УДК 004.4'2

Ю.А. Лапшов

АВТОМАТИЗАЦИЯ УПРАВЛЕНИЯ ИНСПЕКЦИЕЙ КОДА ПРОГРАММНЫХ ПРОТОТИПОВ НА ОСНОВЕ ВОПРОСНО-ОТВЕТНЫХ ПРОТОКОЛОВ АНАЛИЗА ТРЕБОВАНИЙ¹

Лапшов Юрий Александрович, кандидат технических наук, окончил Ульяновский государственный технический университет. Доцент кафедры «Вычислительная техника» УлГТУ. Имеет статьи, монографию, свидетельства о регистрации программы для ЭВМ. Область научных интересов: системы автоматизированного проектирования, управление потоками проектных работ. [e-mail: y.lapshov@ulstu.ru].

Аннотация

Данная работа представляет собой описание архитектуры программных средств автоматизации управления инспекцией кода программных прототипов проектных решений. Эти средства позволяют получить такие эффекты, как обеспечение интерактивного взаимодействия между проектировщиком и экспертом, выполняющим инспекцию кода, а также сокращение временных затрат эксперта на комментирование кода. Комментирование осуществляется путем выбора комментария из списка заранее подготовленных для каждого требования из полученных в ходе анализа требований. Разрабатываемые средства состоят из трех основных частей. Первая часть представляет требования в вопросно-ответном формате, включающие стандартные комментарии для вставки в код прототипа в случае несоответствия данным требованиям. Вторая часть представляет реляционную базу данных, которая предназначена для хранения исходных кодов прототипов, направленных на инспекцию, и прототипов, ее прошедших, включающих комментарии эксперта. Третья является Web-приложением, позволяющим обеспечить проектировщику направление кодов прототипов на инспекцию и ознакомление с результатами инспекции, а эксперту – обеспечить просмотр и редактирование исходного кода прототипов со вставкой как стандартных, заранее подготовленных комментариев, так и написанных в свободной форме.

Ключевые слова: прототипирование, инспекция кода, анализ требований, вопросно-ответные модели.

doi: 10.35752/1991-2927-2021-1-63-119-129

AUTOMATED CONTROL OF THE SOFTWARE PROTOTYPE CODE REVIEW BASED ON THE QUESTION-ANSWERING PROTOCOLS FOR REQUIREMENT ANALYSIS

Iurii Aleksandrovich Lapshov, Candidate of Sciences in Engineering; graduated from Ulyanovsk State Technical University; Associate Professor of the Computer Engineering Department at UlSTU; an author of articles, a monograph; a holder of certificates of software registration; the area of his scientific interests relates to computer-aided design systems, workflow management. e-mail: y.lapshov@ulstu.ru.

Abstract

The paper describes the architecture of software tools for automating the management of code review of software prototypes of design solutions that allow to obtain such effects as ensuring interactive interaction between the designer and an expert performing code review, as well as reducing the expert's time spent on commenting the code by selecting a comment from the list prepared in advance for each requirement from the requirements obtained during the analysis. The tools being developed consist of three main parts. The first part presents requirements in a question-and-answer format including standard comments to be inserted into the prototype code in case of non-compliance with these requirements. The second one is a relational database, which is designed to store the source codes of prototypes aimed at inspection and passed it with expert comments. The third one is a Web-application that allows the designer to send prototype codes for review and see the results of the inspection, and an expert to provide viewing and editing of the prototype source code with the insertion of both standard, pre-prepared comments, and written in free form.

Keywords: prototyping, code review, requirements analysis, question-answer models.

¹ Исследование выполнено при финансовой поддержке РФФИ в рамках научных проектов 18-47-730016 p_a и 18-07-00989 A.

ВВЕДЕНИЕ

В настоящее время одной из основных проблем разработки сложных программных систем, к классу которых относятся и автоматизированные системы, является низкая степень успешности проектов [1]. Среди путей повышения успешности проектирования можно выделить прототипирование на всех этапах проектирования, при этом одной из полезных, и при этом, легко внедряемых практик, позволяющих повысить качество прототипов, является практика инспекции кода (Code review). Данная практика позволяет повысить качество кода и определить степень соответствия прототипа требованиям, определенным на этапе их анализа.

В процессе инспекции кода принимают участие как минимум два актора – проектировщик, являющийся автором кода прототипа, и эксперт, рецензирующий код и принимающий решение о его готовности к включению в общую базу кода проекта. При этом прототипирование, а следовательно, и инспекция кода, возможны и на этапе анализа требований, когда требования к проекту не являются полностью сформированными. Сам же анализ требований является итеративным процессом, в ходе которого требования дополняются и уточняются. Таким образом, эксперты, выполняющие инспекцию кода, должны иметь доступ к изменяющимся требованиям в реальном времени.

В случае обнаружения несоответствий исходных кодов прототипов требованиям, эксперт может прокомментировать исходный код, отметив несоответствие и, возможно, пути устранения этих несоответствий. Поскольку требования зачастую являются общими для многих прототипов проекта и даже для различных проектов, то полезной возможностью, оптимизирующей работу эксперта, будет выбор комментариев из набора стандартных комментариев, соответствующих определенному требованию, и автоматическая вставка этого комментария в исходный код прототипа. К таким комментариям относятся и вердикты, вынесенные в результате инспекции кода. Обеспечить реализацию этой возможности, а также возможности интерактивного взаимодействия эксперта с проектировщиком можно путем реализации программных средств автоматизации управления инспекцией кода прототипов.

Поскольку структура требований является иерархической и представленной на естественном языке, то эффективным форматом представления требований, сформированных на этапе анализа требований к коду программы, является вопросно-ответное (QA) моделирование [2]. Сама специфика представления кода в формате вопросов и ответов на них позволяет четко структурировать требования к коду в соответствии с иерархической структурой вопросно-ответной памяти.

В ходе работы был рассмотрен ряд публикаций по тематике автоматизации управления инспекцией кода. В своей работе [3] автор указывает, что инспекция кода это систематическая проверка исходного кода программы, предназначенная для поиска ошибок, упущенных

на начальном этапе разработки, улучшения общего качества проекта и снижения риска ошибок. Также он дает обзор современных инструментов проверки кода. В статье [4] авторы отметили, что инспекция кода играет важную и успешную роль в разработке современного программного обеспечения (ПО). Обычно она выполняется только один раз, прежде чем новый код будет объединен с основной веткой. Авторы представляют концепцию, которая помогает разработчикам постоянно давать отзывы об исходном коде непосредственно в интегрированной среде разработки (IDE), используя метафору социальных сетей. Авторы работы [5] выделяют ряд факторов, характеризующих степень негативно-го отношения разработчика к результатам Code Review, и предлагают пути минимизации данных эффектов. В работе [6] авторы рассматривают проблему инспекции кода с образовательной позиции и предлагают варианты обучения специалистов инспекции кода в игровой форме. Авторы статьи [7] отмечают особенности автоматизированного выбора наиболее подходящего эксперта для проведения инспекции кода на основе средств обработки естественного языка. Все эти работы описывают современное состояние исследований в области инспекции кода и применяемый инструментарий. Эти публикации не содержат описания механизмов, обеспечивающих эффективный способ представления результатов анализа требований для последующего применения их в целях автоматизированного комментирования кода прототипов экспертом. Данная работа посвящена реализации такого механизма.

Для работы с вопросно-ответной памятью существуют мощные средства, реализованные в составе вопросно-ответного процессора *WIQA (Working in Questions and Answers)*, разработанного на кафедре «Вычислительная техника» УлГТУ. Данные средства позволяют не только хранить данные в вопросно-ответном формате, но также и обеспечивают возможность программировать обработку этих данных на псевдокодом языке программирования *L^{WIQA}* [8]. Эти возможности позволяют отображать на вопросно-ответную память *WIQA* функциональные и параметрические требования к исходному коду, сгруппированные в виде набора списков требований, отображаемых эксперту в соответствии с семантикой прототипа, над которым выполняется его работа. При этом аналитик, не изменяя требований, может, с одной стороны, оставлять комментарии в вопросно-ответных списках требований, отмечая степень соответствия им реализованной в прототипе функциональности. Более того, прототипов может быть прислано на инспекцию более одного, и вопросно-ответная структура списков требований позволяет отмечать степень соответствия каждого из прототипов и возможности их доработки.

Данная статья предлагает формат представления требований системных аналитиков в вопросно-ответной памяти, механизм взаимодействия аналитика и проектировщика, реализовавшего прототип, а также

описание архитектуры компонента среды *OwnWIQA* [9], обеспечивающего функциональность инспекции кода прототипа с доступом к данным, получаемым в ходе анализа требований.

1 СПЕЦИФИКАЦИЯ ТРЕБОВАНИЙ НА ОСНОВЕ QA-ПРОТОКОЛА И ОРГАНИЗАЦИЯ ИНТЕРФЕЙСА ВЗАИМОДЕЙСТВИЯ С ВОПРОСНО-ОТВЕТНОЙ ПАМЯТЬЮ

Стандартный глоссарий терминологии программной инженерии IEEE (IEEE Standard Glossary of Software Engineering Terminology) [10] определяет требования как:

1. Условия или возможности, необходимые пользователю для решения проблем или достижения целей;
2. Условия или возможности, которыми должна обладать система или системные компоненты, чтобы выполнить контракт или удовлетворять стандартам, спецификациям или другим формальным документам;
3. Документированное представление условий или возможностей для пунктов 1 и 2.

Таким образом, требования являются спецификацией того, что должно быть реализовано.

По Карлу Вигерсу [11], требования разделяются на функциональные и нефункциональные. К функциональным относятся бизнес-требования, определяющие высокоуровневые цели заказчика, а также требования к функциональности, задающие функциональность *ПО*, реализуемую разработчиками для того, чтобы пользователи смогли выполнять задачи в рамках бизнес-требований. Поскольку суть Code Review направлена на рецензирование исходного кода, то рамки данной работы включают требования к функциональности, которые может проверить аналитик, изучая исходные коды прототипов.

Требования к функциональности, которые можно использовать в процессе инспекции кода прототипов, можно разделить на две основные части – общие для всего проекта, всех его компонентов и частные, являющиеся специфическими для каждого отдельного компонента проекта. К общим требованиям относятся такие, как требования к точности обработки данных, требования к размерным параметрам (лимитами) значений данных, ограничения на используемые библиотеки и фреймворки. Частные требования могут включать уточнения общих, а также учитывающие специфику разрабатываемого программного компонента проекта. Списки требований являются открытыми и не могут быть ограничены заранее специфицированными их наборами, так как должны учитывать специфику проекта. Таким образом, при разработке формата представления наборов требований необходимо учитывать открытость списков требований и возможность их расширения.

Как было сказано во введении, одним из наиболее эффективных форматов представления результатов анализа требований является вопросно-ответное представление. Ключевыми элементами вопросно-ответной памяти является задача (*Z*), вопрос (*Q*) и ответ

(*A*). Задача может включать в себя подзадачи и вопросы. Вопросы также могут включать в себя подвопросы и ответы. Таким образом формируется иерархическая структура [12].

Иерархическую структуру требований к функциональности ПО в вопросно-ответной памяти логично начать с узла-задачи *Zf* «Функциональные требования». В этой задаче имеет смысл выделить две подзадачи *Zfa* «Общие требования» и *Zfb* «Требования к отдельным функциональным элементам». Задача *Zfa* должна содержать подзадачи *Zfa₁...Zfa_N*, описывающие категории общих требований, где *N* – количество категорий общих требований к разрабатываемым программным средствам. Каждая из этих задач *Zfa_i* должна включать в себя список требований в виде набора вопросов *Qfa_{i1}...Qfa_{iM}*, где *M* – количество требований в категории. Поскольку вопросы могут включать в себя подвопросы, то атрибуты каждого из требований можно представлять в виде подвопросов.

Таким образом, на место первого подвопроса вопроса *Qfa_{ij}* ставится вопрос о том, какие из представленных прототипов удовлетворяют данному требованию. Во вложенных в него вопросах перечисляются прототипы, а ответом на каждый вопрос является степень его соответствия требованию в виде текстового описания степени соответствия на естественном языке. Вторым подвопросом вопроса *Qfa_{ij}* ставится вопрос о стандартном текстовом комментарии, который можно вставить в inspectируемый прототип в том случае, если его код не соответствует данному требованию. В том случае, если требование не предполагает стандартного комментария, данный вопрос в дереве вопросно-ответного протокола можно оставить без ответа. Задача *Zfb* «Требования к отдельным функциональным элементам» на первом уровне включает в себя список подзадач *Zfb₁...Zfb_K*, где каждая из этих подзадач описывает тот компонент, к которому относятся данные требования. При этом подзадачи задач *Zfb_i* будут аналогичны подзадачам задачи *Zfa* с аналогичной структурой вложенных подзадач, вопросов и ответов. Также в задачу *Zfa* включается подзадача, описывающая общие для всех проектов требования, содержащие стандартные текстовые комментарии, которые могут пригодиться в инспекции кода прототипа любого проекта, такие как: «Риск нарушения требований по точности», «Риск нарушения требований ко времени обработки», «Риск нарушения лицензионных соглашений», «Риск нарушения требований к представлению данных», а также с репликами на результат инспекции «Прототип соответствует требованиям», «Подготовить функциональность для согласования с заказчиком». Данный список стандартных текстовых комментариев, общий для всех проектов, не является исчерпывающим и может быть расширен добавлением *QA*-единиц в дерево вопросно-ответного

протокола. Полезной особенностью среды **WIQA** является возможность подготовки шаблонов-фрагментов вопросно-ответного дерева, что позволяет сохранить задачу **Zf** в качестве шаблона с набором общих для всех проектов требований и использовать ее для различных проектов.

Вопросно-ответное представление списка требований позволяет, с одной стороны, просматривать их эксперту, занимающемуся инспекцией кода, в древовидном формате в интерфейсе вопросно-ответного протокола среды **OwnWIQA** [13], а с другой стороны обрабатывать их с помощью языка **L^{WIQA}** [9,10] для дальнейшей передачи в подсистему инспекции кода, являющейся редактором, позволяющим как просматривать код прототипа, так и комментировать его вручную и с использованием заранее заготовленных комментариев, соответствующих каждому из требований, формирующихся в процессе анализа требований.

2 ФОРМАЛИЗАЦИЯ ПРЕДСТАВЛЕНИЯ ДАННЫХ, С КОТОРЫМИ ОСУЩЕСТВЛЯЕТСЯ РАБОТА СРЕДСТВ УПРАВЛЕНИЯ АВТОМАТИЗАЦИЕЙ ИНСПЕКЦИИ КОДА ПРОТОТИПОВ

В качестве инструмента формализации здесь будет использована расширенная Бэкус-Наурова форма (РБНФ). Перед тем, как построить отображение проектных требований на память **WIQA**, сформулируем в виде РБНФ саму память, ограничившись теми ее атрибутами, которые востребованы в данной работе. Вопросно-ответная память **WIQA** представляет собой иерархически организованное хранилище **QA**-объектов:

QA-память = {**QA-объект**}.

Причем сам **QA**-объект представляет собой пару, состоящую из вопроса и ответа. Причем ответ может быть пустым или отсутствовать:

QA-объект = **Задача** | **Вопрос**, «←», **Ответ**.

Здесь и далее нетерминал «←» означает вхождение (соответствие), а нетерминал «↓» – иерархическое подчинение. Мы можем ввести эти нетерминалы, поскольку сама память **WIQA** предполагает такие отношения.

Как вопрос может содержать в себе подвопросы, так и ответ может содержать иерархически подчиненные ему ответы. Задача может содержать в себе как подзадачи, так и вопросы:

Задача = **Z** | (**Z**, «↓», {**Z**}) | (**Z**, «↓», {**Q**});

Вопрос = **Q** | (**Q**, «↓», {**Q**});

Ответ = **A** | (**A**, «↓», {**A**}).

Как вопрос, так и ответ, являются узлами вопросно-ответного дерева. Каждый узел дерева имеет определенные базовые атрибуты **g**, такие как: дата создания, текстовое описание, индекс, статус и некоторые другие. Кроме основных, каждый узел этого дерева

может содержать в себе произвольный набор дополнительных атрибутов **ag**. Кроме основных и дополнительных атрибутов, каждый узел дерева может содержать ряд прикрепленных к нему файлов:

Z = ({**g**}, {{**ag**}}, {{**файл**}});

Q = ({**g**}, {{**ag**}}, {{**файл**}});

A = ({**g**}, {{**ag**}}, {{**файл**}});

g = (**дата**, **описание**, **индекс**, ..., **статус**).

Одним из атрибутов **g** является описание – текстовая строка в кодировке **UTF-8**, в среде вопросно-ответного процессора **OwnWIQA** имеет ограничение 4 килобайта, при этом в редакторе вопросно-ответного протокола это поле отображается пользователю. Таким образом, атрибут «описание» является наиболее подходящим для хранения содержательной части требования или комментария.

Таким образом, задача «Функциональные требования» отображается на вопросно-ответную память следующим образом:

Zf = (**Z**, «↓», (**Zfa**, **Zfb**)).

Задача **Zfa** содержит набор категорий функциональных требований ко всем прототипам проекта в виде задач:

Zfa = (**Z**, «↓», {**Zfat**}).

Каждая из этих задач включает в себя набор вопросов **Qfa**, каждый из которых соответствует определенному функциональному требованию, общему для всех прототипов.

Zfat = (**Z**, «↓», {**Qfa**});

Qfa = (**Q**, «↓», (**Qp**, **Qc**)).

Здесь **Qp** – вопрос о том, какие прототипы удовлетворяют требованию, а **Qc** – вопрос о стандартном комментарии, который может быть вставлен в прототип в том случае, если этот прототип не удовлетворяет требованию. Эти вопросы предполагают наличие ответов:

Qp = (**Q**, «↓», **Ap**);

Qc = (**Q**, «↓», {**Ac**}).

Здесь **Ap** – ответ на вопрос о прототипах, удовлетворяющих требованиям. Описание его выглядит следующим образом:

Описание_Ap = **Имя_прототипа** | **Имя_прототипа**{**Имя_Прототипа**}.

Ac – ответ на вопрос о комментарии. Его описание содержит текст стандартного комментария.

Задача **Zfb** содержит набор подзадач, описывающих требования к компонентам проекта.

Zfb = (**Z**, «↓», {**Zfc**}).

Каждая задача Zfc содержит описание набора требований к определенному компоненту проекта. В описании, содержащемся в этой задаче, представляется название компонента.

$$Zfc = (Z, \langle \downarrow \rangle, \{Qfb\});$$

$$Qfb = (Q, \langle \downarrow \rangle, (Qe, Qc)).$$

Здесь Qe – вопрос о том, требования к какому компоненту проекта представлены в этом узле вопросно-ответного дерева, а Qc – вопрос о стандартном комментарии, который может быть вставлен в прототип в том случае, если этот прототип не удовлетворяет требованию:

$$Qe = (Q, \langle \downarrow \rangle, Ap);$$

$$Qc = (Q, \langle \downarrow \rangle, \{Ac\}).$$

Эти вопросы также предполагают наличие ответов:

$$Qe = (Q, \langle \downarrow \rangle, Ae);$$

$$Qc = (Q, \langle \downarrow \rangle, \{Ac\}).$$

Здесь Ae – ответ на вопрос о компоненте проекта. Его описание содержит текст названия компонента, Ac – ответ на вопрос о комментарии, описание которого содержит текст стандартного комментария.

На рисунке 1 представлена онтологическая модель средств автоматизации управления инспекцией кода,

на которой также представлены акторы, взаимодействующие с данными средствами.

3 БАЗОВЫЕ СЦЕНАРИИ ДЕЯТЕЛЬНОСТИ ИНСПЕКТОРОВ КОДА ПРОТОТИПА

В процессе проектирования к инспекции кода имеют непосредственное отношение три роли: аналитик требований, поставляющий данные для вопросно-ответного протокола анализа требований, разработчик кода прототипа и эксперт, осуществляющий инспекцию кода.

Разработчик прототипа, получив набор требований, выполняет написание кода прототипа и отправляет его на инспекцию. Эксперт, получая требования разработчика и код прототипа, получает возможность просматривать требования, вставлять в код как комментарии, соответствующие требованиям из списка, так и добавлять в код собственные комментарии, в том числе и с результатом инспекции. После этого эксперт сохраняет комментированный код в базу данных (БД), и разработчик получает возможность ознакомиться с комментариями, и, при необходимости, доработать код прототипа. Также, после выполнения инспекции, эксперт отмечает в вопросно-ответном дереве требований степень соответствия прототипа предъявляемым требованиям. В процессе комментирования кода, отправленного на инспекцию, эксперт может как написать комментарий в свободной форме, так и выбрать из списка, соответ-

Рис. 1. Онтологическая модель базы знаний разрабатываемых средств с акторами

Рис. 2. Варианты использования средств инспекции кода

ствующего стандартным комментариям, размещенным в вопросно-ответном протоколе.

На рисунке 2 представлена диаграмма вариантов использования средств инспекции кода. Данная диаграмма имеет три актора – следовательно, комплекс средств инспекции кода должен содержать три варианта пользовательских интерфейсов:

- интерфейс аналитика;
- интерфейс проектировщика;
- интерфейс эксперта.

Та часть пользовательских интерфейсов, непосредственно затрагивающих вопросно-ответный протокол, в которой выполняются действия «Построение списков требований», «Просмотр списков требований», «Отметка степени соответствия прототипов требованиям», не требует реализации дополнительных программных средств, так как в составе вопросно-ответной среды **OwnWIQA** имеется редактор вопросно-ответного протокола, а также механизмы импорта-экспорта фрагментов дерева вопросно-ответного протокола в **XML**-формате для синхронизации хранящихся в нём данных между участниками проектного процесса.

Действие «Написание кода прототипа» также не требует реализации программных средств, поскольку проектировщик использует для этого требуемые проектом средства разработки и языки программирования, которые могут различаться как внутри одного проекта, так и в различных проектах.

Действия «Отправка прототипа на инспекцию кода», «Просмотр кода, направленного на инспекцию», «Ком-

ментирование кода» требуют разработки визуального редактора, обеспечивающего как взаимодействие между проектировщиком и экспертом, так и возможность комментирования кода. Наиболее подходящим вариантом реализации данного редактора является веб-интерфейс, поскольку он обеспечивает интерактивность. Кроме того, требуются механизмы, обеспечивающие взаимодействие между веб-интерфейсом и вопросно-ответным протоколом. О том, как можно реализовать данные программные средства, пойдет речь в следующем пункте.

4 ПРЕДСТАВЛЕНИЕ ПРОГРАММНЫХ ПРОТОТИПОВ В КОМПЛЕКСЕ СРЕДСТВ УПРАВЛЕНИЯ АВТОМАТИЗАЦИЕЙ ИНСПЕКЦИИ КОДА ПРОТОТИПОВ

На рисунке 3 схематично представлена минимальная архитектура разрабатываемых программных средств инспекции кода.

Потоки данных, возникающие при использовании средств автоматизации инспекции кода, представлены на рисунке 4.

Рассмотрим более подробно представленные на рисунке 3 компоненты. Дерево требований (Задача **Zf**) представлено в вопросно-ответной памяти **WIQA**.

Компонент «БД требований» представляет собой реляционную БД, являющуюся промежуточным форматом, обеспечивающим взаимодействие между вопросно-ответным протоколом и Web-редактором исходного кода.

Рис. 3. Схема архитектуры средств инспекции программного кода

Рис. 4. Потоки данных инспекции программного кода

БД требований должна содержать в себе, как минимум, следующие информационные компоненты:

- Коды прототипов, отправляемые проектировщиком;
- Списки требований и комментарии к этим требованиям.

На рисунке 5 представлена ER-диаграмма БД.

Данные требований из вопросно-ответного протокола (Задача Zf) возможно [14] экспортировать в реляционную БД с помощью программы, реализованной на языке L^{WQA} . На вход данная программа должна получать задачу Zf , представленную в вопросно-ответном формате, а на выходе – генерировать последовательность команд на языке SQL для формирования таблиц «Требования» и «Стандартные комментарии». Этот скрипт должен выполнять аналитик после завершения формирования требований в вопросно-ответном протоколе.

Рис. 5. Диаграмма реляционной БД средств автоматизации инспекции кода

Как видно из рисунка 5, между таблицами «Разработчики», «Исходные коды» и таблицами «Требования» и «Стандартные комментарии».

бования», «Стандартные комментарии» отсутствуют отношения. Связи между этими таблицами должен обеспечивать Web-редактор исходного кода. Страница добавления прототипа должна обеспечивать функциональность загрузки файла исходного кода прототипа в БД с указанием названия прототипа и последующим добавлением его в таблицу «Исходные коды». Страница выбора прототипа для просмотра должна отображать содержащийся в таблице «Исходные коды» список прототипов для последующего их открытия для просмотра и редактирования. Также, в случае реализации подсистемы авторизации, может отображаться только список прототипов, доступных конкретному проектировщику или эксперту. После выбора прототипа из перечня должна открываться следующая страница – страница просмотра и редактирования кода прототипа.

Страница просмотра и редактирования кода прототипа обеспечивает ключевую функциональность инспекции кода для эксперта. Данная страница должна содержать как минимум следующие элементы:

- Поле для редактирования и просмотра исходного кода;
- Поле, отображающее списки требований и стандартные комментарии для них;
- Кнопка для вставки стандартного комментария, выбранного в списке;
- Кнопка для сохранения отредактированного прототипа.

5 ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ

В рамках данного исследования была разработана макетная версия системы автоматизации управления инспекцией кода, основанная на программных средствах, разработанных на кафедре «Вычислительная техника» УлГТУ, таких как вопросно-ответный процессор *OwnWIQA* и реализованный в рамках средств поддержки проектного обучения Web-редактор исходного кода. В макетную версию также включен шаблон во-

просно-ответного дерева с задачей «Функциональные требования». Разработка макетной версии позволила осуществить эксперимент, подтверждающий гипотезу, формулируемую следующим образом: средства автоматизации управления инспекцией кода позволяют повысить как скорость процесса инспектирования кода, так и снизить количество ошибок в процессе инспектирования. Целью проведения эксперимента является проверка данной гипотезы. План эксперимента следующий:

1. Осуществить имитацию работы аналитика путем формирования задачи функциональных требований в среде **OwnWIQA** с примерами стандартных комментариев.
2. Осуществить имитацию работы разработчика путем добавления образца прототипа в Web-инструментарий комментирования кода.
3. Осуществить имитацию работы эксперта путем просмотра кода прототипа и добавления в него комментариев как автоматически из стандартных комментариев, соответствующих требованиям, так и вручную, с последующим сохранением кода прототипа.
4. Осуществить имитацию работы разработчика путем просмотра кода прототипа с комментариями.
5. Оценить эффекты от применения разработанных программных средств с позиции повышения скорости процесса инспектирования кода и снижения количества ошибок.

Ход выполнения эксперимента:

1. Выполняя роль «Аналитик», в среде *WIQA* был создан проект и загружен стандартный шаблон. Аналитик на данном этапе дополняет шаблон требованиями. Затем была выполнена задача «Экспорт в БД Web-системы», что представлено на рисунке 6.
2. Выполняя роль «Разработчик», в Web-систему был добавлен пример исходного кода и сохранен в БД. В верхней части окна уже отобразились данные, экспортированные из вопросно-ответного дерева (рис. 7).

Рис. 6. Эксперимент, пункт 1. Формирование задачи функциональных требований

3. Выполняя роль «Эксперт», был открыт сохраненный прототип, установлен курсор в коде на требуемое место, выбран стандартный комментарий (стрелка 1) и вставлен путём нажатия кнопки «Вставить». Комментарий появился в коде (стрелка 2). Также был добавлен обычный комментарий, соответствующий синтаксису языка C/C++, отличающийся подсветкой

кода (стрелка 3) на рисунке 8. Далее код прототипа был сохранен.

4. Выполняя роль разработчика, был открыт исходный прототип. Все комментарии в нем сохранены (рис. 9).

5. Выполнение указанных действий показало, что эксперту не потребовалось вводить с клавиатуры комментарии, заранее определенные как стандартные для

Рис. 7. Эксперимент, пункт 2. Добавление исходного кода

Рис. 8. Эксперимент, пункт 3. Вставка комментариев

Рис. 9. Эксперимент, пункт 4. Просмотр кода с комментариями

данного прототипа. Это позволило исключить как временные затраты, связанные с вводом комментария, так и вероятность ошибок при вводе комментария, соответствующего данному требованию. При этом эксперт не потерял возможности вставки комментариев путем ввода их с клавиатуры в свободной форме.

Эксперимент показал работоспособность макетной версии средств автоматизации управления инспекцией кода прототипов, произошла вставка комментария из БД, созданной с использованием вопросно-ответной среды *OwnWQA*. Таким образом, можно говорить о том, что эксперимент подтвердил гипотезу об эффективности разрабатываемых средств

ЗАКЛЮЧЕНИЕ

Данная статья описывает принципы построения системы автоматизированного управления инспекцией кода, которая позволяет получить следующие положительные эффекты от ее применения:

- Представление списков требований, полученных на этапе анализа требований, в вопросно-ответном формате позволяет четко их структурировать и осуществить обмен этими требованиями между аналитиком, проектировщиком и экспертом, выполняющим инспекцию кода;
- Использование Web-интерфейса и реляционной БД прототипов позволяет осуществить интерактивное взаимодействие между разработчиком и экспертом;
- Эксперт получает возможность вставлять стандартные для данного требования замечания в виде комментариев в исходный код прототипа без необходимости ввода его каждый раз вручную. К тому же, ряд комментариев может быть общим для любого прототипа проекта или ряда проектов, такой как результат инспекции. Тем самым достигается сокращение временных затрат эксперта и сокращение риска допущения им ошибок в комментариях;
- Визуальный редактор, используемый для просмотра и редактирования кода прототипа, позволит зрительно выделить комментарии проектировщика, тем самым переключив на них фокус внимания сразу после открытия проинспектированного кода.

Представленная в работе архитектура программных средств является минимальной и может быть расширена в процессе их реализации, которая будет выполнена в ходе дальнейшей работы по данной тематике.

В качестве дальнейших путей расширения разрабатываемых средств можно отметить следующие:

- Реализация контроля доступа, механизмов аутентификации и авторизации пользователей;
- Реализация средства комментирования в виде дополнения к таким существующим системам, используемым в практике Code Review, как GitHub;
- Реализация иерархической структуры групп требований, позволяющих группировать их на более чем одном уровне вложенности.

СПИСОК ЛИТЕРАТУРЫ

1. The Standish Group report 83.9% of IT projects partially or completely fail. URL: <https://www.opendoorerp.com/the-standish-group-report-83-9-of-it-projects-partially-or-completely-fail/> (дата обращения: 24.10.2020).
2. Маклаев В.А., Соснин П.И. Создание и использование автоматизированной базы опыта проектной организации. Ульяновск : УлГТУ, 2012. 360 с.
3. Khleel Nasraldeen, Nehéz Károly. Tools, processes and factors influencing of code review // Multidiszciplináris tudományok. 2020. 10. kötet. 3 sz. pp. 277–284.
4. Dürschmid Tobias. Continuous Code Reviews – A Social Coding tool for Code Reviews inside the IDE // First International Conference on the Art, Science, and Engineering of Programming, Association for Computing Machinery. New York City, NY, United States. 2017. Article № 41. pp. 1–3.
5. Predicting Developers' Negative Feelings about Code Review / Carolyn D. Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Maggie Morrow Hodges, Collin Green, Ciera Jaspán, James Lin // International Conference on Software Engineering (2020). URL: <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/7ac08fa960dfe10561c1f5953419a0c945279faa.pdf> (дата обращения: 29.11.2020).
6. Ardic B., Yurdakul I., Tüzün E. Creation of a Serious Game for Teaching Code Review: An Experience Report // 2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T), Munich, Germany, 2020. pp. 1–5.
7. Sadman N., Ahsan Md M., Parvez M.A. ADCR: An Adaptive Tool to select «Appropriate Developer for Code Review» based on Code Context // Proc. of the 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conf. (UEMCON), New York City, NY, 2020. pp. 0583–0591.
8. Sosnin P. Automated Design Thinking Oriented on Innovatively Personified Projects, Communications in Computer and Information Science. 2018. pp. 312–323.
9. Sosnin P. Experience-Based Human-Computer Interactions: Emerging Research and Opportunities. IGGlobal Press, 2017. 294 p.
10. IEEE 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. URL: https://standards.ieee.org/standard/610_12-1990.html (дата обращения: 27.11.2020).
11. Вигерс К, Битти Д. Разработка требований к программному обеспечению. 3-е изд., доп. М. : Русская редакция ; СПб. : БХВ-Петербург, 2018. 716 с.
12. Sosnin P. Question-answer nets as a valuable source of information in designing the software intensive system // 25th Telecommunications Forum, TELFOR 2017. Proceedings, 2017. pp. 1–4.
13. Гибкое управление в проектировании автоматизированных систем : учеб. пособие / П.И. Соснин, Ю.А. Лапшов, В.А. Маклаев, К.В. Святов. Ульяновск : УлГТУ, 2015. 204 с.

14. Программное управление потоками работ в компьютеризованных средах / П.И. Соснин, Ю.А. Лапшов, В.А. Маклаев, К.В. Святков. Ульяновск : УлГТУ, 2014. 308 с.

REFERENCES

1. *The Standish Group Report 83.9% of IT Projects Partially or Completely Fail*. Available at: <https://www.opendoorerp.com/the-standish-group-report-83-9-of-it-projects-partially-or-completely-fail/> (accessed 24.10.2020).
2. Maklaev V.A., Sosnin P.I. *Sozdanie i ispolzovanie avtomatizirovannoi bazy opyta proektnoi organizatsii* [Creation and Application of Automated Experience Base of Project Organization]. Ulyanovsk, UISTU Publ., 2012. 360 p.
3. Khleel Nasraldeen, Nehéz Károly. Tools, Processes and Factors Influencing of Code Review. *Multidiszciplináris Tudományok*, 2020, vol. 10, iss. 3, pp. 277–284.
4. Dürschmid Tobias. Continuous Code Reviews – A Social Coding Tool for Code Reviews inside the IDE. *First International Conference on the Art, Science, and Engineering of Programming, Association for Computing Machinery*. New York City, NY, United States, 2017, Article no. 41, pp. 1–3.
5. Carolyn D. Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Maggie Morrow Hodges, Collin Green, Ciera Jaspan, James Lin. Predicting Developers' Negative Feelings about Code Review. *International Conference on Software Engineering (2020)*. Available at: <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/7ac08fa960dfe10561c1f5953419a0c945279faa.pdf> (accessed 29.11.2020).
6. Ardic B., Yurdakul I., Tüzün E. Creation of a Serious Game for Teaching Code Review: An Experience Report. *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*. Munich, Germany, 2020, pp. 1–5.
7. Sadman N., Ahsan Md M., Parvez M.A. ADCR: An Adaptive Tool to Select "Appropriate Developer for Code Review" based on Code Context. *Proc. of the 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. New York City, NY, 2020, pp. 0583–0591.
8. Sosnin P. Automated Design Thinking Oriented on Innovatively Personified Projects, Communications in Computer and Information Science. *Information and Software Technologies. Proc. 24th Int. Conf. ICIST 2018*. 2018, pp. 312–323.
9. Sosnin P. *Experience-Based Human-Computer Interactions: Emerging Research and Opportunities*. IGGlobal Press, 2017. 294 p.
10. IEEE 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*. Available at: https://standards.ieee.org/standard/610_12-1990.html (accessed 27.11.2020).
11. Vigers K, Bitti D. *Razrabotka trebovaniy k programmnomu obespecheniiu*. 3-e izd., dop. [Design of Software Requirements 3d Ed. enl.]. Moscow, Russkaia redaktsiia, St. Petersburg, BHV-Peterburg, 2018. 716 p.
12. Sosnin P. Question-Answer Nets as a Valuable Source of Information in Designing the Software Intensive System. *Proceedings of 25th Telecommunications Forum, TELFOR 2017*. 2017, pp. 1–4.
13. Sosnin P.I., Lapshov Iu.A., Maklaev V.A., Sviatov K.V. *Gibkoe upravlenie v proektirovanii avtomatizirovannykh system*. Ucheb. posobie [Adaptive Management in Computer-Aided System Design. Tutorial]. Ulyanovsk, UISTU, 2015. 204 p.
14. Sosnin P.I., Lapshov Iu.A., Maklaev V.A., Sviatov K.V. *Programmnoe upravlenie potokami rabot v kompiuterizovannykh sredakh* [Adaptive Management of Workflows in Computer Environments]. Ulyanovsk, UISTU Publ., 2014. 308 p.