УДК: 004

1.2. Практическая реализация методологии CODE

Зекирьяев Р.Т., Болбаков Р.Г.,
МИРЭА – Российский технологический университет, Москва, Россия

В статье рассматривается интеграция методологии CODE (Codebase Operations and Development Practices) в процесс разработки программного обеспечения через создание плагина для интегрированной среды разработки IntelliJ IDEA. Методология CODE, основанная на адаптации цикла Деминга-Шухарта (PDCA), включает четыре этапа: Prepare, Develop, Control и Apply, каждый из которых направлен на стандартизацию и автоматизацию управления кодовой базой.

В работе обоснована необходимость практической реализации методологии в виде инструмента, интегрированного в среду разработки. Показано, что создание плагина для среды разработки позволяет разработчикам эффективно соблюдать стандарты, улучшать качество кода и ускорять выполнение повседневных задач. В статье описаны логика работы плагина, ключевые функции на каждом этапе методологии и примеры действий, инициируемых пользователем, таких как создание новых веток, проверка соблюдения стиля написания кода, запуск тестов и управление процессом слияния изменений.

Особое внимание уделено начальной стадии разработки плагина, включая настройки проекта и использование API IntelliJ IDEA для взаимодействия с системами контроля версий, анализа кода и тестирования. Предложены концептуальные примеры кода, иллюстрирующие реализацию функциональности.

Рассмотренные аспекты демонстрируют потенциал использования плагина для упрощения внедрения методологии CODE в реальную практику командной разработки. Продолжением этого исследования может стать непосредственная техническая реализация плагина. Представленные идеи закладывают основу для практического применения методологии CODE, направленного на повышение эффективности и стандартизации процессов разработки.

Введение

В научной статье, посвящённой теоретическому описанию методологии CODE (Codebase Operations and Development Practices) [1], был введён систематизированный подход к работе с кодовой базой, объединивший разрозненные практики разработки, проверки и интеграции изменений. Эта методология была предложена как решение проблемы отсутствия единого стандарта, охватывающего полный цикл работы с исходным кодом в профессиональной среде. Описанный в работе цикл Prepare-Develop-Control-Apply, вдохновлённый принципами Деминга-Шухарта [2], уже демонстрирует свою концептуальную состоятельность и предоставляет разработчикам чёткие рекомендации на каждом этапе.

Теоретическое описание, каким бы детальным оно ни было, не даёт возможности немедленного внедрения и апробации технологии. Современные требования разработки требуют практических инструментов, которые способны автоматически контролировать выполнение этапов, адаптироваться к разным условиям проектов и минимизировать человеческий фактор. Именно поэтому возникает необходимость создания инструментальной реализации методологии CODE, позволяющей сделать её применение удобным, интуитивно понятным и масштабируемым.

Работа с кодовой базой программного продукта связана с повседневной рутиной: создание веток, написание кода, соблюдение стиля написания кода (далее код-стайла) [3], тестирование и проверка написанного инкремента (далее код-ревью) [4]. Каждое из этих действий может быть стандартизировано, но для этого разработчику требуется вспомогательный инструмент, который будет направлять его, отслеживать этапы и предоставлять рекомендации.

Проблемы, связанные с применением методологии, как правило, обусловлены субъективным подходом команд к её внедрению. Например, разработчик может забыть отследить покрытие тестами только что разработанного инкремента или не проверить соответствие кода установленным стандартам. Такие ошибки могут становится систематическими, если отсутствует надлежащий контроль. Таким образом, необходимость автоматизации этапов методологии становится очевидной.

Создание модуля, или, рассуждая в контексте расширений для сред разработки, плагина для интегрированной среды разработки (IDE) [5], позволит не только облегчить процесс внедрения CODE, но и значительно ускорить адаптацию новых участников команды. Инструментальная реализация сделает методологию не просто сводом правил, а функциональной частью рабочего процесса, которую можно использовать

ежедневно. В то же время она сохраняет гибкость и адаптивность, позволяя командам вносить изменения и адаптировать подход под конкретные задачи.

Практическая реализация методологии также играет важную роль в обеспечении её жизнеспособности. Если CODE может быть применена в различных сценариях — от небольших стартапов до крупных корпоративных проектов — она имеет шансы стать стандартом в индустрии. Инструментальная поддержка обеспечит доказательство её эффективности, демонстрируя реальные результаты, такие как сокращение времени на доработки, повышение качества кода и улучшение коммуникации внутри команды.

Таким образом, необходимость практической реализации методологии CODE становится очевидной. Это шаг, который переводит теоретическую концепцию в реальный рабочий процесс, облегчая внедрение и гарантируя соответствие заявленным принципам. А наилучший способ внедрить полезную практику в существующий процесс – максимально автоматизировать применение этой практики. Основной инструмент разработчика программного обеспечения – это IDE или среда разработки [5], поэтому наибольшей эффективности применения методологии и охвата процессов создания программного продукта можно добиться модернизацией IDE. Основу расширений для среды разработки составляют плагины [6]. Поэтому далее будет рассмотрен выбор среды разработки для реализации инструмента и рассмотрен процесс интеграции.

Интеграция методологии в среду разработки

Для эффективного внедрения методологии CODE в повседневную практику разработчиков необходим инструмент, который будет интегрирован непосредственно в их рабочую среду. Такой подход обеспечит непрерывность процессов и минимизирует дополнительные усилия, связанные с переключением между различными приложениями или инструментами. В этом контексте интеграция методологии в виде модуля для среды разработки представляется наиболее целесообразным решением.

Интеграция методологии CODE в виде модуля станет важным шагом, который позволит разработчикам соблюдать установленные правила управления кодовой базой непосредственно в ходе своей повседневной работы. В процессе разработки программного обеспечения, где задачи часто связаны с созданием и поддержкой сложной структуры кода, стандартизированные подходы позволяют минимизировать количество ошибок, ускорить адаптацию новых участников команды и обеспечить соблюдение стандартов качества. Такой инструмент, как плагин, становится связующим звеном между теоретической моделью методологии и её практическим применением, делая её неотъемлемой частью рабочего процесса. Плагин не просто напоминает разработчикам о правилах, но активно направляет их на каждом этапе выполнения методологии, помогая им избегать распространённых ошибок, поддерживать стандарты и следовать лучшим практикам.

Среди множества существующих сред разработки IntelliJ IDEA выделяется как удобная платформа для реализации подобного инструмента. Её универсальность, мощные встроенные функции и возможность гибкого расширения делают её подходящей для интеграции методологии CODE. IntelliJ IDEA предоставляет разработчикам интеллектуальную поддержку кода, которая включает в себя автодополнение, статический анализ, подсказки по рефакторингу и автоматизацию рутинных задач [7]. Эти функции позволяют минимизировать затраты времени на исправление ошибок, обеспечивая высокое качество разрабатываемого программного обеспечения. Кроме того, встроенные средства работы с системами контроля версий, такими как Git, позволяют автоматизировать процессы управления изменениями в кодовой базе, что идеально сочетается с требованиями методологии CODE.

Ещё одним преимуществом IntelliJ IDEA является её архитектура, ориентированная на создание и интеграцию плагинов. Используя предоставленный компанией JetBrains SDK [8], разработчики могут создавать плагины, которые бесшовно интегрируются в интерфейс и функциональность среды. Это означает, что плагин для реализации CODE будет не просто дополнительным инструментом, а органичной частью рабочей среды разработчика. Интерфейс IntelliJ IDEA позволяет пользователям быстро освоить работу с новым функционалом, не теряя времени на сложные настройки или обучение.

Ценность IntelliJ IDEA представляет и для команд, работающих с различными языками программирования. Её широкая поддержка технологий делает её универсальным инструментом, способным удовлетворить потребности разработчиков вне зависимости от того, используют ли они Java, Kotlin, Python, JavaScript или другой язык. Это особенно важно в контексте современной разработки, где проекты часто требуют междисциплинарного подхода и взаимодействия между командами, использующими разные технологии. Универсальность IntelliJ IDEA позволяет интегрировать методологию CODE в проекты любого масштаба, от небольших стартапов до крупных корпоративных решений.

Интеграция методологии CODE в среду разработки через плагин для IntelliJ IDEA открывает значительные перспективы для оптимизации процессов разработки и управления кодовой базой. Это решение позволит командам стандартизировать свои подходы, минимизировать негативное влияние человеческого фактора и ускорить разработку проектов. Кроме того, реализованный плагин может быть адаптирован под специфические нужды различных проектов, что сделает его универсальным и перспективным инструментом для профессиональной среды.

Перед тем, как перейти к описанию логики работы плагина, вспомним основные концепции и идеи методологии CODE.

Методология CODE

Методология наследует структуру цикла Деминга-Шухарта PDCA (Plan-Do-Check-Act). В рамках методологии CODE цикл PDCA имеет структуру: Prepare-Develop-Control-Apply. На первом этапе выполняется

подготовка к разработке, подготовка включает в себя информацию, необходимую уже непосредственно при работе с кодом. На втором этапе методологии выполняется разработка по характерным для конкретного проекта правилам. Третий этап – проверка, он включает в себя принятые практики анализа написанного кода. Четвёртый этап включает в себя применение созданных изменений. Сам цикл представлен на рисунке 1. Рассмотрим, что конкретно включает в себя каждый из этапов.

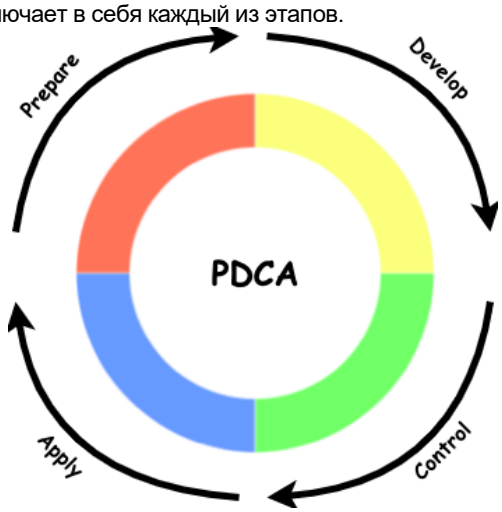

Рисунок 1 – Цикл Prepare-Develop-Control-Apply

Prepare (Подготовка)

При работе с большими объёмами кода используются системы контроля версий. Для эффективной работы с системой в параллели с одной кодовой базой используются различные модели ветвления (Branching) – Feature Based Development (FBD), Trunk Based Development (TBD). Очень важно, чтобы все разработчики понимали, какая модель используется в их кодовой базе, как выполнять доработки и слияние своих доработок с общей кодовой базой, сколько коммитов должно быть в локальной ветке. Поэтому самым первым шагом является изучение модели ветвления и всех принятых практик по работе с системой контроля версий – отведение веток для релиза, хотфиксов [9] и простых доработок. Основа этого этапа – описанные практики работы с системой контроля версий.

Develop (Разработка)

После выполнения подготовки, создания собственного пространства для работы в системе контроля версий, наступает черёд непосредственно разработки. Разработка также не должна носить хаотичный характер – поэтому при написании кода хорошей практикой является следование стилю написания кода, или Code style (CS). Code style содержит в себе информацию об именовании сущностей в проекте, расположении свойств и методов в рамках класса, рекомендации по оформлению кода и написанию комментариев. В рамках Code style также могут быть даны рекомендации по написанию методов или реализации задач, специфичных для конкретной кодовой базы. Здесь же могут описываться используемые в разработке шаблоны проектирования. Ключевой регламент этапа разработки – стиль написания кода.

Control (Проверка)

За разработкой следует этап проверки, или тестирования. На этом этапе код уже приобрёл бизнес-ценность [10] и её следует проверить на соответствие ожиданиям. В первую очередь выполняется ручная проверка разработанной функциональности. Далее на этом же этапе выполняется автоматизации тестирования через написание тестов. Например, хорошей практикой будет следование упомянутой ранее пирамиде тестирования – написание малой доли End to end (E2E) тестов, чуть большего числа интеграционных тестов и большого числа, относительно предыдущих двух типов, Unit-тестов. На этом этапе следует описать практики автоматизированного тестирования.

Завершающий этап цикла – применение разработанных изменений. В соответствии с практиками работы с системой контроля версий создаётся запрос на слияние с основной кодовой базой созданных изменений (Pull/Merge Request – PR/MR). Создание таких запросов также желательно описывать, чтобы избежать различий в форматах, например: Название, описание, пояснения к доработкам, размер изменений в одном запросе. За этапом создания запроса следует Code Review (CR) – проверка вносимых изменений другими разработчиками проекта. CR может быть очень сложным процессом, поэтому следует заранее договориться о правилах его проведения – просматривать поступившие запросы на CR за конкретный, ограниченный во времени срок, абстрагироваться от эмоционального контекста, если он мешает качественному просмотру доработок [11].

Это ключевые концепции каждого этапа методологии CODE. Далее рассмотрим возможную логику работы плагина, включая действия пользователя, инициирующие переходы между этапами методологии CODE, и соответствующие реакции системы.

Логика работы плагина

Методология CODE может использоваться в реальных проектах уже в текущем виде, в том, в котором она описана в первой научной статье цикла, и в том же виде, что она описана выше. В таком варианте исполнения она может находиться в корневом пакете проекта и выполнять роль документации для введения в контекст проекта (онбординга) [12] новых разработчиков или для обращения уже работающих участников проекта с целью вспомнить основные договорённости, зафиксированные для работы с кодовой базой. Но если автоматизировать пункты, описанные в методологии, то можно получить мощный инструмент для регулярной поддержки качества и управления кодовой базой. Перейдём к описанию возможных вариантов реализации методологии CODE как плагина для среды разработки.

Плагин, реализующий методологию CODE в среде IntelliJ IDEA, направлен на упрощение и автоматизацию процессов разработки, он обеспечивает последовательное выполнение этапов методологии: Prepare, Develop, Control и Apply.

Этап Prepare открывает цикл создания инкремента кодовой базы, на нём закладывается основа для эффективной работы с кодом. На этом этапе плагин может предоставлять разработчику возможность создать новую ветку для работы над функциональностью или исправлением ошибок. Плагин может отображать диалоговое окно с предложением выбрать тип ветки, например, feature, bugfix или hotfix, основываясь на принятой модели ветвления проекта [13]. После выбора типа ветки пользователь вводит её имя. Здесь плагин может играть роль контролирующего механизма: он проверяет, соответствует ли имя ветки заданным стандартам. Например, если имя ветки не начинается с префикса feature/ или содержит запрещённые символы, плагин может уведомлять разработчика об ошибке и предлагать её исправить. В дополнение к этому плагин может предлагать ссылку на внутреннюю документацию проекта, где описаны правила именования веток, или открывать файл в проекте, в котором подробно изложены эти стандарты. Это может позволить разработчику сразу ознакомиться с необходимыми требованиями без необходимости покидать среду разработки.

На этапе Develop плагин обеспечивает поддержку соблюдения стандартов кодирования и стиля. Это может достигаться благодаря интеграции с инструментами статического анализа кода [14]. После каждого сохранения файла плагин может инициировать проверку текущего состояния кода, выявляя потенциальные нарушения стиля, использование устаревших конструкций или наличие логических ошибок. Если такие нарушения обнаруживаются, плагин может уведомлять пользователя через всплывающее окно или вывод сообщений в консоли IDE. При этом плагин может не только указывать на проблему, но и предлагать варианты её решения. Например, если код не соответствует принятому стилю, плагин может предоставлять кнопку для автоматического форматирования, используя заранее настроенные правила. Для более сложных случаев, таких как обнаружение устаревших методов, плагин, используя средства IDE, может предлагать ссылки на документацию или ресурсы, объясняющие, почему метод больше не рекомендуется и какие альтернативы существуют. Таким образом, плагин может не только исправлять ошибки, но и способствовать обучению разработчика, предоставляя ему доступ к полезной информации.

Этап Control представляет собой центральный элемент процесса проверки качества кода. После завершения разработки функциональности плагин может инициировать процесс тестирования. Он может автоматически запускать все модульные тесты, настроенные в проекте, и проверять уровень покрытия кода тестами. Если тесты успешно проходят, плагин может уведомлять разработчика и предлагать следующий шаг — создание PR для последующего код-ревью. Здесь плагин может предоставлять шаблон для описания изменений, в который включаются автоматически сгенерированные метрики, такие как количество новых строк кода, уровень покрытия тестами и список затронутых файлов. Это поможет разработчику представить свои изменения в структурированном виде. Если же тесты не пройдены, плагин может формировать подробный отчёт, включающий описание ошибок, их местоположение и возможные причины. Отчёт может быть представлен в виде HTML-документа с гиперссылками, ведущими к строкам кода, вызвавшим ошибки. Кроме того, плагин может предоставлять рекомендации по устранению проблем, основанные на анализе тестов и стандартных шаблонах решений.

На завершающем этапе Apply плагин играет важную роль в интеграции изменений в основную кодовую базу. После успешного прохождения CR плагин может проверять, были ли исправлены все замечания, оставленные ревьюерами, и предлагать разработчику завершить процесс слияния. Перед слиянием плагин может автоматически обновлять ветку разработки последними изменениями из основной ветки, чтобы минимизировать вероятность конфликтов. Если конфликты всё же возникают, плагин может предоставлять инструменты для их разрешения, такие как визуальный редактор, позволяющий наглядно сравнивать конфликтующие версии. После успешного завершения слияния плагин может предложить удалить временную ветку, чтобы сохранить порядок в репозитории. Это действие может сопровождаться напоминанием о необходимости обновления документации, связанной с реализованной функциональностью, если такое требуется. В качестве дополнительной опции плагин может отправлять уведомления другим участникам команды о завершении работы над задачей, предоставляя ссылку на итоговый PR и описание внесённых изменений.

Для каждого из описанных этапов плагин может работать в двух режимах: индикативном и активном. В индикативном режиме плагин может предоставлять пользователю подсказки и ссылки на ресурсы, такие как методология или правила проекта. Например, на этапе подготовки он может открывать файл CODE.md или раздел документации, где подробно описаны правила именования веток. В активном режиме плагин может выполнять проверки, предоставлять рекомендации и инициировать автоматические действия, такие как запуск тестов или форматирование кода. Это может делать его универсальным инструментом, подходящим как

для опытных разработчиков, которым нужна только справочная информация, так и для начинающих, которым требуется активная поддержка на каждом этапе.

Таким образом, плагин может стать не просто инструментом, а полноценным проводником методологии CODE, обеспечивая её интеграцию в повседневную работу разработчиков.

Конфигурация плагина

Разработка плагина для IntelliJ IDEA начинается с создания нового проекта с использованием Gradle [15] и с подключением необходимых зависимостей. Настроим плагин для работы не только с IntelliJ, но и с Android Studio [16], модификацией среды для Android-разработки. Выполним этот шаг для расширения охвата разработчиков [17], которые могут воспользоваться потенциальными возможностями будущего плагина, реализующего методологию CODE.

Приступим непосредственно к конфигурации будущего плагина. Принцип действия плагина описан в файлах конфигурации. После создания пустого проекта плагина в IntelliJ IDEA в файле build.gradle.kts указываются основные параметры плагина. Рассмотрим этот файл, он представлен в листинге 1.

Листинг 1 – Часть конфигурации плагина в файле build.gradle.kts

```
plugins {
    id("java")
    id("org.jetbrains.kotlin.jvm") version "1.9.23"
    id("org.jetbrains.intellij") version "1.17.3"
}

group = "ru.mirea"
version = "1.0.0"

repositories {
    mavenCentral()
}

intellij {
    version.set("2023.2.6")
    plugins.set(listOf("Kotlin", "java", “androi”))
}
```

Версии подключённых плагинов актуальны на момент написания работы, однако могут меняться в будущем.

В данном файле подключаются следующие плагины:

Java-плагин (id("java")): обеспечивает поддержку разработки на языке Java, предоставляя необходимые инструменты для компиляции и сборки Java-проектов.

Kotlin JVM плагин (id("org.jetbrains.kotlin.jvm")): добавляет поддержку языка Kotlin в проекты, предназначенные для Java Virtual Machine (JVM), что позволяет писать код на Kotlin и компилировать его в байт-код JVM.

IntelliJ плагин (id("org.jetbrains.intellij")): предназначен для разработки плагинов под платформу IntelliJ, включая Android Studio. Он упрощает процесс создания, сборки и тестирования плагинов, обеспечивая интеграцию с экосистемой JetBrains.

Параметры group и version определяют группу и версию создаваемого плагина соответственно, что необходимо для его идентификации и управления версиями.

В разделе repositories указывается репозиторий Maven Central [18], из которого будут загружаться необходимые зависимости.

Блок intellij настраивает параметры среды разработки, для которой создаётся плагин:

version.set("2023.2.6"): определяет версию IntelliJ IDEA, с которой будет совместим плагин.

plugins.set(listOf("Kotlin", "java", "android")): задаёт список плагинов, необходимых для работы разрабатываемого плагина. В данном случае подключаются плагины для поддержки Kotlin, Java и Android, что обеспечивает совместимость плагина как с IntelliJ IDEA, так и с Android Studio.

Особенностью разработки плагина для Android Studio является необходимость учёта специфики Android-разработки, включая работу с Android SDK [19], ресурсами и специфическими инструментами. Поэтому важно убедиться, что в настройках плагина подключены соответствующие модули и зависимости, обеспечивающие корректную работу в среде Android Studio.

Далее создаётся файл манифеста плагина plugin.xml, в котором определяются действия, расширения и слушатели, предоставляемые плагином:

Листинг 2 - plugin.xml

```
<idea-plugin>
    <id>ru.mirea.codeplugin</id>
```

```
    <name>CODE Methodology Plugin</name>
    <version>1.0.0</version>
    <description>
      Плагин для интеграции методологии CODE в IntelliJ IDEA.
    </description>
    <depends>com.intellij.modules.platform</depends>
    <depends>com.intellij.java</depends>
    <depends>org.jetbrains.kotlin </depends>
    <depends>org.jetbrains.android </depends>

    <extensions defaultExtensionNs="com.intellij">
      <!-- Определение действий и расширений -->
    </extensions>

    <actions>
      <!-- Определение пользовательских действий -->
    </actions>
  </idea-plugin>
```

В этом файле указываются идентификатор плагина, его название, версия, информация о поставщике (это поле удалено из листинга), а также зависимости и расширения, которые плагин добавляет в среду разработки. Особое внимание нужно уделить блокам <depends>, т.к. они предоставляют зависимости, с которыми в дальнейшем и предстоит реализовать плагин.

После настройки проекта и определения основных компонентов, таких как файл манифеста plugin.xml, следующим шагом будет реализация функциональности, соответствующей каждому этапу методологии: Prepare, Develop, Control и Apply. В рамках этой работы будут очень кратко рассмотрены варианты действий для каждого этапа, но без конкретной технической реализации.

Автоматизация этапов методологии CODE основывается на действиях разработчика, которые являются триггерами для перехода между этапами. Плагин будет отслеживать эти действия в процессе работы и предоставлять необходимые рекомендации или выполнять автоматизированные операции. Каждый этап — Prepare, Develop, Control, Apply — инициируется конкретными действиями пользователя и сопровождается логической реакцией системы, что позволяет сделать рабочий процесс структурированным и контролируемым.

Этап Prepare. На этапе подготовки плагин реагирует на создание новой ветки в системе контроля версий. В большинстве современных команд создание веток — это первый шаг при начале работы над новой задачей. Для реализации этого процесса плагин должен уметь взаимодействовать с системой контроля версий Git [20] через соответствующие API IntelliJ IDEA.

Пользователь инициирует этап Prepare либо выбирая пункт из меню плагина «Создать ветку» или вызывая соответствующую команду через горячие клавиши, либо через введение имени новой ветки в окне создания ветки, которое отображается при активации плагина. Также начало выполнения этапа возможно при выполнении команды fetch для обновления локального репозитория.

На данном этапе плагин выполняет несколько последовательных операций. Сначала проверяется корректность введённого имени ветки. Например, имя ветки должно соответствовать определённому паттерну, принятому в проекте (например, feature/, bugfix/ или hotfix/). Если имя ветки не соответствует стандартам, плагин уведомляет пользователя о необходимости корректировки.

После успешного создания ветки система фиксирует событие и автоматически выполняет git-checkout для переключения на новую ветку. Пример реализации этого шага может выглядеть следующим образом:

Листинг 3 – Псевдокод варианта реализации этапа Prepare

```
val project: Project = getProject(…)
val git = Git.getInstance()
val branchName = "feature/new-feature"

// Проверка имени ветки
if (isBranchNameValid(branchName)) {
  git.createBranch(project, branchName)
  git.checkout(project, branchName)
} else {
  notifyUser("Имя ветки не соответствует стандарту")
}
```

Завершение создания ветки является сигналом о переходе на следующий этап методологии.

Этап Develop. Этап разработки начинается сразу после успешного создания ветки и выполнения команды checkout. На данном этапе действия пользователя сводятся к редактированию исходного кода, что

плагин отслеживает по событиям изменения файлов в редакторе. Каждое сохранение файла является триггером для выполнения дополнительных проверок на соответствие стандартам кодирования и стилю.

Интеграция со статическим анализатором (например, CheckStyle или встроенные инспекции IntelliJ IDEA) позволяет плагину анализировать код и автоматически выявлять нарушения. Результаты анализа отображаются в виде уведомлений в панели задач IDE или встроенном окне проблем. Для исправления нарушений плагин может предлагать быстрые исправления с использованием Quick Fix.

Выполнение проверок запускается автоматически при сохранении изменений пользователем.

Листинг 4 – Псевдокод варианта реализации этапа Develop

```
override fun documentChanged(event: DocumentEvent) {
    val file = event.document.virtualFile
    if (isSourceCode(file)) {
        val issues = runCodeStyleCheck(file)
        if (issues.isNotEmpty()) {
            notifyUser("Найдены проблемы с код-стайлом: $issues")
        }
    }
}
```

На этапе Develop плагин также может предоставлять пользователю подсказки по рефакторингу или улучшению производительности кода, активируя их по запросу через соответствующее меню.

Этап Control начинается, когда разработчик завершает редактирование функциональности и инициирует процесс тестирования. Триггером для этого этапа служит сохранение всех изменений и запуск тестов вручную через плагин или IDE.

Результаты тестов обрабатываются плагином и отображаются пользователю в структурированном виде. В случае успешного выполнения всех тестов плагин предлагает создать commit и выполнить push изменения в удалённый репозиторий для дальнейшего создания PR и прохождения CR. Если тесты завершились с ошибками, плагин уведомляет пользователя и предоставляет детальный отчёт с указанием причин их падения. Пример инициации тестов выглядит следующим образом:

Листинг 5 – Псевдокод варианта реализации этапа Control

```
val testManager = TestManager.getInstance(project)
testManager.runAllTests {
    if (it.isSuccess) {
        notifyUser("Все тесты пройдены. Создайте Pull Request")
    } else {
        notifyUser("Ошибка тестов: ${it.errors}")
    }
}
```

На завершающем этапе Apply разработчик принимает решение об интеграции изменений в основную ветку. Триггером для перехода к этапу Apply является успешное прохождение код-ревью и отсутствие конфликтов при слиянии. Плагин отслеживает статус PR и, в случае положительного результата, предлагает выполнить слияние ветки в основную ветку. После завершения слияния плагин автоматически удаляет временную ветку и обновляет локальный репозиторий.

Процесс слияния может быть реализован следующим образом:

Листинг 6 – Псевдокод варианта реализации этапа Apply

```
val branchName = "feature/new-feature"
git.mergeBranch(project, "main", branchName)
git.deleteBranch(project, branchName)
notifyUser("Ветка $branchName успешно слита и удалена")
```

Описанная выше логика может стать основой активного режима работы плагина, в то время, как индикативный режим работы плагина будет отображать подсказки на основе текущего этапа выполнения методологии.

Таким образом, действия пользователя на каждом этапе методологии CODE становятся триггерами для выполнения автоматизированных операций, контролируемых плагином. Система не только направляет разработчика по методологии, но и минимизирует ошибки, повышая эффективность и качество разработки. Плагин, реализующий методологию CODE, сможет обеспечить последовательное и стандартизированное выполнение всех этапов разработки, повышая качество кода и эффективность работы команды. Следующим этапом является непосредственная разработка плагина с техническими подробностями, однако это выходит за рамки настоящей статьи.

Заключение

Итак, в данной работе мы рассмотрели концептуальный подход к интеграции методологии CODE в процесс разработки программного обеспечения посредством создания плагина для среды разработки. Было обсуждено, что практическая реализация методологии в виде инструмента, интегрированного в рабочую среду разработчиков, значительно упрощает её внедрение и использование. Подробно описаны этапы методологии (Prepare, Develop, Control и Apply) и логика работы плагина на каждом из них. Приведены примеры действий пользователя, инициирующих переход между этапами, и реакции системы, включая проверку стандартов именования веток, соблюдение код-стайла, автоматизацию тестирования и управление процессом слияния изменений.

Рассмотренные аспекты продемонстрировали, как инструментальная реализация методологии может повысить стандартизацию, удобство и качество процессов разработки. Плагин предлагает последовательное выполнение этапов методологии, минимизирует влияние человеческого фактора и способствует улучшению коллективной работы.

Перспективность развития методологии обусловлена в том числе запуском нового национального проекта «Экономика данных», пришедшим на смену предыдущему проекту «Цифровая экономика» [21]. Национальный проект «Цифровая экономика» был рассчитан на 6 лет, и 10 января 2025 года были подведены его итоги [22]. Нормативное регулирование цифровой среды было одной из целей предыдущего этапа развития цифровизации, на новом этапе выделяются направления развития стандартов работы с данными, а также обработки и анализа данных, репозиториев открытого кода [23], что свидетельствует о высоком интересе на уровне государства к сфере формализации информационных технологий, в том числе управления кодом как данными.

Продолжением данной работы станет техническая реализация описанного плагина, включая разработку пользовательских интерфейсов, интеграцию с системой контроля версий и реализацию функций для каждого этапа методологии. Технические аспекты разработки, включая кодовые решения, архитектуру плагина и детали его тестирования, будут подробно освещены в следующем исследовании. Таким образом, предложенная концепция переходит в стадию реализации, что станет важным шагом к практическому применению методологии CODE в реальных проектах.

Литература

1. Зекирьяев, Р. Т. Методология управления кодовой базой программных комплексов в условиях цифровой экономики / Р. Т. Зекирьяев, Р. Г. Болбаков // Цифровая экономика. – 2024. – № 1(27). – С. 78-85.
2. Deming, W. Edwards (2000). Out of the crisis (1. MIT Press ed.). Cambridge, Mass.: MIT Press, 2000. p. 88. ISBN 0262541157.
3. Coding conventions. URL: https://kotlinlang.org/docs/coding-conventions.html
4. Baum, Tobias; Liskin, Olga; Niklas, Kai; Schneider, Kurt (2016). "A Faceted Classification Scheme for Change-Based Industrial Code Review Processes". 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). pp. 74–85.
5. What is an IDE? URL: https://aws.amazon.com/what-is/ide
6. IDE (Integrated Development Environment) Plugins. URL: https://startup-house.com/glossary/ide-integrated-development-environment-plugins
7. Getting started | IntelliJ IDEA Documentation URL: https://www.jetbrains.com/help/idea/getting-started.html
8. IntelliJ Platform SDK. URL: https://plugins.jetbrains.com/docs/intellij/welcome.html
9. Hotfix. URL: https://en.wikipedia.org/wiki/Hotfix
10. Sward, David. Measuring the Business Value of Information Technology Practical Strategies for IT and Business Managers. Intell Press. 2006. 282 p.
11. Marco Ortu, Giuseppe Destefanis, Daniel Graziotin, Michele Marchesi, Roberto Tonelli. How Do You Propose Your Code Changes? Empirical Analysis of Affect Metrics of Pull Requests on GitHub. 2020. 11 p.
12. Amanda J. Painter, Brenda A. Haire. The Onboarding Process: How to Connect Your New Hire. 2022.
13. Mike Terhar. GitLab Book: Git Branching Strategies 2020. URL: https://brownfield.dev/post/2020-06-27-git-branching-book/
14. Ryan Dewhurst. Static Code Analysis. URL: https://owasp.org/www-community/controls/Static_Code_Analysis
15. Gradle User Manual. URL: https://docs.gradle.org/current/userguide/userguide.html
16. Android Studio. URL: https://developer.android.com/studio
17. How Many Software Engineers Are There in the World? URL: https://www.allstarsit.com/blog/how-many-software-engineers-are-there-in-the-world
18. Maven Central. URL: https://mvnrepository.com/
19. SDK Platform Tools release notes. URL: https://developer.android.com/tools/releases/platform-tools
20. Scott Chacon, Ben Straub. Pro Git. New York, NY: Apress. 2025. 495 p.
21. «Цифровая экономика РФ»: Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации. URL: https://digital.gov.ru/ru/activity/directions/858/

22. Нацпроект «Цифровая экономика»: Результаты за 6 лет. URL: https://национальныепроекты.рф/news/natsproekt-tsifrovaya-ekonomika-rezultaty-za-6-let/
23. В России появится новый нацпроект — «Экономика данных»: Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации. URL: https://digital.gov.ru/ru/events/45686/ .

References in Cyrillics

1. Zekir'yaev, R. T. Metodologiya upravleniya kodovoj bazoj programmnyh kompleksov v usloviyah cifrovoj ekonomiki / R. T. Zekir'yaev, R. G. Bolbakov // Cifrovaya ekonomika. – 2024. – № 1(27). – S. 78-85.
2. Deming, W. Edwards (2000). Out of the crisis (1. MIT Press ed.). Cambridge, Mass.: MIT Press, 2000. p. 88. ISBN 0262541157.
3. Coding conventions. URL: https://kotlinlang.org/docs/coding-conventions.html
4. Baum, Tobias; Liskin, Olga; Niklas, Kai; Schneider, Kurt (2016). "A Faceted Classification Scheme for Change-Based Industrial Code Review Processes". 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). pp. 74–85.
5. What is an IDE? URL: https://aws.amazon.com/what-is/ide
6. IDE (Integrated Development Environment) Plugins. URL: https://startup-house.com/glossary/ide-integrated-development-environment-plugins
7. Getting started | IntelliJ IDEA Documentation URL: https://www.jetbrains.com/help/idea/getting-started.html
8. IntelliJ Platform SDK. URL: https://plugins.jetbrains.com/docs/intellij/welcome.html
9. Hotfix. URL: https://en.wikipedia.org/wiki/Hotfix
10. Sward, David. Measuring the Business Value of Information Technology Practical Strategies for IT and Business Managers. Intell Press. 2006. 282 p.
11. Marco Ortu, Giuseppe Destefanis, Daniel Graziotin, Michele Marchesi, Roberto Tonelli. How Do You Propose Your Code Changes? Empirical Analysis of Affect Metrics of Pull Requests on GitHub. 2020. 11 p.
12. Amanda J. Painter, Brenda A. Haire. The Onboarding Process: How to Connect Your New Hire. 2022.
13. Mike Terhar. GitLab Book: Git Branching Strategies 2020. URL: https://brownfield.dev/post/2020-06-27-git-branching-book/
14. Ryan Dewhurst. Static Code Analysis. URL: https://owasp.org/www-community/controls/Static_Code_Analysis
15. Gradle User Manual. URL: https://docs.gradle.org/current/userguide/userguide.html
16. Android Studio. URL: https://developer.android.com/studio
17. How Many Software Engineers Are There in the World? URL: https://www.allstarsit.com/blog/how-many-software-engineers-are-there-in-the-world
18. Maven Central. URL: https://mvnrepository.com/
19. SDK Platform Tools release notes. URL: https://developer.android.com/tools/releases/platform-tools
20. Scott Chacon, Ben Straub. Pro Git. New York, NY: Apress. 2025. 495 p.
21. «Cifrovaya ekonomika RF»: Ministerstvo cifrovogo razvitiya, svyazi i massovyh kommunikacij Rossijskoj Federacii. URL: https://digital.gov.ru/ru/activity/directions/858/
22. Nacproekt «Cifrovaya ekonomika»: Rezul'taty za 6 let. URL: https://национальныепроекты.рф/news/natsproekt-tsifrovaya-ekonomika-rezultaty-za-6-let/
23. V Rossii poyavitsya novyj nacproekt — «Ekonomika dannyh»: Ministerstvo cifrovogo razvitiya, svyazi i massovyh kommunikacij Rossijskoj Federacii. URL: https://digital.gov.ru/ru/events/45686/ .

Зекирьяев Руслан Тимурович,
аспирант кафедры инструментального и прикладного программного обеспечения Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: auriax@ya.ru
ORCID: 0000-0003-3787-7511

Болбаков Роман Геннадьевич, к.т.н., доцент,
заведующий кафедрой инструментального и прикладного программного обеспечения Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78).
E-mail: bolbakov@mirea.ru . ORCID: 0000-0002-4922-7260

Ключевые слова
Разработка программного обеспечения, управление разработкой программного обеспечения, кодовая база, разработка плагина, методология CODE

Keywords
Software Development, Software Project Management, Code Base, Information Technology Standards, Plugin Development, CODE Methodology.

DOI: 10.34706/DE-2025-01-02
JEL classification O32.

Abstract
The article deals with the integration of the CODE (Codebase Operations and Development Practices) methodology into the software development process through the creation of a plug-in for the IntelliJ IDEA integrated development environment. The CODE methodology, based on the Deming-Schuhart cycle adaptation (PDCA), includes four stages: Prepare, Develop, Control and Apply, each of which is aimed at standardization and automation of codebase management.

The paper substantiates the necessity of practical implementation of the methodology in the form of a tool integrated into the development environment. It is shown that creating a plug-in for the development environment allows developers to effectively comply with standards, improve code quality and speed up the performance of everyday tasks. The article describes the logic of the plugin, key functions at each stage of the methodology and examples of user-initiated actions such as creating new branches, checking code compliance, running tests and managing the process of merging changes.

Special attention is given to the initial stage of plugin development, including project settings and using the IntelliJ IDEA API to interact with version control systems, code analysis, and testing. Conceptual code samples are provided to illustrate the implementation of functionality.

The aspects discussed demonstrate the potential of using a plug-in to simplify the implementation of the CODE methodology into real-world team development practices. A direct technical implementation of the plug-in could be a continuation of this research. The presented ideas lay the foundation for practical application of the CODE methodology aimed at increasing the efficiency and standardization of development processes.