удобный и универсальный инструмент.

5) Другие диаграммы для специфических случаев разработки, в частности диаграмма Ганта.

Рассмотрим процедуру разработки программного обеспечения в окне одной итерации, на основе подхода, известного как TDD (разработка, посредством тестирования). Модель разработки TDD, известная в гибких технологиях как "red-green-refactor" (красное, зеленое, рефакторинг), представляет собой мощный инструмент создания качественного программного обеспечения в короткие сроки. В соответствии с требованиями бизнес-аналитика, системный архитектор создает модульные тесты (для разработчиков) вида: "в ходе проверки некоей еще не реализованной функциональности {X} разработчик должен получить результат {Y}". Системный архитектор, в итоге, стремится к созданию ПО с максимально высоким качеством и может создать для проверки одной частной функции 20-30 тестов, которые охватывают ее "с разных сторон" и дают гарантию безотказной работы в конечном промышленном продукте.

Рассмотрим конкретный пример, допустим, системный архитектор дает задание разработчику – реализовать в программном коде отыскания критического пути на графе для произвольной структуры данных. Для этого он выбирает 15 – 20 произвольных графов и пишет модульные тесты для каждого из них, зная, какой результат обязан представить разработчик в конце итерации. Разработчики в итоге должны закрыть все модульные тесты, чтобы завершить итерацию, после чего ее оценит бизнес-аналитик.

Выражаясь образным языком, подход "red-green-refactor" представляет собой следующее: одна итерация представляет собой набор перекрестков, через которые движется команда разработчиков в автомобиле. На каждом перекрестке стоит "маячок функциональности" – светофор, по сути, модульный тест, который написал системный архитектор (или сам разработчик). На светофоре зажжен красный цвет. Чтобы ехать дальше, на перекрестке должен быть зажжен зеленый цвет. Когда все перекрестки пройдены, итерация успешно заканчивается. Высокое качество конечного продукта обуславливается необходимостью полного прохождения всех тестов (которых может быть больше тысячи) к завершению проекта. Прохождение всех тестов гарантирует, что добавляемая в текущей итерации функциональность не разрушает результаты, полученные на предыдущем этапе, что постоянно случается при использовании «классических» подходов.

Механизм стимулирования гибких разработчиков

Существует множество механизмов стимулирования [6], однако их авторы, как правило, не акцентируют внимание на конкретной предметной области. В работе [7] описан механизм стимулирования на основе экономии бюджета, в работе [8] рассматривается сбалансированный (согласованный) механизм, однако представленные работы не подходят для работы гибкой команды в рамках жесткой итеративной модели жизненного цикла.

Авторы предлагают новый, не описанный ранее «тестовый» механизм стимулирования, который основан на вкладе каждого гибкого разработчика в покрытие функциональности текущей итерации. Этот механизм может использоваться гибкими командами, работающих в рамках XP, Scrum подходов, а также классическими командами, практикующими модульное тестирование при разработке проектов. Пусть Б – фонд стимулирования, предоставляемый заказчиком на проект в целом, тогда вознаграждение конкретного программиста на *i* -й итерации определяется в соответствии с выражением (1):

$$St_i^{ar} = \frac{t_i}{\sum_{i=1}^{N} t_i} \cdot Б \cdot F_i^{ar} \qquad (1)$$

где F_i^{ar} – процент выполненных тестов разработчиком аг, на *i* -й итерации выполнения проекта.

Заключение

Авторы кратко представили суть большинства современных подходов последних лет т.н. «гибкой» (agile) разработки программного обеспечения. Подробно описан итерационный рабочий процесс для XP и Scrum-команд, который использует элементы управляемой тестированием разработки - TDD (Test Driven Development), модель "red-green-refactor" и модульное тестирование. Предложен механизм стимулирования для разработчиков гибких команд, работающих по итерационной модели жизненного цикла.

Библиография

1. Рассел Дж. Жизненный цикл программного обеспечения / Дж. Рассел // Bookvika Publishing, 2012. 89 с.

2. Beck Kent. Test-Driven Development By Example. Addison-Wesley Professional, 2002. — 240 p. — ISBN-10: 0321146530 ISBN-13: 978-0321146533.

3. Добрынин А. С. Модель неполного жизненного цикла программного обеспечения / А. С. Добрынин, Р. С. Койнов, С. М. Кулаков // Вестник АГТУ. Серия: Управление, вычислительная техника и информатика. – 2015. – № 2.-С. 65-69. – Библиогр.: с. 69 (9 назв.). – Режим доступа: http://library.sibsiu.ru.

4. Мартин Р. С. Agile software development / Р. С. Мартин, Дж. В. Ньюкирк, Р. С. Косс // Principles, Patterns and Practicies Principles, Patterns and Practicies. М.: Вильямс, 2004. 752 с.

5. Cohn Mike. Scrum: Succeedind with Agile: Software Development Using Scrum (Addison-Wesley Signature Series). М.: Вильямс, 2011. 576 с. URL: http://www.jot.fm/books/review7.pdf.

6. Бурков В.Н., Коргин Н.А., Новиков Д.А. Введение в теорию управления организационными системами / Под ред. чл.-корр. РАН Д.А. Новикова. – М.: Либроком, 2009. – 264 с.

7. О механизме стимулирования разработчиков ИТ-систем / А. С. Добрынин, Р. С. Койнов, А. В. Грачев, В. В. Митьков // Экономика, статистика и информатика. Вестник УМО. – 2015. – № 5.-С. 164-167. – Библиогр.: с. 167 (6 назв.). – Режим доступа: http://elibrary.ru.

8. Зимин В.В. О сбалансированном стимулировании разработчиков ИТ-сервисов / В.В. Зимин, С.М. Кулаков, А.В. Зимин // Системы управления и информационные технологии, №3(49), 2012.-С. 73-76

References (transliterated)

1. Rassel Dzh. Zhiznennyi tsikl programmnogo obespecheniya / Dzh. Rassel // Bookvika Publishing, 2012. 89 s.

2. Beck Kent. Test-Driven Development By Example. Addison-Wesley Professional, 2002. — 240 p. — ISBN-10: 0321146530 ISBN-13: 978-0321146533.

3. Dobrynin A. S. Model' nepolnogo zhiznennogo tsikla programmnogo obespecheniya / A. S. Dobrynin, R. S. Koinov, S. M. Kulakov // Vestnik AGTU. Seriya: Upravlenie, vychislitel'naya tekhnika i informatika. – 2015. – № 2.-S. 65-69. – Bibliogr.: s. 69 (9 nazv.). – Rezhim dostupa: http://library.sibsiu.ru.

4. Martin R. S. Agile software development / R. S. Martin, Dzh. V. N'yukirk, R. S. Koss // Principles, Patterns and Practicies Principles, Patterns and Practicies. M.: Vil'yams, 2004. 752 s.

5. Cohn Mike. Scrum: Succeedind with Agile: Software Development Using Scrum (Addison-Wesley Signature Series). M.: Vil'yams, 2011. 576 s. URL: http://www.jot.fm/books/review7.pdf.

6. Burkov V.N., Korgin N.A., Novikov D.A. Vvedenie v teoriyu upravleniya organizatsionnymi sistemami / Pod red. chl.-korr. RAN D.A. Novikova. – M.: Librokom, 2009. – 264 s.

7. O mekhanizme stimulirovaniya razrabotchikov IT-sistem / A. S. Dobrynin, R. S. Koinov, A. V. Grachev, V. V. Mit'kov // Ekonomika, statistika i informatika. Vestnik UMO. – 2015. – № 5.-S. 164-167. – Bibliogr.: s. 167 (6 nazv.). – Rezhim dostupa: http://elibrary.ru.

8. Zimin V.V. O sbalansirovannom stimulirovanii razrabotchikov IT-servisov / V.V. Zimin, S.M. Kulakov, A.V. Zimin // Sistemy upravleniya i informatsionnye tekhnologii, №3(49), 2012.-S. 73-76