ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г.

СОДЕРЖАНИЕ

Журнал зарегистрирован в Федеральной службе по надзону в сфере связи, информационных технологий и массовых коммуникаци. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индексы: по каталогу агентства "Роспечать" — **22765**, по Объединенному каталогу "Пресса России" — **39795**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru E-mail: prin@novtex.ru

УДК 004.41

В. В. Липаев, д-р техн. наук, проф., гл. науч. сотр., Институт системного программирования РАН,
e-mail: alexlip@mail.ru

Сертификация программных продуктов для управляющих систем

Изложены методические основы обеспечения качества производства и сертификации программных продуктов для управляющих систем. Представлены два метода обеспечения гарантий качества таких комплексов программ: сертификация технологических процессов производства и сертификация готовых программных продуктов. Для каждого метода представлены структуры процедур и содержания документов при проектировании, а также при производстве сложных программных продуктов и их сертификации.

Ключевые слова: *сертификация, программные продукты, качество программ, производство и испытания программ, документирование процессов и результатов сертификации*

1. Методические основы сертификации качества производства и программных продуктов

Потребителей — заказчиков программных продуктов для управляющих систем интересует, прежде всего, ***качество готового продукта*** и обычно не очень беспокоит, как оно достигнуто. Однако это качество должно быть ответственно ***удостоверено и гарантировано*** компетентными, независимыми организациями. Гарантировать качество продукции можно посредством сертификационных испытаний ***процессов производства*** комплексов программ и/или испытаний их результатов — ***готовых программных продуктов*** [1]. Рассматриваемые далее комплексы программ для систем управления и обработки информации в реальном времени активно применяются в сложных критических системах динамического управления объектами, в высокоточном технологическом производстве, в авиации и космонавтике, в атомной энергетике и оборонной промышленности. Проектирование и производство таких программных продуктов систем управления реального времени различного назначения заказчиками обычно требуется базировать на международных стандартах, охватывающих весь их жизненный цикл. Такие изделия являются одними из наиболее сложных интеллектуальных систем высочайшего качества, создаваемых человеком, для которых доступна и необходима сертификация не только производственных процессов, но и их результатов — программных продуктов.

Сертификация — это процедуры подтверждения соответствия продукции требованиям и стандартам, установленным заказчиком, независимым от изготовителя и потребителя. Сертификационные испытания должны ***технически и юридически удостоверять*** в письменной форме то, что состояние продукции, процессов его производства и системы менеджмента качества способны обеспечить требуемое качество и стабильность характеристик изготовляемой продукции любыми ***двумя методами*** (рис. 1).

Первый метод (сертификация процессов производства) должен обеспечивать высокое качество выполнения всего технологического процесса проектирования и производства, и, тем самым, минимум экономических потерь от брака, что особенно важно при создании сложных дорогих систем. Результаты испытаний качества ***процессов производства*** при таком подходе трудно измерять количественными критериями, и они, как правило, характеризуются рядом требований к качественному выполнению наборов стандартизированных производственных процессов. Такие результаты оцениваются свойствами различных процессов, которые непосредственно отражаются на характеристиках качества программного продукта.

Рис. 1. Базовая схема сертификации сложных программных продуктов — испытания технологий и конечного программного продукта

Однако при этом нет гарантии адекватного и однозначного выполнения требований по качеству конечного продукта. При производстве этот метод может приводить к неконтролируемому, неизвестному качеству компонентов и комплексов программ в целом, и к значительным экономическим потерям за счет затрат на создание непригодного к использованию продукта (брака), что может быть для сложных систем относительно дорого.

Второй метод (сертификация продуктов производства) сертификации акцентирован на анализе, контроле и удостоверении качества готового программного продукта, которое подтверждается при его испытаниях. Отсутствие или недостатки системы обеспечения качества в технологическом процессе разработки могут приводить к длительному итерационному процессу доработок и повторных испытаний. При сертификационных испытаниях ***готового программного продукта*** могут использоваться его стандартизированные количественные и качественные критерии качества и характеристики, которые непосредственно отражают функции и свойства продукции, интересующие заказчика и потребителей, их можно измерить и достоверно установить реальные значения.

Соответственно можно выделить ***два вида сертификационных испытаний: испытания технологий*** обеспечения жизненного цикла программных комплексов, поддержанных регламентированными системами качества и испытания готового программного ***продукта*** с полным комплектом эксплуатационной документации (см. рис. 1). Взаимосвязь качества разработанных комплексов программ с качеством технологии их создания и с затратами на производство становится особенно существенной в случае, если необходимо получение ***критического программного продукта реального времени*** с особенно высокими значениями характеристик качества при ограниченных ресурсах [2]. Этот вид комплексной сертификации должен обеспечивать контроль реализации требований алгоритмической и функциональной корректности программного продукта. Такой подход особенно важен в программных комплексах для ***обеспечения функциональной безопасности применения сложных управляющих систем (см. стандарт IEC 61508)***.

Сертификация производства продукции различных видов регламентирована стандартами: ГОСТ Р ИСО 9001—2001; ГОСТ Р 40.003—2005 и ГОСТ Р ИСО 19011—2003, а также ***комплексом международных стандартов*** создания жизненного цикла программных продуктов и их компонентов. Они акцентированы на ***системе менеджмента качества производства***. При этом сертификация производства определена как ***процедура подтверждения соответствия***, посредством которой независимая от изготовителя (продавца, исполнителя) и потребителя (покупателя) организация удостоверяет в письменной форме, что процедура (процесс) производства (системы менеджмента качества производства) способна обеспечить требуемое качество и стабильность характеристик изготовляемой конкретной продукции.

Испытания процедур производства программных продуктов должны осуществлять эксперты (аудиторы) по сертификации производств, зарегистрированные в Регистре системы сертификации персонала — ***серти-***

фицированные специалисты. Область сертификации определяет заказчик по согласованию с председателем комиссии органа по сертификации конкретной продукции. На практике акцент, распределение ресурсов и усилий на два вида сертификации зависят от особенностей характеристик комплекса программ, квалификации коллектива специалистов-разработчиков, требований заказчиков-потребителей и наличия у сертифицирующей организации соответствующей тематической квалификации. Для этого организация и процессы сертификации должны специализироваться на определенных классах программных продуктов, предусматривать соответствующие технологические работы и документы, обеспечивающие создание продукта требуемого качества [1, 2].

В двух последующих разделах статьи приводятся примеры процедур и содержания документов для сертификации технологических процессов производства и сертификации качества готовых программных продуктов. Эти разделы могут использоваться в качестве методической базы для подготовки руководящих документов и инструкций при производстве конкретных программных продуктов и их сертификации, а также для подготовки специалистов-сертификаторов.

2. Сертификация технологических процессов производства сложных программных продуктов

Для сертификации технологических процессов предприятие должно установить ***перечень процессов и документов***, которые необходимы для управления производством программных продуктов, а также для обеспечения уверенности в соответствии продукции требованиям заказчика и стандартам. Головным стандартом, регламентирующим производство, может быть стандарт на ***основе модели — CMMI*** (Capability Maturity Model Integration). Достижение высоких значений качества комплексов программ существенно зависит от ***зрелости*** технологии и инструментальных средств, используемых разработчиками при проектировании и производстве программного продукта. Оценивание уровня зрелости технологической базы жизненного цикла (ЖЦ) позволяет прогнозировать возможное качество продукта и ориентировать заказчика и пользователей при выборе разработчика и поставщика проекта с требуемыми заказчиком характеристиками. По этой причине определение уровня ***зрелости*** технологической поддержки процессов ЖЦ организационного и инструментального обеспечения, непосредственно ***связано с оцениванием реальных или возможных характеристик качества производства*** конкретного программного продукта.

Практически все требования к производству программного продукта в ***модели CMMI соответствуют*** регламентированным и детализированным требованиям в стандартах ISO 9001:2000, ISO 12207 и базовых компонентах ***стандартов жизненного цикла сложных*** комплексов программ. При практической реализации и обеспечении ***всего ЖЦ сложных*** комплексов программ разработчикам и поставщикам целесообразно применять конкретные рекомендации CMMI, стандарта ISO 9001:2000 и ***комплекс базовых международных стандартов*** — рис. 2. На их основе формируются процедуры сертификации производства программных продуктов.

Базовые стандарты организации и управления производством программными продуктами – Руководства по улучшению деятельности – Системы менеджмента качества – **ISO 9000:2000** и версия модели зрелости **CMMI**;

Комплекс адаптированных базовых стандартов для сертификации технологии производства и системы качества предприятия конкретных программных продуктов:

- административного управления системой качества предприятия, разрабатывающего программный продукт – **ISO 90003**;
- управления жизненным циклом программного комплекса – **ISO 12207, ISO 15939; ISO 16326**;
- сопровождения и конфигурационного управления версиями программного продукта – **ISO 14764, ISO 15846**;
- характеристики качества программных продуктов – **ISO 9126, ISO 14598, ISO 25000;**
- организации функциональной безопасности программных продуктов – **IEC 61508, ISO 15408, ISO 13335**;
- документирования программных продуктов – **ISO 6592, ISO 9127, ISO 15910, ISO 18019.**

Рис. 2. Базовые стандарты для сертификации производства программных продуктов

Для сертификации производства и системы менеджмента качества предприятия необходима ***четкая организация документирования производства***. Входные документы для производства должны включать все требования, существенные для проектирования и разработки программного продукта. Выходные данные процесса проектирования и/или производства должны быть зарегистрированы в документах в форме, дающей возможность проверки их по отношению к входным требованиям. Документы, содержащие выходные данные проектирования и производства, должны быть утверждены до их применения при сертификации.

Документ, содержащий ***результаты сертификационных испытаний*** производства и системы менеджмента качества программных продуктов, должен включать Программу и методики сертификационных испытаний — аудита производства предприятия, протоколы и отчет аудиторов о результатах испытаний качества производства программного продукта. В документе должны быть представлены результаты аудита, выводы и рекомендации комиссии, оформленные в виде акта, удостоверяющего качество производства. Завершение сертификации, выдача и регистрация сертификата по результатам аудита — испытаний производства программного

Требования нормативных документов для организации сертифицируемого производства:

- комплект должностных инструкций, определяющих ответственность, полномочия и порядок взаимодействия специалистов коллектива для производства конкретного программного продукта;
- положение о подразделениях и должностные инструкции, обязанности и полномочия специалистов, реализующих процессы производства программных продуктов;
- определение набора характеристик комплекса программ для сертификационных испытаний.

Состав документации процессов и результатов сертификации – аудита технологии и системы качества производства программных продуктов:

- заявка клиента на проведение сертификации производства программного продукта;
- задание клиента на проведение сертификационных испытаний производства программного продукта;
- план сертификации производства программного продукта.

Документы сертификационной лаборатории:

- требования к Программе сертификационных испытаний производства программного продукта;
- Программа испытаний и методики аудита – испытаний процессов производства программного продукта;
- требования к документации результатов испытаний производства программного продукта;
- состав технологических средств автоматизации и порядок сертификационных испытаний производства;
- методики испытаний по каждому разделу требований процессов производства программного продукта;
- отчеты выполнения и результаты испытаний производства;
- заключение по результатам сертификационных испытаний процессов производства программного продукта;
- отчет клиенту-заявителю о проверках организации сертификационных испытаний – аудита и системы качества производства программного продукта.

Рис. 3. Документы организации процессов сертификации производства и системы менеджмента качества предприятия

продукта, должны определять достаточность качества программного продукта для поставки потребителям и обеспечения его ЖЦ, а также для регистрации лицензии на применение знаков соответствия.

Сертификация проектирования и производства программных продуктов высокого качества включает перечисленные далее основные процессы.

2.1. ***Определение конкретной среды, процессов производства и основных характеристик программного продукта.***

2.1.1. Выбор процессов производства, используемых стандартов и технологии, в зависимости от характеристик конкретного проекта.

2.1.2. Подготовка наборов и требований стандартов для сертификации производства программных продуктов.

2.1.3. Анализ и адаптация соответствующих требований и процессов в ЖЦ комплекса программ для удовлетворения конкретным требованиям стандартов производства.

2.2. ***Подготовка к сертификации производства и системы качества программных комплексов и предприятия*** (рис. 3).

2.2.1. Подготовка технологии производства и системы качества к сертификации предприятия программных продуктов.

2.2.2. Адаптация базовых стандартов управления производством ISO 12207 и системой качества ISO 90003 для сертификации программных комплексов.

2.2.3. Определение требования к личным характеристикам, к компетентности, знаниям и навыкам аудиторов производства и систем качества для сертификации программных продуктов.

2.3. Подготовка и документирование организации процессов сертификации производства и системы менеджмента качества предприятия.

2.3.1. Отбор и формирование требований к нормативным документам на сертифицируемое производство.

2.3.2. Подготовка положения о подразделениях, должностных инструкциях, обязанностях и полномочиях специалистов, связанных с реализацией процессов производства.

2.3.3. Подготовка состава стандартов, документации процессов и результатов сертификации — аудита технологии и системы качества производства программных продуктов.

2.3.4. Подготовка документов сертификационной лаборатории, а именно программы испытаний и методик аудита, включая исследования процесса производства, формы отчетности выполнения и анализ результатов испытаний.

2.3.5. Подготовка содержания отчетов организации сертификационных испытаний — аудита процесса производства и системы оценки качества программного продукта.

2.3.6. Передача главным аудитором клиенту-заявителю отчета об организации сертификационных испытаний производства и системы оценки качества программного продукта.

2.4. ***Организация сертификационных испытаний и системы менеджмента управления качеством производства программных продуктов.***

2.4.1. Определение условий, стандартов и документов для обеспечения процессов сертификационных испытаний производства программных продуктов.

Программа и методики сертификационных испытаний – аудита производства и системы менеджмента качества предприятия программного продукта:

- результаты проведения аудита – испытаний на месте производства и подготовка акта аудита предприятия и контроля качества производства программного продукта;
- протоколы и отчет главного аудитора о результатах испытаний качества производства программного продукта;
- результаты оценивания достаточности испытаний качества и возможности завершить процесс тестирования производства.

Результаты аудита, выводы и рекомендации комиссии, оформленные в виде акта удовлетворительного качества производства программного продукта:

- завершение сертификации, выдача и регистрация сертификата по результатам аудита – испытаний производства программного продукта;
- сертификат производства программного продукта, достаточный для поставки потребителям и обеспечения его жизненного цикла, лицензии на применение знаков соответствия;
- результаты эпизодического инспекционного контроля состояния сертифицированного производства программного продукта.

Рис. 4. Результаты сертификационных испытаний проектирования, производства и системы менеджмента качества программных продуктов

2.4.2. Утверждение стандартов, программы и методик испытаний — аудита производства и системы менеджмента качества предприятия.

2.4.3. Организация первоначального заседания комиссии по контролю процессов сертификации и системы менеджмента качества производства программного продукта.

2.4.4. Заочная (по документам) предварительная оценка организации качества производства программного продукта.

2.5. ***Анализ результатов и завершение сертификационных испытаний производства программного продукта*** (рис. 4).

2.5.1. Проведение аудита — полных сертификационных испытаний программного продукта на месте его производства.

2.5.2. Подготовка акта по результатам анализа аудита предприятия и испытаний качества производства программного продукта.

2.5.3. Утверждение результатов аудита, выводов и рекомендаций комиссии, оформленных в виде акта качества проектирования и производства программного продукта на предприятии.

2.5.4. Завершение сертификации, выдача и регистрация сертификата по результатам аудита — испытаний производства программного продукта.

2.5.5. Инспекционный контроль состояния и процесса производства сертифицированных программных продуктов.

3. Сертификация качества готовых программных продуктов

При сертификации программных продуктов и разработке тестов сертификаторы должны иметь четкое ***представление о потребностях заказчика***. Необходимо изучить системные требования, сценарии использования и/или описание назначения подлежащих сертификации продуктов для того, чтобы лучше понять цель их разработки, выбор методов и средств его тестирования. Требования к тестам и документы должны содержать подробный перечень того, что и как должно быть испытано [3, 4]. При систематическом ***восходящем сертификационном тестировании***, прежде всего, проверяется качество программных компонентов нижних иерархических уровней в функциональной группе программ. К ним последовательно подключаются вызывающие их компоненты, что должно быть отражено в соответствующем документе. ***Стратегия сертификационных испытаний представляет собой*** документ, отражающий совокупность выбранных методов, требований и решений, вытекающих из целей, задач проекта и его тестирования, общие правила и принципы, способствующие достижению целей разработки программного продукта высокого качества.

При сертификационных испытаниях программного продукта целесообразно выборочно или полностью использовать результаты предварительных испытаний с учетом их полноты и достоверности. Предварительные испытания разработчиков завершаются предъявлением заказчику на утверждение комплекта документов, содержащих результаты, необходимые для сертификационных испытаний программного продукта. Результатом сертификационных испытаний должен быть комплект отчетных документов и подтверждение результатов, зафиксированных при предварительных испытаниях.

Утверждение этого комплекта документов для конкретного программного продукта дает право на присвоение ему ***сертификата и знака качества***. Отчетный доклад о результатах испытаний должен содержать перечень ***всех неустранимых дефектов***, с соглашением и планом того, будут ли они исправлены в более поздних версиях или их исправление откладывается на неопределенное время.

Снятие с эксплуатации и развития версий сертифицированного программного продукта должно быть подготовлено анализом, обосновывающим это решение. При снятии программного продукта с сопровождения следует определить необходимые для этого действия, а затем разработать и документально оформить этапы работ, обеспечивающие их эффективное выполнение. Должны быть предусмотрены возможности ***доступа к полным архивным документам*** снятого с сопровождения программного продукта.

Сертификация готовых программных продуктов высокого качества включает перечисленные далее основные процессы.

3.1. Формирование требований к характеристикам качества для сертификации программного продукта (рис. 5).

3.1.1. Определение общих задач и требований к качеству функционирования и ограничений ресурсов программного продукта для сертификации.

3.1.2. Формирование требований лиц и организаций, заинтересованных в программном продукте.

3.1.3. Формирование требований к функциям программного продукта и к операционной системе, которые призваны поддерживать его работу в реальном времени.

3.1.4. Формирование требований к надежности функционирования, к функциональной безопасности применения программного продукта, к эффективности динамического использования им ресурсов ЭВМ в реальном времени.

3.1.5. Требования к допустимым рискам динамического применения программного продукта.

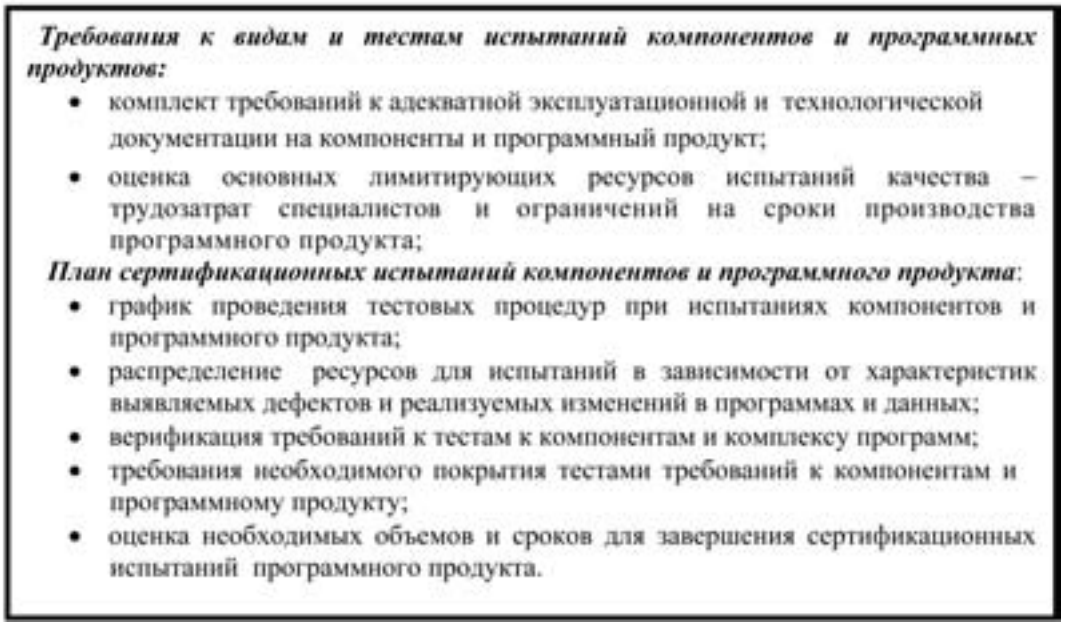

Рис. 5. Организация сертификационных испытаний компонентов и программного продукта на соответствие требованиям

3.2. Оценка процессов и ограничений сертификационных испытаний программных продуктов на соответствие требованиям.

3.2.1. Анализ и формирование требований к видам и тестам сертификационных испытаний программного продукта.

3.2.2. Оценка основных ресурсов, ограничивающих испытания, а именно — трудозатрат специалистов, ограничений на сроки сертификации программного продукта и ряд других.

3.2.3. Определение исходных данных для планирования сертификационных испытаний программного продукта.

3.2.4. Распределение ресурсов для видов и этапов сертификационных испытаний в зависимости от характеристик компонентов и комплексов программ.

3.2.5. Оценивание необходимого объема результатов тестирования, достаточных для завершения процесса сертификационных испытаний программного продукта.

3.3. Организация и планирование сертификационных испытаний программных продуктов и компонентов на соответствие требованиям.

3.3.1. Выбор стратегии сертификационных испытаний, определение требований к тестам программного продукта и компонентов.

3.3.2. Планирование сертификационных испытаний характеристик программного продукта и компонентов.

3.3.3. Разработка графика выполнения тестов для сертификационных испытаний характеристик компонентов и программного продукта.

3.3.4. Верификация тестов и требований к компонентам и к комплексу программ.

3.3.5. Оценка необходимого покрытия тестами требований к характеристикам компонентов и к программному продукту.

3.4. Стратегии испытаний качества программных продуктов (рис. 6).

3.4.1. Выбор стратегии сертификационных испытаний — методов, характеристик решений, способствующих достижению программным продуктом высокого качества.

3.4.2. Выбор процессов сертификационных испытаний для требований к компонентам и к продукту с наивысшим приоритетом, которые наиболее важны пользователям, так как могут причинить наибольшие неприятности.

3.4.3. Выбор видов сертификационных испытаний для новых функций и/или совершенствования характеристик программного продукта.

3.4.4. Выбор функций и конфигураций компонентов, с которыми наиболее часто может иметь дело конечный пользователь или система.

3.5. Подготовка тестов для сертификационных испытаний программных продуктов.

3.5.1. Методы и компоненты определения обобщенных характеристик качества сертифицированного программного продукта.

3.5.2. Методы автоматизированной обработки результатов динамических испытаний обобщенных характеристик программного продукта.

3.5.3. Оценки экономической эффективности и рентабельности сертификационных испытаний качества программных продуктов.

3.6. Предварительные испытания и опытная эксплуатация разработчиками качества программного продукта (рис. 7).

3.6.1. Подготовка полных текстов программ, содержания базы данных, технологической и экс-

плуатационной документации программного продукта для предварительных испытаний.

3.6.2. Разработка программы и методик предварительных испытаний качества программного продукта.

3.6.3. Опытная эксплуатация программного продукта.

3.6.4. Завершение предварительных испытаний и оценка готовности программного продукта к сертификационным испытаниям.

3.7. Завершение сертификационных испытаний и удостоверение качества программных продуктов (рис. 8).

3.7.1. Испытания технологической и эксплуатационной документаций на соответствие требованиям к функциям и характеристикам программного продукта.

3.7.2. Завершение сертификационных испытаний, документирование процессов и результатов сертификации программного продукта.

Компоненты и порядок сертификационных испытаний сложного программного продукта:

- Программа испытаний программного продукта на соответствие утвержденным требованиям;
- методики и протоколы испытаний по разделам Программы и требованиям спецификаций на программный продукт;
- содержание функциональных испытаний для оценивания характеристик и качества программного продукта;
- содержание испытаний надежности функционирования и функциональной безопасности программного продукта:
- оценивание временного баланса между длительностью решения задач и производительностью ЭВМ.

Стратегия выбора тестов для сертификационных испытаний программных модулей и компонентов:

- определение критериев выбора маршрутов для тестирования модулей и программных компонентов;
- определение стратегий упорядочения маршрутов для тестирования модулей и программных компонентов.

Стратегии сертификационных испытаний программных компонентов и продуктов:

- сертификационные испытания на соответствие требованиям к компонентам и программному продукту с наивысшим приоритетом или которые важны заказчику и пользователям;
- сертификационные испытания новых функциональных возможностей в целях исправления или совершенствования функций и характеристик программного продукта;
- сертификационные испытания функций, наиболее значимых для применения системы и/или снижения риска.

Требования к генерации динамических тестов внешней среды в реальном времени:

- исходные данные для генерации тестов внешней среды;
- задачи автоматизированного формирования динамических тестов внешней среды;
- оценка качества программной имитации динамических тестов.

Требования к обработке результатов динамических испытаний программных продуктов в реальном времени:

- средства оперативной обработки результатов динамических испытаний;
- средства и компоненты определения обобщенных характеристик качества программного продукта и системы;
- оценки затрат на программную имитацию динамических тестов;
- оценки экономической эффективности программной имитации динамических тестов.

Рис. 6. Подготовка тестов для сертификационных испытаний сложных программных продуктов

Техническое задание – требования к функциям, характеристикам, качеству программного продукта, системы и внешней среды:

Договор заказчика с производителем на качество программного продукта:

- описание целей, требований и обязательств производителя качества программного продукта;
- полные тексты программ, содержание базы данных и технологической документации программного продукта;
- результаты испытания эксплуатационной документации на соответствие требованиям к программному продукту;
- комплект эксплуатационных документов, поставляемых заказчику и пользователям для применения программного продукта;
- план, Программа и методики испытаний, применения и оценки качества программного продукта;
- методики конфигурационного управления, утверждения, хранения, защиты, копирования программного продукта и сопровождающих документов в архиве предприятия;

Технические условия на программный продукт, базу данных управления конфигурацией и эксплуатационную документацию для тиражирования и серийного производства:

- результаты опытной эксплуатации программного продукта;
- руководство по генерации и инсталляции пользовательских версий и загрузке базы данных в соответствии с условиями и характеристиками внешней среды;
- проект акта о завершении предварительных испытаний и готовности программного продукта к поставке и/или предъявлению для завершения и утверждения сертификационных испытаний.

Рис. 7. Результаты предварительных испытаний разработчиков и опытной эксплуатации программного продукта

Договор заявителя с сертифицирующей организацией на проведение испытаний версии программного продукта:

- отчет заявителя о наличии, актуальности и систематичности тестирования, оформлении программного продукта и документации на протяжении жизненного цикла программного продукта;
- отчет сертификаторов о реализации программы и методик проведения сертификационных испытаний программного продукта в соответствии с требованиями Договора на сертификацию с заявителем;
- результаты аттестации имитаторов внешней среды и генераторов динамических тестов для сертификационных испытаний программного продукта;
- результаты выполнения планов и методик сертификационных испытаний, протоколы соответствия испытаний предъявляемым требованиям, утвержденным сертификаторами и согласованным с заявителями;
- протоколы достигнутых при сертификационных испытаниях характеристик качества программного продукта.

Акт результатов сертификационных испытаний:

- реальные характеристики программного продукта, выводы о их соответствии требованиям к характеристикам заказчика программного продукта;
- сертификат программного продукта и обеспечения его жизненного цикла, лицензия на применение знаков соответствия;
- удостоверение для поставки и применения пользователями сертифицированного программного продукта.

Рис. 8. Результаты сертификационных испытаний программного продукта

3.7.3. Формирование отчетных документов и завершающего Акта испытателей, отражающих полные результаты сертификационных испытаний программного продукта.

3.7.4. Удостоверение допустимости поставки сертифицированного программного продукта для применения пользователями.

3.7.5. Анализ процессов и результатов сертификации, усовершенствование методов и достоверности качества испытаний программного продукта.

Заключение

Современные комплексы программ для управления и обработки информации активно применяются в сложных, практически значимых, в том числе — критически важных системах динамического управления объектами различного назначения. Ущерб от дефектов программных продуктов таких систем может определяться их огромной стоимостью и даже жизнью пользователей. На реализацию необходимого для этого качества требуются ресурсы, которые могут быть соизмеримы с первичными затратами на весь процесс проектирования и производства программного продукта, а также на подготовку и воспитание профессиональных квалифицированных коллективов специалистов в области индустрии сложных программных продуктов высокого качества. Для достижения и удостоверения такого качества необходимо создание и внедрение методов и средств автоматизации сертификации производственных процессов и их результатов — программных продуктов сложных систем. На настоящее время реализация такого подхода является настоятельной необходимостью и одной из важнейших задач применения программной инженерии по заказам государства.

Список литературы

1. **Липаев В. В.** Сертификация программных средств. Учебник. М.: СИНТЕГ. 2010. 344 с.
2. **Фатрелл Р. Т., Шафер Д. Ф., Шафер Л. И.** Управление программными проектами: достижение оптимального качества при минимальных затратах. Пер. с англ. М.: Вильямс, 2003. 1136 с.
3. **Липаев В. В.** Тестирование компонентов и комплексов программ. Учебник. М.: СИНТЕГ, 2010. 400 с.
4. **Дастин Э., Рэшка Д., Пол Д.** Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация. Пер с англ. М.: ЛОРИ, 2003. 568 с.

УДК 004.054; 519.257

С. В. Орлик, эксперт по архитектуре, Microsoft Россия,
e-mail: sorlik@gmail.com

Программная инженерия как дисциплина и роль SWEBOK в ее развитии. Часть II

Обсуждаются концептуальные основы жизненного цикла проектов по разработке программного обеспечения и программной инженерии в контексте IEEE Guide to the Software Engineering Body of Knowledge — SWEBOK. Рассмотрены и систематизированы различные модели жизненного цикла программных проектов. Данная работа основана на публично доступном переводе SWEBOK на русский язык, подготовленном автором.

Ключевые слова: *SWEBOK, программная инженерия, модели жизненного цикла программных проектов*

Одним из ключевых понятий управления проектами, в том числе в приложении к индустрии программного обеспечения является *жизненный цикл проекта (Project Lifecycle Management — PLM)*. В силу высокой практической значимости тематики и накопленного в индустрии опыта в части успешно зарекомендовавших процессов разработки ПО, а также разработки соответствующих стандартов, рассматриваемых и активно упоминаемых практически во всех областях знаний SWEBOK, этой теме хочется уделить особое внимание. Далее представлены основные идеи управления жизненным циклом разработки программного обеспечения как неотъемлемой части программной инженерии, отраженные как в стандарте ГОСТ Р/ИСО МЭК 12207, так и в спиральной модели жизненного цикла.

Жизненный цикл проекта по разработке программного обеспечения

Известный эксперт по управлению высокотехнологичными проектами Арчибальд так определяет жизненный цикл проекта в работах [1, 2]: "Жизненный цикл проекта имеет определенные начальную и конечную точки, привязанные к временной шкале. Проект в своем естественном развитии проходит ряд отдельных фаз.

Жизненный цикл проекта включает все фазы от момента инициации до момента завершения. Переходы от одного этапа к другому редко четко определены, за исключением тех случаев, когда они формально разделяются принятием предложения или получением разрешения на продолжение работы. Однако в начале концептуальной фазы часто возникают сложности с точным определением момента, когда работу можно уже идентифицировать как проект (в терминах управления проектами), особенно если речь идет о разработке нового продукта или новой услуги.

Существует общее соглашение о выделении четырех обобщенных фаз жизненного цикла (в скобках приведены используемые в различных источниках альтернативные термины):

- концепция (инициация, идентификация, отбор);
- определение (анализ);
- выполнение (практическая реализация или внедрение, производство и развертывание, проектирование или конструирование, сдача в эксплуатацию, инсталляция, тестирование и т. п.);
- закрытие (завершение, включая оценивание после завершения).

Однако эти фазы столь широки, что ... необходимы конкретные определения, быть может пяти-десяти основных фаз для каждой категории и подкатегории проекта, обычно с несколькими подфазами, выделяемыми внутри каждой из этих фаз.

...Нередко можно наблюдать частичное совмещение или одновременное выполнение фаз проекта, называемое "быстрым проходом" в строительных и инжиниринговых проектах и "параллелизмом" — в военных и аэрокосмических. Это усложняет планирование проекта и координацию усилий его участников, а также делает более важной роль менеджера проектов".

В общем случае, *жизненный цикл определяется моделью и описывается в форме методологии (метода). Модель или парадигма* жизненного цикла определяет концептуальный взгляд на организацию жизненного цикла и часто основные фазы жизненного цикла, а также принципы перехода между ними. *Методология (метод)* задает комплекс работ, их детальное содержание и ролевую ответ-

ственность специалистов на всех этапах выбранной модели жизненного цикла, обычно определяет и саму модель, а также рекомендует *практики (best practices)*, позволяющие максимально эффективно воспользоваться соответствующей методологией и ее моделью.

В индустрии программного обеспечения возможно (так как это уже конкретная область приложения концепций и практик проектного управления) и необходимо (для обеспечения возможности управления) более четкое разграничение фаз проекта (что не подразумевает их линейного и последовательного выполнения).

Ниже приведены определения модели жизненного цикла программной системы, даваемые, например, в различных вариантах стандартов ГОСТ:

- Модель жизненного цикла — структура, состоящая из процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение программного продукта, охватывающая жизнь системы от установления требований к ней до прекращения ее использования [3].
- Жизненный цикл автоматизированной системы (АС) — совокупность взаимосвязанных процессов создания и последовательного изменения состояния АС от формирования исходных требований к ней до окончания эксплуатации и утилизации комплекса средств автоматизации АС [4].

ГОСТ Р ИСО/МЭК 12207 является переводом международного стандарта ISO/IEC 12207, на основе которого, в свою очередь, создан соответствующий стандарт IEEE 12207. Другой ГОСТ в рамках семейства ГОСТ 34 разрабатывался в СССР самостоятельно, как стандарт на содержание и оформление документов на программные системы в рамках Единой системы программной документации (ЕСПД) и Единой системы конструкторской документации (ЕСКД). В последние годы акцент делается на стандарты ГОСТ, соответствующие международным стандартам. В то же время, 34-я серия является важным дополнительным источником информации для разработки и стандартизации внутрикорпоративных документов и формирования целостного понимания и видения концепций жизненного цикла в области программного обеспечения.

В определенном контексте понятия "модель" и "методология" могут использоваться взаимозаменяемым образом, например, когда мы обсуждаем разграничение фаз проекта. Говоря "жизненный цикл" мы, в первую очередь, подразумеваем "модель жизненного цикла". Несмотря на данное в стандартах 12207 определение модели жизненного цикла, все же, *модель* чаще подразумевает именно *общий принцип* организации жизненного цикла, чем детализацию соответствующих работ. Соответственно, определение и выбор модели, в первую очередь, касаются вопросов определенности и стабильности требований, жесткости и детализированности плана работ, а также частоты сборки работающих версий создаваемой программной системы.

Скотт Амблер (*Scott W. Ambler)* [5], автор концепций и практик гибкого моделирования (*Agile Modeling*) и *Enterprise Unified Process* (расширение *Rational Unified Process*), предлагает следующие уровни жизненного цикла, определяемые соответствующим содержанием работ (рис. 1):

Рис. 1. Содержание четырех категорий жизненного цикла по Амблеру [5]

- жизненный цикл разработки программного обеспечения — проектная деятельность по разработке и развертыванию программных систем;
- жизненный цикл программной системы, включающий разработку, развертывание, поддержку и сопровождение;
- жизненный цикл информационных технологий (ИТ), включающий всю деятельность ИТ-департамента;
- жизненный цикл организации/бизнеса, охватывающий всю деятельность организации в целом.

В данном контексте SWEBOK описывает области знаний *жизненного цикла системы* и *жизненного цикла разработки* программного обеспечения. В свою очередь, как упоминается в SWEBOK, одними из фундаментальных взглядов на жизненный цикл являются стандарты процессов жизненного цикла ISO/IEC, IEEE, ГОСТ Р ИСО/МЭК 12207.

Стандарт 12207: Процессы жизненного цикла программного обеспечения

В 1997 г. Международная Организация по Стандартизации — ИСО (*International Organization for Standardization — ISO*) и Международная Электротехническая Комиссия — МЭК (*International Electrotechnical Commission — IEC*) создали Совместный Технический Комитет по Информационным Технологиям — *Joint Technical Committee* (JTC1) *on Information Technology*. Содержание работ JTC1 определено как "стандартизация в области систем и оборудования информационных технологий (включая микропроцессорные системы)". В 1989 г. этот комитет инициировал разработку стандарта ISO/IEC 12207, создав для этого подкомитет SC7 (*SuCommittee* 7) по программной инженерии. Соответствующий стандарт впервые был опубликован 1 августа 1995 г. под заголовком *"Software Life Cycle Processes"* — "Процессы жизненного цикла

программного обеспечения". Национальный стандарт [3] получил название "Процессы жизненного цикла программных средств".

Цель разработки данного стандарта была определена как создание общего фреймворка по организации жизненного цикла программного обеспечения для формирования общего понимания жизненного цикла ПО всеми заинтересованными сторонами и участниками процесса разработки, приобретения, поставки, эксплуатации, поддержки и сопровождения программных систем, а также возможности управления, контроля и совершенствования процессов жизненного цикла.

Данный стандарт определяет жизненный цикл как структуру декомпозиции работ. Детализация, техники и метрики проведения работ — вопрос программной инженерии. Организация последовательности работ — модель жизненного цикла. Совокупность моделей, процессов, техник и организации проектной группы задаются методологией. В частности, выбор и применение метрик оценки качества программной системы и процессов находятся за рамками стандарта 12207, а концепция совершенствования процессов рассматривается в стандарте ISO/IEC 15504 *"Information Technology — Software Process Assessment"* ("Оценка процессов в области программного обеспечения").

Необходимо отметить заложенные в стандарте ключевые концепции рассмотрения жизненного цикла программных систем.

Организация стандарта 12207 и архитектура жизненного цикла

Стандарт определяет область применения, дает ряд важных определений (таких, как заказчик, разработчик, договор, оценка, выпуск — релиз, программный продукт, аттестация и т. п.), процессов жизненного цикла и включает ряд примечаний по процессу и вопросам адаптации стандарта.

Стандарт описывает 17 процессов жизненного цикла, распределенных по трем категориям — группам процессов (названия представлены с указанием номеров разделов стандарта, следуя определениям на русском и английском языке, определяемыми в [3] и оригинальной версией ISO/IEC 12207, соответственно):

5. Основные процессы жизненного цикла — Primary Processes.

5.1. Заказ — *Acquisition.*

5.2. Поставка — *Supply.*

5.3. Разработка — *Development.*

5.4. Эксплуатация — O*peration.*

5.5. Сопровождение — *Maintenance.*

6. Вспомогательные процессы жизненного цикла — *Supporting Processes.*

6.1. Документирование — *Documentation.*

6.2. Управление конфигурацией — *Configuration Managemen*t.

6.3. Обеспечение качества — *Quality Assurance.*

6.4. Верификация — *Verification.*

6.5. Аттестация — *Validation.*

6.6. Совместный анализ — *Joint Review.*

6.7. Аудит — *Audit.*

6.8. Решение проблем — *Problem Resolution.*

7. Организационные процессы жизненного цикла — *Organizational Processes.*

7.1. Управление — *Management.*

7.2. Создание инфраструктуры — *Infrastructure.*

7.3. Усовершенствование — *Improvement.*

7.4. Обучение — *Training.*

Стандарт определяет высокоуровневую архитектуру жизненного цикла. Жизненный цикл начинается с идеи или потребности, которую необходимо удовлетворить с использованием программных средств (может быть и не только их). Архитектура строится как набор процессов и взаимных связей между ними. Например, основные процессы жизненного цикла обращаются к вспомогательным процессам, в то время, как организационные процессы действуют на всем протяжении жизненного цикла и связаны с основными процессами.

Дерево процессов жизненного цикла представляет собой структуру декомпозиции жизненного цикла на соответствующие процессы (группы процессов). Декомпозиция процессов строится на основе следующих двух важнейших принципов, определяющих правила разбиения (*partitioning*) жизненного цикла на составляющие процессы.

- Модульность:
  - задачи в процессе являются функционально связанными;
  - связь между процессами минимальна;
  - если функция используется более, чем одним процессом, она сама является процессом;
  - если процесс *Y* используется процессом *X* и только им, значит, процесс *Y* принадлежит (является его частью или его задачей) процессу *X*, за исключением случаев потенциального использования процесса *Y* в других процессах в будущем.
- Ответственность:
  - каждый процесс находится под ответственностью конкретного лица (управляется и/или контролируется им), определенного для заданного жизненного цикла, например, в виде роли в проектной команде;
  - функция, чьи части находятся в компетенции различных лиц, не может рассматриваться как самостоятельный процесс.

Общая иерархия (декомпозиция) составных элементов жизненного цикла выглядит следующим образом:

- группа процессов
  - процессы
    - работы
      - задачи.

В общем случае, разбиение процесса базируется на широко распространенном PDCA-цикле:

- "P"— *Plan* — Планирование;
- "D" — *Do* — Выполнение;
- "C" — *Check* — Проверка;
- "A" — *Act* — Реакция (действие).

Модели жизненного цикла

Наиболее часто говорят о следующих моделях жизненного цикла:

- каскадная (водопадная) или последовательная;
- итеративная и инкрементальная — эволюционная (гибридная, смешанная);
- спиральная (*spiral*) или модель Боэма.

Легко обнаружить, что в разное время и в разных источниках приводится разный список моделей и их интерпретация. Например, ранее инкрементальная модель понималась как построение системы в виде последовательности сборок (релизов), определенной в соответствии с *заранее подготовленным планом* и заданными (уже сформулированными) и *неизменными требованиями*. Сегодня об инкрементальном подходе чаще всего говорят в контексте постепенного наращивания функциональности создаваемого продукта.

Может показаться, что индустрия пришла, наконец, к общей "правильной" модели. Однако каскадная модель, многократно "убитая" и теорией и практикой, продолжает встречаться в реальной жизни, и в ряде случаев ее использование представляется обоснованным, как например для разработки критически важных приложений и систем (*mission-critical*). Спиральная модель является ярким представителем эволюционного взгляда, но в то же время представляет собой единственную модель, которая уделяет явное внимание *анализу и предупреждению рисков*.

Каскадная (водопадная) модель

Данная модель предполагает строго последовательное (во времени) и однократное выполнение всех фаз проекта с жестким (детальным) предварительным планированием в контексте предопределенных или однажды и целиком определенных требований к программной системе.

На рис. 2 изображены типичные фазы каскадной модели жизненного цикла и соответствующие активы проекта, являющиеся для одних фаз выходами, а для других — входами. Марри Кантор [6] отмечает ряд важных аспектов, характерных для водопадной модели: "Водопадная схема включает несколько важных операций, применимых ко всем проектам:

- составление плана действий по разработке системы;
- планирование работ, связанных с каждым действием;
- применение операции отслеживания хода выполнения действий с контрольными этапами.

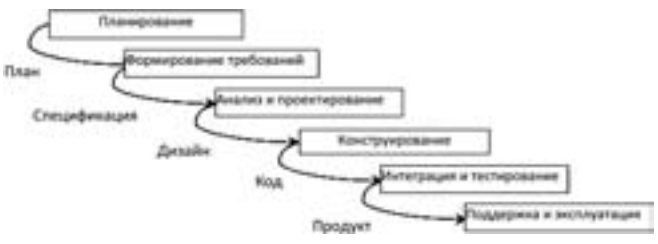

Рис. 2. Каскадная модель жизненного цикла

В связи с тем, что упомянутые задачи являются неотъемлемым элементом всех хорошо управляемых процессов, практически не существует причин, препятствующих утверждению полнофункциональных, классических методов руководства проектом, таких как анализ критического пути и промежуточные контрольные этапы. Я часто встречался с программными менеджерами, которые ломали себе голову над тем, почему же столь эффективный набор методик на практике оборачивается неудачей...".

Будучи активно используема (де-факто и, например, в свое время, как часть соответствующего отраслевого стандарта в США), эта модель продемонстрировала свою "проблемность" в подавляющем большинстве ИТ-проектов, за исключением, может быть, отдельных проектов обновления программных систем для критически важных программно-аппаратных комплексов (например, авионики или медицинского оборудования). Практика показывает, что в реальном мире, особенно в мире бизнес-систем, каскадная модель не должна применяться. Специфика таких систем (если можно говорить о "специфике" для подавляющего большинства создаваемых систем) — требования характеризуются высокой динамикой корректировки и уточнения, невозможностью четкого и однозначного определения требований до начала работ по реализации (особенно для новых систем) и быстрой изменчивостью в процессе эксплуатации системы.

Фредерик Брукс во втором издании своего классического труда "Мифический человеко-месяц" описывает главную беду каскадной модели следующим образом [7].

"Основное заблуждение каскадной модели состоит в предположениях, что проект проходит через весь процесс *один раз*, архитектура хороша и проста в использовании, проект осуществления разумен, а ошибки в реализации устраняются по мере тестирования. Иными словами, каскадная модель исходит из того, что все ошибки будут сосредоточены в реализации, а потому их устранение происходит равномерно во время тестирования компонентов и системы".

В каскадной модели переход от одной фазы проекта к другой предполагает полную корректность результата (выхода) предыдущей фазы. Однако неточность например, какого-либо требования или некорректная его интерпретация, в результате приводит к тому, что приходится "откатываться" к ранней фазе проекта и требуемая переработка не просто выбивает проектную команду из графика, но часто приводит к качественному росту затрат и, не исключено, к прекращению проекта в той форме, в которой он изначально задумывался. Кроме того, эта модель не способна гарантировать необходимую скорость отклика и внесение соответствующих изменений в ответ на быстро меняющиеся потребности пользователей, для которых программная система является одним из инструментов исполнения бизнес-функций. И таких примеров проблем, порождаемых самой природой модели, можно привести достаточно много для того, чтобы отказаться от каскадной модели жизненного цикла.

Итеративная и инкрементальная модель — эволюционный подход

Итеративная модель предполагает *разбиение жизненного цикла проекта на последовательность итераций*, каждая из которых напоминает "мини-проект", включая все фазы жизненного цикла в применении к созданию меньших фрагментов функциональности по сравнению с проектом в целом. Цель каждой итерации – получение работающей версии программной системы, включающей функциональность, определенную интегрированным содержанием всех предыдущих и текущей итерации. Результат финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации *продукт развивается инкрементально*.

С точки зрения структуры жизненного цикла такую модель называют *итеративной (iterative)*. С точки зрения развития продукта — *инкрементальной (incremental)*. Опыт индустрии показывает, что невозможно рассматривать каждый из этих взглядов изолированно. Чаще всего такую *смешанную эволюционную модель* называют просто итеративной (говоря о процессе) и/или инкрементальной (говоря о наращивании функциональности продукта).

Эволюционная модель подразумевает не только сборку работающей (с точки зрения результатов тестирования) версии системы, но и ее развертывание в реальных операционных условиях с анализом откликов пользователей для определения содержания и планирования следующей итерации. "Чистая" инкрементальная модель не предполагает развертывания промежуточных сборок (релизов) системы и все итерации проводятся по заранее определенному плану наращивания функциональности, а пользователи (заказчик) получают только результат финальной итерации как полную версию системы. Вместе с тем, С. Амблер [8], например, определяет эволюционную модель как сочетание итеративного и инкрементального подходов. В свою очередь, М. Фаулер [9] пишет: "Итеративную разработку называют по-разному: инкрементальной, спиральной, эволюционной и постепенной. Разные люди вкладывают в эти термины разный смысл, но эти различия не имеют широкого признания и не так важны, как противостояние итеративного метода и метода водопада".

Ф. Брукс пишет [7], что в идеале, поскольку на каждом шаге мы имеем работающую систему:

- можно очень рано начать тестирование пользователями;
- можно принять стратегию разработки в соответствии с бюджетом, полностью защищающую от перерасхода времени или средств (в частности, за счет сокращения второстепенной функциональности).

Таким образом, значимость эволюционного подхода на основе организации итераций особенно проявляется в снижении неопределенности с завершением каждой итерации. В свою очередь, снижение неопределенности позволяет уменьшить риски. Рис. 3 иллюстрирует некоторые идеи эволюционного подхода, предполагая, что итеративному разбиению может

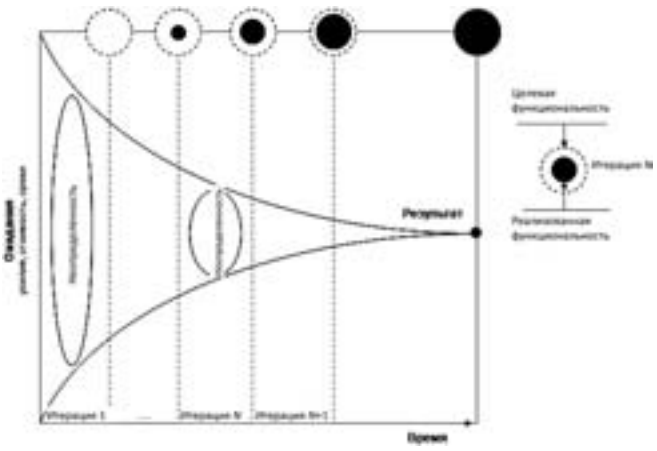

Рис. 3. Снижение неопределенности и инкрементальное расширение функциональности при итеративной организации жизненного цикла

быть подвержен не только жизненный цикл в целом, включающий перекрывающиеся фазы — формирование требований, проектирование, конструирование и т. п., но и каждая фаза в свою очередь может быть разбита на уточняющие итерации, связанные, например, с детализацией структуры декомпозиции проекта — например архитектуры модулей системы.

Наиболее известным и распространенным вариантом эволюционной модели является *спиральная модель*, ставшая уже самостоятельной моделью, имеющей различные сценарии развития и детализации.

Спиральная модель

Спиральная модель (рис. 4) была впервые сформулирована Барри Боэмом (*Barry Boehm*) в 1988 г. [10]. Отличительной особенностью этой модели является специальное внимание *рискам*, влияющим на организацию жизненного цикла.

Боэм формулирует "top-10" наиболее распространенных (по приоритетам) рисков (используется с разрешения автора):

1. Дефицит специалистов.
2. Нереалистичные сроки и бюджет.
3. Реализация несоответствующей функциональности.
4. Разработка неправильного пользовательского интерфейса.
5. "Золотая сервировка", перфекционизм, ненужная оптимизация и оттачивание деталей.
6. Непрекращающийся поток изменений.
7. Нехватка информации о внешних компонентах, определяющих окружение системы или вовлеченных в интеграцию.
8. Недостатки в работах, выполняемых внешними (по отношению к проекту) ресурсами.
9. Недостаточная производительность получаемой системы.
10. "Разрыв" в квалификации специалистов разных областей знаний.

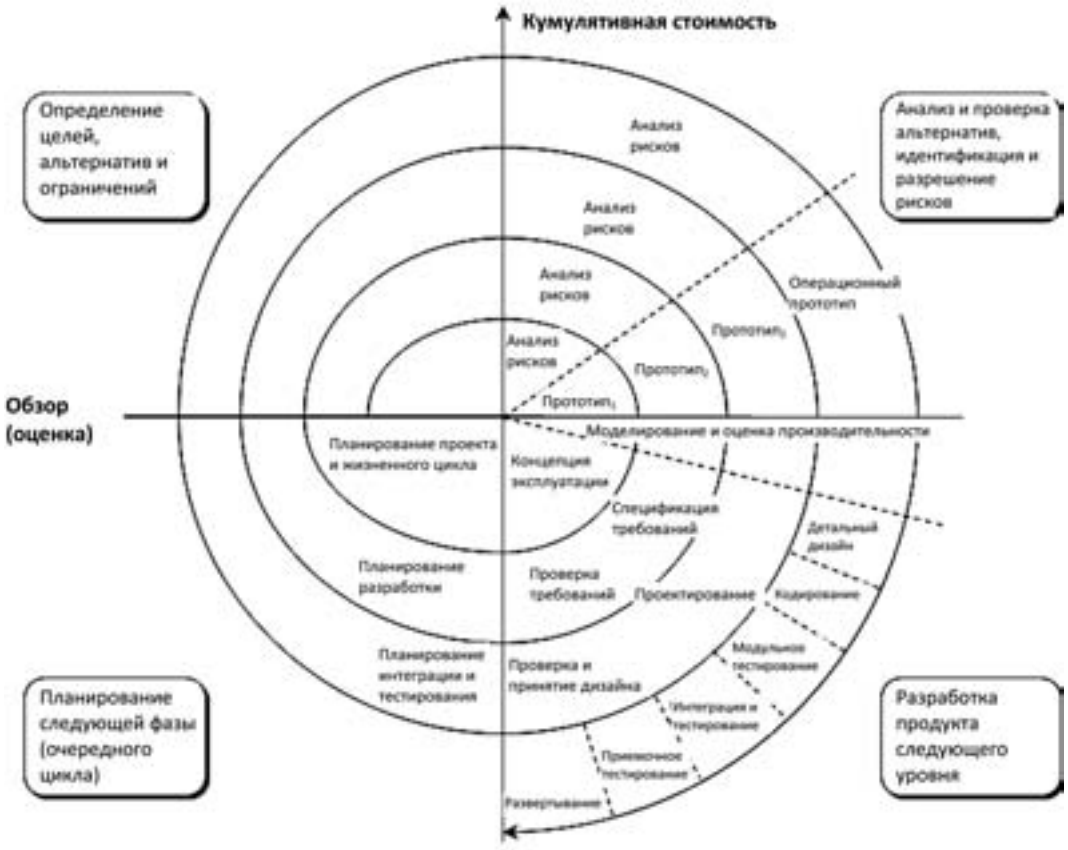

Рис. 4. Оригинальная спиральная модель жизненного цикла разработки по Боэму [10]

Большая часть этих рисков связана с организационными и процессными аспектами взаимодействия специалистов в проектной команде.

Сам Барри Боэм так характеризует спиральную модель разработки: "Главное достижение спиральной модели состоит в том, что она предлагает спектр возможностей адаптации удачных аспектов существующих моделей процессов жизненного цикла. В то же время, ориентированный на риски подход позволяет избежать многих сложностей, присутствующих в этих моделях. В определенных ситуациях спиральная модель становится эквивалентной одной из существующих моделей. В других случаях она обеспечивает возможность наилучшего соединения существующих подходов в контексте данного проекта.

Спиральная модель обладает рядом преимуществ:

Модель уделяет специальное внимание раннему анализу возможностей повторного использования. Это обеспечивается, в первую очередь, в процессе идентификации и оценки альтернатив.

Модель предполагает возможность эволюции жизненного цикла, развитие и изменение программного продукта. Главные источники изменений заключены в целях, для достижения которых создается продукт. Подход, предусматривающий скрытие информации о деталях на определенном уровне дизайна, позволяет рассматривать различные архитектурные альтернативы так, как если бы мы говорили о единственном проектном решении, что уменьшает риск невозможности согласования функционала продукта и изменяющихся целей (требований).

Модель предоставляет механизмы достижения необходимых параметров качества как составную часть процесса разработки программного продукта. Эти механизмы строятся на основе идентификации всех типов целей (требований) и ограничений на всех "циклах" спирали разработки. Например, ограничения по безопасности могут рассматриваться как риски на этапе специфицирования требований.

Модель уделяет специальное внимание предотвращению ошибок и отбрасыванию ненужных, необоснованных или неудовлетворительных альтернатив на ранних этапах проекта. Это достигается явно определенными работами по анализу рисков, проверке различных характеристик создаваемого продукта (включая архитектуру, соответствие требованиям и т. п.) и подтверждение возможности двигаться дальше на каждом "цикле" процесса разработки.

Модель позволяет контролировать источники проектных работ и соответствующих затрат. По сути, речь идет об ответе на вопрос — как много усилий необходимо затратить на анализ требований, планирование, конфигурационное управление, обеспечение качества, тестирование, формальную верификацию и т. д. Модель, ориентированная на риски, позволяет в контексте конкретного проекта решить задачу приложения адекватного уровня

усилий, определяемого уровнем рисков, связанных с недостаточным выполнением тех или иных работ.

Модель не проводит различий между разработкой нового продукта и расширением (или сопровождением) существующего. Этот аспект позволяет избежать часто встречающегося отношения к поддержке и сопровождению как ко "второсортной" деятельности. Такой подход предупреждает большое количество проблем, возникающих в результате одинакового уделения внимания как обычному сопровождению, так и критичным вопросам, связанным с расширением функциональности продукта, всегда ассоциированным с повышенными рисками.

Модель позволяет решать интегрированные задачи системной разработки, охватывающей и программную и аппаратную составляющие создаваемого продукта. Подход, основанный на управлении рисками и возможности своевременного отбрасывания непривлекательных альтернатив (на ранних стадиях проекта) сокращает расходы и одинаково применим и к аппаратной части, и к программному обеспечению".

Описывая созданную спиральную модель, Боэм обращает внимание на то, что обладая явными преимуществами по сравнению с другими взглядами на жизненный цикл, необходимо уточнять, детализировать шаги, т. е. циклы спиральной модели для обеспечения целостного контекста для всех лиц, вовлеченных в проект.

Организация ролей (ответственности членов проектной команды), детализация этапов жизненного цикла и процессов, определение активов (артефактов), значимых на разных этапах проекта, практики анализа и предупреждения рисков — все это вопросы уже конкретного процессного фреймворка или, как принято говорить, *методологии разработки.*

Действительно, детализация процессов, ролей и активов — вопрос методологии. Однако, рассматривая (спиральная) модель разработки, являясь концептуальным взглядом на создание продукта, требует, как и в любом проекте, *определения ключевых контрольных точек проекта — milestones.* Это в большой степени связано с попыткой ответить на вопрос "где мы?". Вопрос, особенно актуальный для менеджеров и лидеров проектов, отслеживающих ход их выполнения и планирующих дальнейшие работы.

В 2000 г. [11], представляя анализ использования спиральной модели и, в частности, построенного на его основе подхода MBASE — *Model-Based (System) Architecting and Software Engineering*, Боэм формулирует 6 ключевых характеристик (или практик), обеспечивающих успешное применение спиральной модели:

1. Параллельное, а не последовательное определение артефактов (активов) проекта.

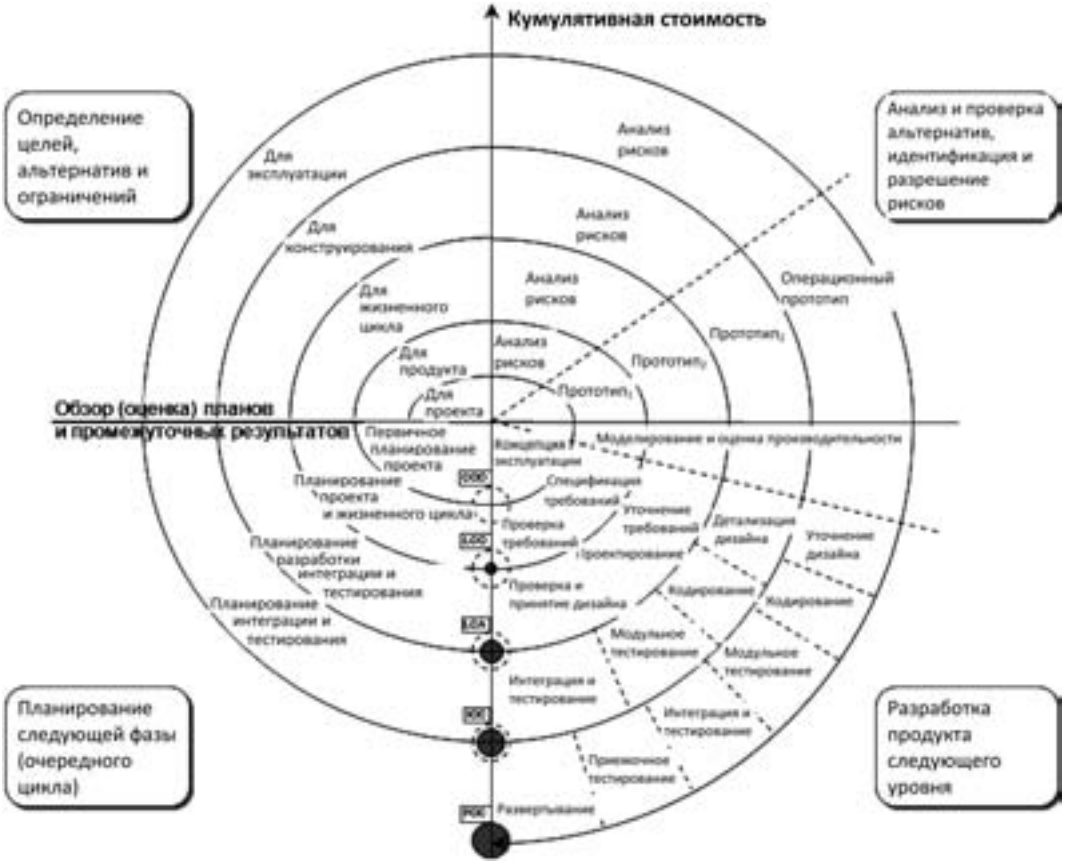

Рис. 5. Обновленная спиральная модель с контрольными точками проекта (данное представление создано автором на основе оригинальной модели Боэма и ее модификациях)

2. Согласие в том, что на каждом цикле уделяется внимание:

- целям и ограничениям, важным для заказчика;
- альтернативам организации процесса и технологических решений, закладываемых в продукт;
- идентификации и разрешению рисков;
- оценке со стороны заинтересованных лиц (в первую очередь заказчика);
- достижению согласия в том, что можно и необходимо двигаться дальше.

3. Использование соображений, связанных с рисками, для определения уровня усилий, необходимого для каждой работы на всех циклах спирали.

4. Использование соображений, связанных с рисками, для определения уровня детализации каждого артефакта, создаваемого на всех циклах спирали.

5. Управление жизненным циклом в контексте обязательств всех заинтересованных лиц на основе трех контрольных точек:

- *Life Cycle Objectives* (LCO);
- *Life Cycle Architecture* (LCA);
- *Initial Operational Capability* (IOC).

6. Уделение повышенного внимания проектным работам и артефактам создаваемой системы (включая непосредственно разрабатываемое программное обеспечение, окружение системы, а также ее эксплуатационные характеристики) и жизненного цикла (разработки и использования).

Эволюционирование спиральной модели, таким образом, связано с вопросами детализации работ. Особенно стоит выделить повышенное внимание к вопросам уточнения — требований, дизайна и кода, т. е. придание большей важности вопросам итеративности, в том числе, увеличения их числа при сокращении длительности каждой итерации.

В результате, можно определить общий набор контрольных точек в сегодняшней спиральной модели:

- *Concept of Operations* (COO) — концепция использования системы;
- *Life Cycle Objectives* (LCO) — цели и содержание жизненного цикла;
- *Life Cycle Architecture* (LCA) — архитектура жизненного цикла, здесь же можно говорить о готовности концептуальной архитектуры целевой программной системы;
- *Initial Operational Capability* (IOC) — первая версия создаваемого продукта, пригодная для опытной эксплуатации;
- *Final Operational Capability* (FOC) — готовый продукт, развернутый (установленный и настроенный) для реальной эксплуатации.

Таким образом, приходим к возможному современному взгляду (см., например, представление спиральной модели в работе [12] на итеративный и инкрементальный — эволюционный жизненный цикл в форме спиральной модели, изображенной на рис. 5).

Похоже, нам удалось более четко и естественно определить контрольные точки проекта, в определенной степени подчеркнув эволюционную природу жизненного цикла. Теперь же пора взглянуть на жизненный цикл в контексте методологий, не просто детализирующих ту или иную модель, но и добавляющих к ним ключевой элемент — *людей*. *Роли*, как представление различных функциональных групп работ, связывают создание, модификацию и использование активов проектов с конкретными участниками проектных команд. В совокупности с *процессами* и *активами* (*артефактами*) они позволяют нам создать целостную и подробную картину жизненного цикла.

Так как взглядов на детализацию описания жизненного цикла может быть много, существуют различные методологии, среди которых наибольшее распространение получили:

- *Rational Unified Process* (RUP).
- *Enterprise Unified Process* (EUP).
- *Microsoft Solutions Framework* (MSF) в обоих представлениях: MSF for *Agile* и MSF for CMMI (анонсированная изначально как "MSF *Formal*").
- *Agile*-практики (*eXtreme Programming* (XP), *Feature Driven Development* (FDD), *Dynamic Systems Development Method* (DSDM), SCRUM, ...).

При этом, подавляющее большинство используемых в индустрии методологий основывается на эволюционном подходе, представленном в спиральной модели, на сегодняшний день наиболее полно отражающей и инкрементальную природу разработки программного обеспечения как систематической инженерной деятельности.

Список литературы

1. **Арчибальд Р. Д.** Управление высокотехнологичными программами и проектами. Изд. третье, перераб. и доп. / Пер. на рус. М.: АйТи — ДМК Пресс, 2004.

2. **Арчибальд Р. Д.** Искусство управления проектами: состояние и перспективы. Возможности и уровень зрелости организации в управлении проектами // Управление проектами. 2005. № 1. С. 14—23. URL: http://www.pmmagazine.ru

3. **Информационная** технология. Процессы Жизненного Цикла Программных Средств: ГОСТ Р ИСО/МЭК 12207—99. М.: Госстандарт России, 2000.

4. **Информационная** технология. Комплекс стандартов и руководящих документов на автоматизированные системы. Термины и определения. ГОСТ 34.003—90. М.: Госстандарт России, 1990.

5. **Ambler S. W., Nalbone J., Vizdos M. J.** The Enterprise Unified Process: Extending the Rational Unified Process. Prentice Hall PTR, 2005.

6. **Кантор М.** Управление программными проектами. Практическое руководство по разработке успешного программного обеспечения. М.: Вильямс, 2002.

7. **Брукс Ф.** Мифический человеко-месяц или как создаются программные системы. 2-е издание, юбилейное. СПб.: Символ-Плюс, 2000, 2005.

8. **Ambler S. W.** One Piece at a Time // Software Development Magazine. 2004. December. URL: http://www.sdmagazine.com/

9. **Фаулер М.** UML. Основы. Краткое руководство по стандартному языку объектного моделирования. 3-е издание. СПб.: Символ-Плюс, 2004.

10. **Boehm B. W.** A Spiral Model of Software Development and Enhancement // Computer. 1988. May. P. 61—72. URL: http://www.computer.org/computer/homepage/misc/Boehm/index.htm

11. **Boehm B. W.** Spiral Development: Experience, Principles, and Refinements. Spiral Experience Workshop. 2000. February 9 // Special Report CMU/SEI-2000-SR-008. 2000. July. URL: http://www.sei.cmu.edu/cbs/spiral2000/Boehm

12. **Фатрелл Р. Т., Шафер Д. Ф., Шафер Л. И.** Управление программными проектами: достижение оптимального качества при минимуме затрат. М.: Вильямс, 2003, 2002.

УДК 004.415.2

Б. Г. Кухаренко, канд. физ.-мат. наук, стар. науч. сотр., вед. науч. сотр.,
Институт машиноведения РАН, г. Москва,
e-mail: kukharenko@imash.ru

Принцип открытости-закрытости в программной инженерии и паттерны проектирования. Часть 1

Показано как принцип открытости-закрытости проявляется в объектно-ориентированном программировании на уровне микроархитектуры программных систем. Паттерны проектирования представляют иерархии классов, которые формируют общее решение задачи проектирования программных систем. Рассматриваются методы обнаружения в программных системах модифицированных версий паттернов, которые отличаются от их стандартного представления дополнительными уровнями наследования. Эффективность используемого метода подсчета сходства графов компонентов программных систем и графов паттернов демонстрируется при обнаружении канонических и демонстративных примеров паттернов на языке Java.

Ключевые слова: *объектно-ориентированное программирование, принцип открытости-закрытости, микроархитектура, паттерны проектирования, обнаружение паттернов, алгоритмы графов*

1. Архитектурные стили объектно-ориентированного программирования, микроархитектура и паттерны проектирования

Длительное время попытки повторно использовать программные компоненты при конструировании крупномасштабных программных систем оставались иллюзорной целью [1]. Конечно, одной из причин являлось отсутствие необходимых компонентов, однако имела место также неспособность локализовать уже имеющиеся компоненты. В настоящее время прогресс в этой области достигается через создание библиотек компонентов и улучшенный механизм доступа к их содержимому. При этом возникает проблема иного класса — несоответствие архитектур (*architectural mismatch*) [2]. Перечислим общие факторы, которые приводят к несоответствию архитектур:

- природа компонентов (включая модель управления);
- природа соединений (*connectors*) (протоколы и данные);
- глобальная архитектурная структура;
- процесс создания (*construction process*), т. е. среда и метод разработки (*development environment and build*).

Сдвигом в положительном направлении является указание, как документировать архитектурную структуру программной системы. Например, можно использовать унифицированный язык моделирования UML (*Unified Modeling Language*) для документирования предоставляемых компонентов и требуемых ресурсов, так же как для описания различных форм поведения компонентов [3, 4]. Результатом являются среды разработки программного обеспечения. Например FUJABA, ориентированная на язык Java, позволяет генерировать Java-код на основе UML-диаграмм и формировать UML-диаграммы, описывающие существующий Java-код [5]. Однако этого стандарта недостаточно для преодоления архитектурного несоответствия компонентов, если им пользуется ограниченное число разработчиков программных средств. Реальный процесс создания программных систем описывается спиральной моделью и основан на итеративной разработке с элементами риска. В такой ситуации невозможно полностью избежать архитектурного несоответствия компонентов. Поэтому по-прежнему необходимы средства, позволяющие преодолевать это

Паттерн	Сокращение	Русское название
Abstract Factory	AF	Абстрактная фабрика
Adapter	A	Адаптер
Bridge	BR	Мост
Builder	BU	Строитель
Command	CD	Команда
Composite	CM	Композит
Chain of Responsibilities	CR	Цепочка ответственности
Decorator	D	Декоратор
Facade	FA	Фасад
Factory Method	FM	Фабричный метод
Flyweight	FL	Приспособленец
Interpreter	IN	Интерпретатор
Iterator	IT	Итератор
Mediator	MD	Посредник
Memento	MM	Хранитель
Observer	O	Наблюдатель
Prototype	PT	Прототип
Proxy	PR	Заместитель
Singleton	S	Синглтон
State	ST	Машина состояний
Strategy	SR	Стратегия
Template Method	TM	Шаблонный метод
Visitor	V	Посетитель

несоответствие. Примеры этих средств: оболочки (*wrapper*s), адаптеры (*adapters*), медиаторы (*mediators*) и мосты (bridges), многие из которых каталогизированы в руководствах по проектированию и архитектуре программных средств (*software design and architecture*). Это облегчает повторное использование компонентов программных систем. В объектно-ориентированном программировании (ООП) необходимость преодоления архитектурного несоответствия компонентов, обеспечивающая их повторное использование, сформулирована как принцип открытости-закрытости (*Open Closed Principle* (OCP)): компонент (модуль) должен быть открыт для расширения и закрыт для модификации. Считается, что из всех принципов объектно-ориентированного проектирования этот принцип является наиболее важным [6, 7]. Таким образом, можно менять компоненты без изменения их исходного кода, т. е. только добавляя к ним новый код. Обеспечение этого — сложная задача, но как показано ниже, уже существует подход к ее решению. Он анализирует микроархитектуру программных компонентов в целях обнаружения паттернов проектирования (*design patterns*).

Паттерны проектирования определяются как описания иерархий взаимодействующих классов, которые формируют общее решение задачи проектирования программных систем. Они привлекают большой интерес разработчиков программных средств, поскольку позволяют конструировать хорошо структурированные, приспособленные для модификации и поэтому повторно используемые компоненты программных систем. Полный атлас наиболее распространенных паттернов содержится в работе [8]. Появление ряда паттернов связано с изобретениями в области ООП. Таблица паттернов в форме таблицы (химических) элементов Менделеева (рис. 1, см. вторую сторону обложки) представлена в работе [9]. Слева в таблице находятся креативные паттерны (*creational patterns*), в центре поведенческие (*behavioral patterns*) и справа — структурные (*structural patterns*).

В таблице перечислены паттерны с краткими обозначениями (приведенными на рис. 1) и русским названием.

В работе [8] приводится перечень стандартных характеристик паттернов, по которым их идентифицируют и различают. В настоящей работе рассматриваются универсальные характеристики паттернов проектирования, которые присущи им всем:

- аддитивность иерархической структуры классов;
- блокирование опасных зависимостей между классами;
- гибридизация, т. е. способность реализации нескольких паттернов в единственной иерархической структуре классов.

2. Универсальные характеристики паттернов проектирования

Аддитивность иерархической структуры классов

Паттерны проектирования ООП являются примером аддитивных схем (моделей) создания иерархий классов, которые упрощают клиентский код, использующий эти классы. Аддитивность паттернов подобна аддитивности систем представления знаний в искусственном интеллекте [10]. Примером аддитивных систем представления знаний являются системы правил (продукций). Большинство канонических паттернов проектирования похожим образом реализуются в языках программирования C++, Java и C#. Однако среди них имеются паттерны, имеющие средства непосредственного выражения в языках программирования Java и C#. Характерным примером канонического паттерна является *Template Method* (шаблонный метод). В языке программирования C++ нет элементов (языка), которые непосредственно поддерживают этот паттерн, поэтому в этом языке программирования этот паттерн выглядит достаточно выпукло. Назначение паттерна *Template Method.*

- стандартизация скелета алгоритма в абстрактном базовом классе (AbstractClass);

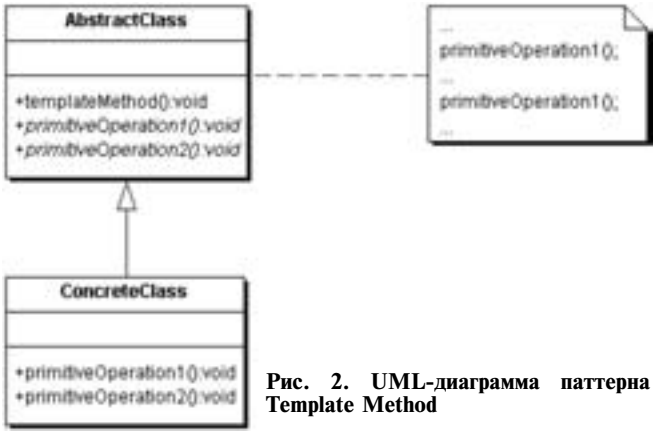

Рис. 2. UML-диаграмма паттерна Template Method

- шаги алгоритма *(template method)*, требующие конкретной интерпретации, в AbstractClass резервируют место (*placeholders*);
- производный класс (ConcreteClass) реализующий эти шаги (*placeholders*).

UML-диаграмма паттерна *Template Method* показана на рис. 2.

В работе [9] приведен пример реализации паттерна *Template Method* на языке C++ и Java. Пример "TemplateMethodDemos.cpp" содержит иерархию классов, реализующую перевод чисел от римской к позиционной (с использованием арабских цифр) системе счисления. Продемонстрируем аддитивность этой реализации паттерна *Template Method* в "TemplateMethodDemos.cpp", реализовав перевод чисел "новой" римской системы счисления в позиционную. Для этого добавим к стандартным римским цифрам символ "B" для обозначения десятков тысяч. Добавление нового символа римской системы счисления приводит к (аддитивному) расширению системы продукций, описывающей переход от римской к позиционной системе счисления (рис. 3). Аналогично расширенной системе продукций (см. рис. 3), реализующая паттерн *Template Method* (аддитивная) иерархия классов обеспечивает переход от "новой" римской (с добавлением символа "B" для обозначения десятков тысяч) к позиционной системе счисления, как это показано на рис. 4 и 5.

После определения производных классов (см. рис. 5) можно объявить соответствующие статические переменные абстрактного базового класса RNInterpreter и определить тело метода (для клиента) /*static*/ int RNInterpreter::interpret (string) (рис. 6).

Клиентский (пользовательский) код и результат его выполнения при вводе "новых" римских цифр показаны на рис. 7. Степень абстрактности иерархии классов определяется как отношение $A = N_a/N_c$, где N_a — число абстрактных классов, а N_c — общее число классов в иерархии [11]. Поэтому для иерархии классов на рис. 4—6 степень абстрактности $A = 1/6$. Добавление в римскую систему счисления еще одного символа для обозначения сотен тысяч осуществляется аналогично приведенному.

Структура кода, реализующего паттерн *Template Method* на языке Java подобна коду на рис. 4—6. Поэтому для краткости на рис. 8 паттерн *Template Method* на языке Java иллюстрируется на искусственном примере из работы [9]. Для иерархии классов на рис. 8 степень абстрактности $A = 2/3$.

Аддитивность представляемых иерархий классов — важное свойство. Но настоящим преимуществом паттернов является их способность ставить непроницаемые барьеры (*firewalls*) для зависимостей между классами [12].

Блокирование нежелательных зависимостей между классами

Зависимости между классами являются причиной снижения качества (деградации) проекта программной системы. Чтобы это предотвратить, зависимости между компонентами должны контролироваться [12]. Этот контроль состоит в создании непроницаемых экранов для зависимостей (*dependency firewalls*).

```
romanNumeral ::= (tenthousands) (thousands) (hundreds) (tens) (ones)
tenthousands, thousands, hundreds, tens, ones ::= nine | four | (five)
(one) (one) (one)
nine ::= "CM" | "XC" | "IX"
four ::= "CD" | "XL" | "IV"
five ::= 'D' | 'L' | 'V'
one  ::= 'B' | 'M' | 'C' | 'X' | 'I'
```

Рис. 3. Расширенная система продукций

```
class TenThousand; class Thousand; class Hundred; class Ten; class One;

class RNInterpreter {
public:
   static int interpret( string input ); // метод для клиента
   void interpret( string& input, int& total ) { // Template Method
      int index = 0;
      if (input.substr(0,2)==nine()){total += 9*multiplier(); index += 2;}
      else if (input.substr(0,2)==four()) {total += 4*multiplier(); index +=
2;}
      else {
         if (input[0] == five()) {total += 5 * multiplier(); index = 1;}
         for (int end = index + 3 ; index < end; index++)
            if (input[index] == one()) total += 1 * multiplier();
            else break;
      }
      // remove all leading chars processed
      input.replace( 0, index, "" );
   }
private:
   virtual char one()        = 0;   virtual string four() = 0;  //
placeholders
   virtual char five()       = 0;   virtual string nine() = 0;  //
placeholders
   virtual int  multiplier() = 0;                                //
placeholders
   static Thousand thousands;       static Hundred hundreds;
   static Ten      tens;            static One     ones;
   static TenThousand tenthousand;
};
```

Рис. 4. Абстрактный базовый класс RNInterpreter

Объектно-ориентированное проектирование (*Object Oriented Design*) обеспечивает принципы и методы создания таких непроницаемых для зависимостей экранов. Пример наиболее распространенных паттернов дает паттерн *Visitor* (Посетитель) (рис. 9) [13, 14]. Продемонстрируем устранение зависимостей между классами иерархии на примере этого паттерна *Visitor*.

```
class TenThousand: public RNInterpreter {
   char  one()        { return 'B'; }   string four() { return ""; }
   char  five()       { return '\0'; }   string nine() { return ""; }
   int   multiplier() { return 10000; }
};
class Thousand : public RNInterpreter {
   char  one()        { return 'M'; }    string four() { return "MA"; }
   char  five()       { return 'A'; }   string nine() { return "MB"; }
   int   multiplier() { return 1000; }
};
class Hundred : public RNInterpreter {
   char  one()        { return 'C'; }   string four() { return "CD"; }
   char  five()       { return 'D'; }   string nine() { return "CM"; }
   int   multiplier() { return 100; }
};
class Ten : public RNInterpreter {
   char  one()        { return 'X'; }   string four() { return "XL"; }
   char  five()       { return 'L'; }   string nine() { return "XC"; }
   int   multiplier() { return 10; }
};
class One : public RNInterpreter {
   char  one()        { return 'I'; }   string four() { return "IV"; }
   char  five()       { return 'V'; }   string nine() { return "IX"; }
   int   multiplier() { return 1; }
};
```

Рис. 5. Классы, производные от RNInterpreter

```
TenThousand RNInterpreter::tenthousand; Thousand RNInterpreter::thousands;
Hundred  RNInterpreter::hundreds; Ten RNInterpreter::tens;
One RNInterpreter::ones;

/*static*/ int RNInterpreter::interpret( string input ) {
// метод для клиента
   int total = 0;
   tenthousand.interpret (input,total);
   thousands.interpret(input,total );
   hundreds.interpret( input, total ); tens.interpret( input, total );
   ones.interpret( input, total );
   // if any input remains, the input was invalid, return 0
   if (input != "") return 0;
   return total;
}
```

Рис. 6. Объявление статических переменных

```
int main() {
   string  input;
   cout << "Enter New Roman Numeral: ";
   while (cin >> input)
   {
   cout << "interpretation is" << RNInterpreter::interpret(input)<< endl;
   cout << "Enter New Roman Numeral: ";
   }
   return 0;
}

// Результат выполнения при введении трех «новых» римских цифр
//Enter New Roman Numeral: BMCVI
//   interpretation is 11106
//Enter New Roman Numeral: BBMCVI
//   interpretation is 21106
//Enter New Roman Numeral: BBBMCVI
//   interpretation is 31106
```

Рис. 7. Клиентский код и результат его выполнения

Паттерн *Visitor* (см. рис. 9) позволяет абстрагировать функциональность, которая применяется к иерархии классов, производных от абстрактного класса *Elemen*t. Новая функциональность добавляется к иерархии наследования *Element* посредством класса, наследующего абстрактный класс *Visitor*. Объектно-ориентированные операции обычно демонстрируют "единственное назначение" (*single dispatch*): они зависят от класса вызывающего объекта и вызываемого метода. Паттерн *Visitor* реализует "двойное назначение" (*double dispatch*). При "двойном назначении" выполняемая операция зависит от класса двух объектов (конкретного класса, производного от *Visitor*, и конкретного класса, производного от *Element*). Для каждого конкретного класса *A* и *B*, производного от *Element*, в абстрактном классе *Visitor* объявляется соответствующий чисто виртуальный метод visit(A&) = 0 и visit(B&) = 0 (каждый чисто виртуальный метод visit(..) = 0 принимает единственный аргумент – ссылку конкретного класса, производного от *Element*). В классе, производном от *Visitor*, реализация перегруженного метода visit(..) содержит запрос относительно типа конкретного класса, производного от *Element*, и приведение типа. В абстрактном классе *Element* объявляется единственный чисто виртуальный метод *accept*(*Visitor*&) = 0 (метод accept(..) получает единственный параметр — ссылку абстрактного класса *Visitor*. Каждый конкретный класс, производный от *Element*, реализует метод accept(..). Если объект класса, производного от *Element*, вызывает метод accept(..) со ссылкой конкретного класса, производного от *Visitor*, то при этом вызывается метод visit(..) этого конкретного класса, которому вызывающий объект класса, производного от *Elemen*t, передается посредством указателя *this*.

Программная реализация паттерна *Visitor* в статических языках C++, Java и т. д. приводит к циклической зависимости (*dependency cycle*) в исходном коде программы. На рис. 9 эта циклическая зависимость показана в UML-нотациях:

- объявление класса *Element* зависит от класса *Visitor*;
- класс *Visitor* имеет отдельную функцию-член для каждого класса, производного от *Element*. Таким образом, объявление класса *Visitor* зависит от объявления каждого конкретного класса, производного от *Element*;
- естественно, что объявление каждого класса, производного от *Element*, зависит от этого класса *Element*.

В результате объявление класса *Element* транзитивно зависит от объявления всех его производных классов. Эту проблему решает паттерн *Acyclic Visitor* (рис. 10) [15].

Опишем классы — участники иерархии на рис. 10:

```
abstract class Generalization {
   // 1. Standardize the skeleton of an algorithm in a "template" method
   public void findSolution() {
      stepOne();
      stepTwo();
      stepThr();
      stepFor();
   }
   // 2. Common implementations of individual steps are defined in base class
   protected void stepOne() { System.out.println( "Generalization.stepOne" );
}
   // 3. Steps requiring peculiar impls are "placeholders" in the base class
   abstract protected void stepTwo();
   abstract protected void stepThr();
   protected void stepFor() { System.out.println( "Generalization.stepFor" );
}
}

abstract class Specialization extends Generalization {
   // 4. Derived classes can override placeholder methods
   // 1. Standardize the skeleton of an algorithm in a "template" method
   protected void stepThr() {
      step3_1();
      step3_2();
      step3_3();
   }
   // 2. Common implementations of individual steps are defined in base class
   protected void step3_1() { System.out.println( "Specialization.step3_1" );
}
   // 3. Steps requiring peculiar impls are "placeholders" in the base class
   abstract protected void step3_2();
   protected void step3_3() { System.out.println( "Specialization.step3_3" );
}
}

class Realization extends Specialization {
   // 4. Derived classes can override placeholder methods
   protected void stepTwo() { System.out.println( "Realization   .stepTwo" );
}
   protected void step3_2() { System.out.println( "Realization   .step3_2" );
}
   // 5. Derived classes can override implemented methods
   // 6. Derived classes can override and "call back to" base class methods
   protected void stepFor() {
      System.out.println( "Realization   .stepFor" );
      super.stepFor();
      }
}

class TemplateMethodDemo {
   public static void main( String[] args ) {
      Generalization algorithm = new Realization();
      algorithm.findSolution();
      }
}
```

Рис. 8. Пример реализации Template Method на языке Java

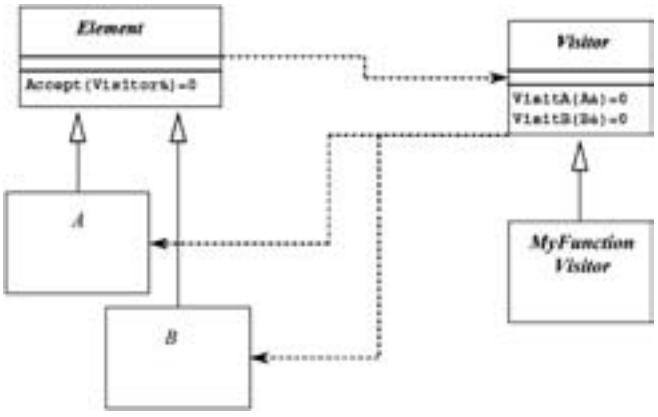

Рис. 9. UML-диаграмма паттерна *Visitor*

- *Element* — абстрактный базовый класс иерархии. Методы классов типа *Visitor* действуют на объектах типа *Element*.
- *E*1, *E*2, ... — конкретные классы производные от *Element*.
- *Visitor* — вырожденный базовый класс (не имеет членов кроме виртуального деструктора). Он играет роль заместителя (*placeholder*) в структуре типов. Это тип аргумента, принимаемого методом accept(..) класса *Element*. Классы, производные от *Element*, принимают этот параметр в операторе приведения dynamic_cast < > (..).
- *E*1*Visitor*, *E*2*Visito*r, ... — абстрактные классы, соответствующие каждому конкретному классу, производному от *Element* (взаимно однозначное соответствие), (каждый такой абстрактный класс имеет чисто виртуальный метод visit(..), который принимает как параметр ссылку на соответствующий ему конкретный класс, производный от *Element*).
- V*isitForF* — конкретный класс, производный от *Visitor*. Поэтому он может быть передан как параметр методу accept(..) класса *Element*. Он также является производным от абстрактных классов *E*1*Visitor*, *E*2*Visitor*, ... Классу *VisitForF* нет необходимости наследовать все абстрактные классы *E*1*Visitor*, *E*2*Visitor*, ..., а только те, для которых он реализует метод visit(..).

Процесс взаимодействия классов на рис. 10 начинается, когда пользователь создает объект класса *VisitForF*.

1. Пользователь вызывает метод accept(..) на объекте типа *Element* и передает ему ссылку на объект типа *Visito*r.

2. Метод accept(..) конкретного класса, производного от *Element*, использует dynamic_cast < .. > (..) чтобы привести объект типа *Visitor* к соответствующему абстрактному классу *E*1*Visitor*.

3. Если приведение посредством dynamic_cast < .. > (..) успешное, то ссылка на объект конкретного класса, производного от *Element*, передается методу visit(..) абстрактного класса *E*1*Visitor*.

4. Фактический объект класса *VisitForF* выполняет реализацию этого метода visit(..).

Гибридизация — способность реализации нескольких паттернов в единственной иерархической структуре классов

UML-диаграмма паттерна *Visitor* показана на рис. 9. UML-диаграмма паттерна *Composite* показана на рис. 11.

Рассмотрим реализацию гибридного паттерна *Visitor* + *Composite*. Структура кода, реализующего паттерн *Visitor* + *Composite* на языках C++ и Java подобна, и на рис. 12 иллюстрируется на языке Java [9].

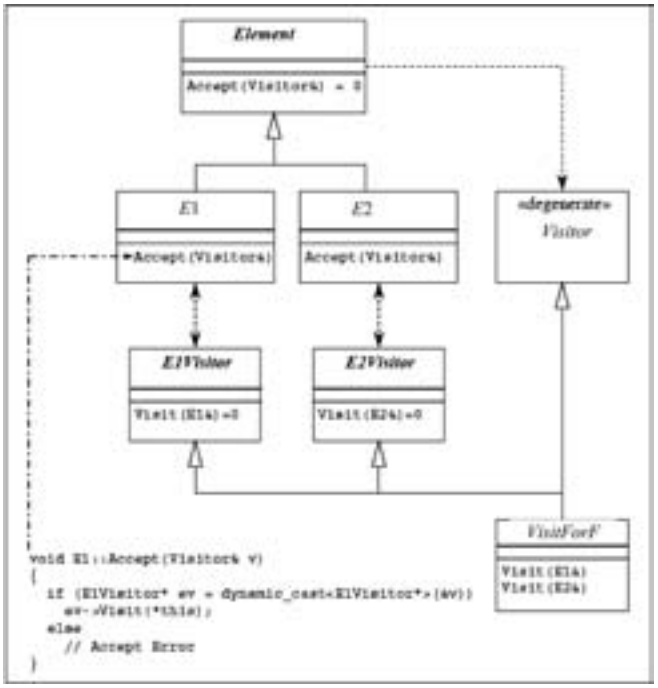

Рис. 10. UML-диаграмма паттерна Acyclic Visitor

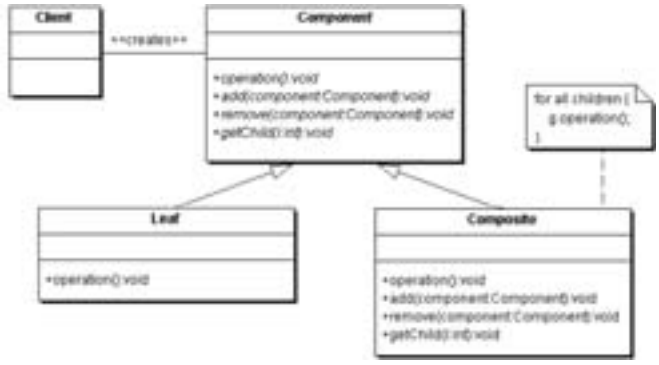

Рис. 11. UML-диаграмма паттерна Composite

Для иерархии классов на рис. 12 степень абстрактности $A = 2/5$.

```
import java.util.*;

interface Component {
  void traverse();
  void accept( Visitor v );
}

class Leaf implements Component {
  private int number;
  public Leaf( int num )          { number = num; }
  public void traverse()          { System.out.print( number + " " ); }
  public void accept( Visitor v ) { v.visit( this ); }
  public int  getNumber()         { return number; }
}

class Composite implements Component {
  private static char next = 'a';
  private List children = new ArrayList();
  private char letter = next++;

  public void add( Component c ) { children.add( c ); }
  public void traverse() {
    System.out.print( letter + " " );
    for (int i=0; i < children.size(); i++)
      ((Component)children.get(i)).traverse();
  }
  public void accept( Visitor v ) {
    v.visit( this );
    for (int i=0; i < children.size(); i++)
      ((Component)children.get(i)).accept( v );
  }
  public char getLetter() { return letter; }
}

interface Visitor {
  void visit( Leaf l );
  void visit( Composite c );
}

class CollectVisitor implements Visitor {
  private StringBuffer letters = new StringBuffer();
  private StringBuffer numbers = new StringBuffer();

  public void   visit( Composite c ) { letters.append( c.getLetter() ); }
  public void   visit( Leaf l )      { numbers.append( l.getNumber() ); }
  public String getLetters()         { return letters.toString(); }
  public String getNumbers()         { return numbers.toString(); }
}

public class VisitorComposite3 {
  public static void main( String[] args ) {
    Composite[] containers = new Composite[3];

    for (int i=0; i < containers.length; i++) {
      containers[i] = new Composite();
      for (int j=1; j < 4; j++)
        containers[i].add( new Leaf( i + containers.length + j ) );
    }

    for (int i=1; i < containers.length; i++)
      containers[0].add( containers[i] );

    containers[0].traverse();
    System.out.println();

    CollectVisitor anOperation = new CollectVisitor();
    containers[0].accept( anOperation );
    System.out.print( "letters are - " + anOperation.getLetters() );
    System.out.println( ", numbers are - " + anOperation.getNumbers() );
      }
}
```

Рис. 12. Пример реализации гибридного паттерна Visitor + Composite на языке Java

Список литературы

1. **Garlan D., Allen R., Ockerbloom J.** Architectural mismatch or, Why it's hard to build systems out of existing parts // IEEE Software. 1995. V. 12. Issue 6. P. 17—26.

2. **Garlan D., Allen R., Ockerbloom J.** Architectural mismatch: Why reuse is still so hard // IEEE Software. 2009. July. V. 26. Issue 4. P. 66—69.

3. **Fowler M., Scott K.** UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, 1999.

4. **Nickel U. A., Niere J., Zundorf A.** Tool demonstration: The FUJABA environment // Proc. of the 22nd International Conf. on Software Engineering (ICSE). Limerick, Ireland. ACM Press. 2000. P. 742—745.

5. **Martin R. C.** Designing Object Oriented Applications using UML. 2nd. ed. Prentice Hall, 1999.

6. **Meyer B.** Object-Oriented Software Construction. 2nd edition. Prentice Hall, 1997.

7. **Martin R. C.** The open-closed principle // More C++ gems. New York, NY: Cambridge University Press. 2000. P. 97—112.

8. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.

9. **Huston V.** Design Patterns in a Nutshell. Sebastopol, CA: O'Reilly & Associates. 2007.

10. **Рассел С., Норвиг П.** Искусственный интеллект: современный подход. 2-е изд. М.: Вильямс, 2006.

11. **Martin R. C.** OO Design Quality Metrics. An Analysis of Dependencies. Gurnee, Illinois: Object Mentor Inc., 1994.

12. **Martin R. C.** Design Principles and Design Patterns. Gurnee, Illinois: Object Mentor Inc., 2000.

13. **Martin R. C.** The Visitor Family of Design Patters. Gurnee, Illinois: Object Mentor Inc., 2002.

14. **Meyer B., Arnout K.** Componentization: the Visitor example // Computer. 2006. July. V. 39. N 1. P. 23—30.

15. **Martin R. C.** Acyclic Visitor / Martin R., Riehle D., Buschmann F., editors. Pattern Languages of Program Design. V. 3: Software Patterns. Addison-Wesley, 1998. P. 93—104.

УДК 004.65:004.413:005.8

С. С. Асадуллаев, канд. физ.-мат. наук, исполнительный архитектор SWG IBM EE/A,
e-mail: Sabir_Asadullaev@ru.ibm.com

Хранилища данных: тройная стратегия на практике

На практическом примере системы сбора и анализа первичных данных показано, как тройная стратегия и рекомендованная архитектура корпоративного хранилища данных позволяют предоставить более высокое качество информационного — аналитического обслуживания при сокращении стоимости и времени разработки корпоративного хранилища данных.

Ключевые слова: *корпоративное хранилище данных, стратегия разработки, архитектура*

Введение

Многие успешные компании обнаруживают, что раздельное управление по линиям бизнеса не дает целостной картины положения компании на рынке. Для принятия точных своевременных решений эксперты, аналитики и руководство компании нуждаются в единой, непротиворечивой информации, которую должны предоставлять корпоративные хранилища данных (КХД).

На практике стоимость и сроки создания КХД оказываются выше ожидаемых. При этом аналитические отчеты зачастую по-прежнему содержат противоречивую информацию. В работе показано, что следование рекомендованным архитектурным решениям, использование апробированной стратегии создания КХД и правильный выбор программных средств способны снизить стоимость разработки КХД и повысить качество его услуг.

На основании тройной стратегии, рекомендованной архитектуры, сформулированных принципов и лучших практик построения КХД предложен план управления проектом разработки программного обеспечения корпоративного хранилища данных.

Компания IBM предлагает полный набор средств для интеграции данных, метаданных и нормативно-справочной информации (НСИ) на всех этапах жизненного цикла проекта создания КХД. Целью данной статьи является анализ упрощенного решения на основе программного обеспечения IBM Forms, IBM InfoSphere Warehouse и IBM Cognos BI. Решение должно быть масштабируемым и функционально расширяемым, легко интегрируемым в корпоративную ИТ-инфраструктуру и способным стать основой для корпоративного хранилища данных.

Архитектура системы сбора и анализа первичных данных

В работах [1, 2] предложено типовое решение для системы сбора, хранения и анализа первичных данных ручного ввода. Напомним основные требования к системе:

- необходим сбор первичных статистических, или отчетных показателей с удаленных рабочих мест;
- необходима проверка данных на рабочем месте до отправки в центр;
- собранные показатели должны выверяться и храниться в течение времени, определенного нормативными документами;
- для выявления закономерностей и принятия управленческих решений необходимо выполнение аналитических вычислений на основе собранной статистики;
- для оценки состояния дел в регионах необходимо формирование управленческой отчетности.

Решение должно быть расширяемым и масштабируемым. Например, необходимо предусмотреть последующую интеграцию системы "Сбор и анализ первичных данных" с системой документооборота.

На рис. 1 представлена централизованная архитектура системы сбора, хранения и анализа данных, которая предполагает, что ввод данных может осуществляться удаленно, а все серверы сбора данных IBM Forms [3], хранения данных InfoSphere Warehouse [4] и анализа и интерпретации данных Cognos [5, 6] установлены в едином центре обработки данных. Аналитики могут работать как локально, так и дистанционно, благодаря тому, что сервер Cognos предоставляет веб-интерфейс для подготовки и выполнения аналитических расчетов.

Предложенная архитектура была основана на максимально простой конфигурации программного обес-

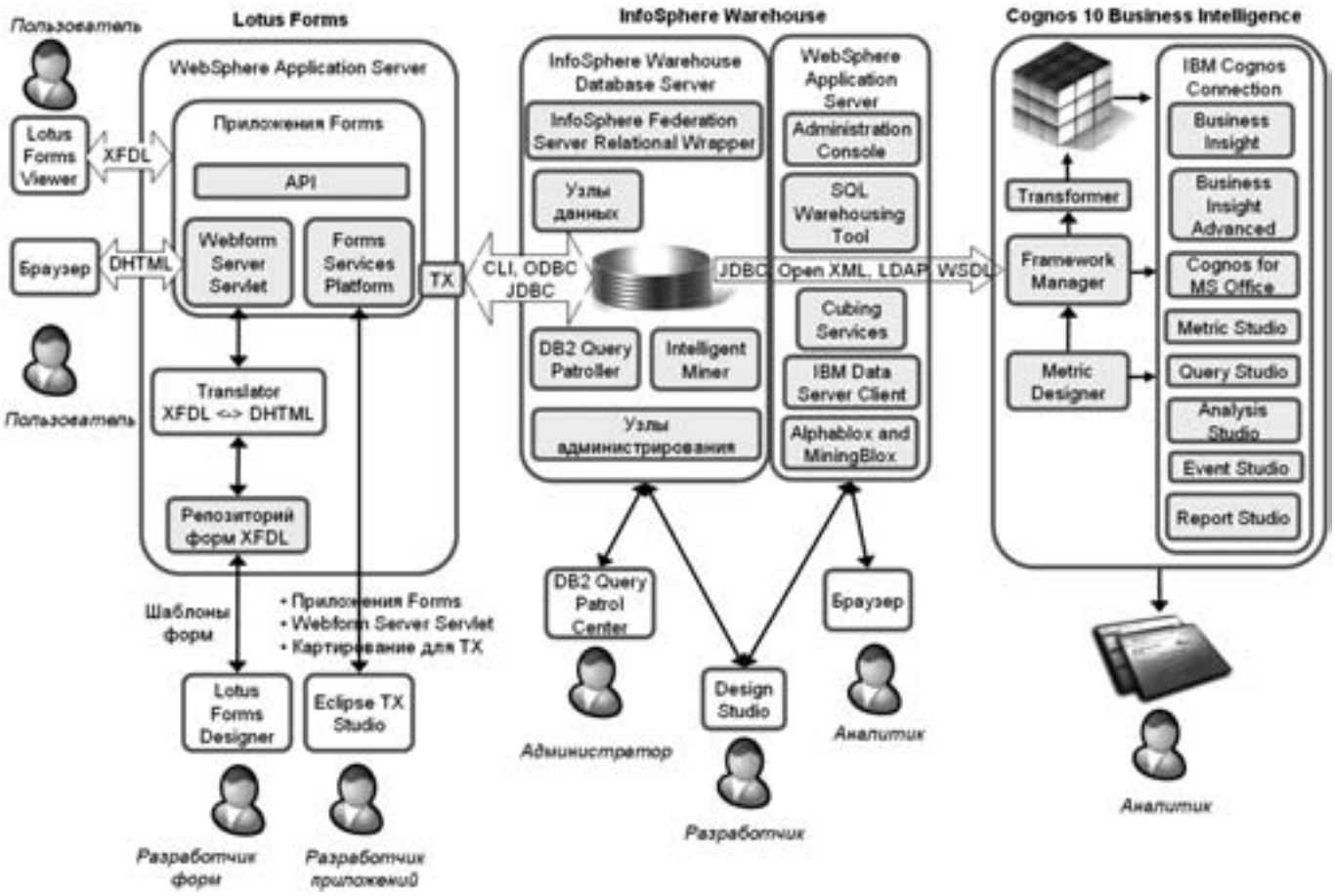

Рис. 1. Архитектура решения на основе IBM Forms, InfoSphere Warehouse и Cognos BI

печения для типовой задачи без учета специфических требований различных проектов. Несмотря на очевидные ограничения, предложенный подход может быть использован в качестве основы самых разных решений для различных отраслей и предприятий.

Разделение подсистем на ввод данных, их сбор, хранение и анализ позволяет строить различные архитектуры в зависимости от потребностей задачи и требований к корпоративной инфраструктуре.

Другим преимуществом предлагаемого модульного построения системы является возможность расширения ее функциональности. Поскольку все модули взаимодействуют по стандартным общепринятым протоколам, возможна интеграция с системами документооборота, ведения метаданных и НСИ, корпоративного планирования ресурсов, а также с различными аналитическими и статистическими пакетами.

Система сбора и анализа первичных данных может быть легко включена в существующую корпоративную ИТ-инфраструктуру. В других условиях она может стать первым этапом реализации корпоративной системы сбора, хранения и анализа данных.

Как можно заметить, рассматриваемая архитектура системы сбора, хранения и анализа данных (см. рис. 1) не содержит средств ведения метаданных или НСИ. На первый взгляд, это противоречит предложенной тройной стратегии создания хранилищ данных [7], которая предполагает интеграцию данных, метаданных и НСИ. С другой стороны, не ясно, как решение соотносится с рекомендованной архитектурой хранилищ данных [8], и чем отличается предложенный подход от многочисленных проектов, основная цель которых — демонстрация быстрого маленького успеха.

Место проектов ведения метаданных и НСИ

Задача ввода первичных данных, их сбор, хранение и анализ обладают несколькими особенностями. Прежде всего, первичные данные вводятся вручную в поля утвержденных экранных форм. Поля форм согласованы между собой как внутри формы, так и между формами. Это означает, что разные сущности имеют разные названия и разные поля. Поэтому заказчик уже на ранних этапах планировал хранение не форм, или отчетов, а показателей, из которых в дальнейшем собираются формы и отчеты. Непротиворечивый набор показателей — это отличный первый шаг к управ-

лению метаданными, даже если это требование не было сформулировано в явном виде.

В данных условиях первый этап ведения метаданных не требует использования специфического программного обеспечения и может быть выполнен с помощью карандаша, ластика и ученической тетради. Основная сложность этого этапа — достижение согласия всех экспертов о терминах, сущностях, об их названиях и способах расчета. Иногда пользователям приходится отказываться от привычных, но неоднозначных наименований, и достижение согласия может потребовать значительных усилий и времени. К счастью, эта работа была выполнена заказчиком до обращения к внешним исполнителям проекта.

Наименования полей форм ручного ввода, методы их выверки и расчета являются необходимыми бизнес-метаданными. На основе собранных метаданных строится архитектура решения, в том числе, модель хранилища данных, создаются и именуются таблицы в КХД и столбцы в таблицах. Тем самым положено начало формирования технических метаданных.

В процессе сопровождения построенной системы неизбежна необходимость внесения изменений. В этот момент разработчики нуждаются в ведении глоссария терминов. Если он не был создан ранее, самое время задуматься о его реализации, поскольку процесс сопровождения системы вынуждает начать централизованное ведение метаданных в явном виде. Данный сценарий подразумевает минимальные накладные расходы на внедрение централизованной системы управления метаданными, поскольку предварительно было создано ядро будущей системы. Это ядро, пусть маленькое и не имеющее достаточной функциональности, обладает важнейшим активом — непротиворечивыми метаданными.

Централизованное ведение нормативно-справочной информации должно быть начато одновременно с ведением метаданных. Причина проста — НСИ и метаданные тесно связаны [7], и ведение одного проекта без другого, как правило, не приводит к успеху.

Основой для системы ведения НСИ может стать совокупность кодификаторов, справочников, классификаторов, идентификаторов, нормативов и словарей, ведущихся при хранилище данных. Гарантией качества для НСИ в данном случае является проработанность метаданных, что исключает конфликты кодировки данных при условии квалифицированной разработки ХД.

Таким образом, систематизация бизнес-метаданных на основе полей форм, выполненная на предпроектном этапе, обеспечила возможность беспроблемного создания систем ведения метаданных и НСИ. Это позволило сократить бюджет проекта внедрения системы сбора, хранения и анализа первичных данных. При этом проектная команда отдавала себе отчет, что проекты ведения метаданных и НСИ ведутся неявным образом, требуя на данном этапе только стратегического видения от проектировщиков и аккуратности от разработчиков.

Рекомендованная архитектура КХД

Рекомендованная архитектура КХД, предложенная в работе [8], построена в соответствии со следующими основными принципами.

- КХД является единым источником непротиворечивых данных и должно обеспечивать пользователей согласованными качественными данными из различных информационных систем.
- Данные должны быть доступны сотрудникам в объеме, необходимом и достаточном для выполнения своих функциональных обязанностей.
- Сотрудники должны иметь единое понимание данных, т. е. должно быть установлено единое смысловое пространство.
- Необходимо устранить конфликты в кодировках данных в системах-источниках.
- Аналитические вычисления должны быть отделены от оперативной обработки данных.
- Необходимо обеспечить и поддерживать многоуровневую организацию данных.
- Необходимо следовать эволюционному подходу, позволяющему обеспечить непрерывность бизнеса и сохранить инвестиции в ИТ.
- Информационное наполнение будущего хранилища данных, этапность построения КХД и ввода функциональных моделей в эксплуатацию определяются, в первую очередь, требованиями бизнес-пользователей.
- Необходимо обеспечить защиту данных и их надежное хранение. Меры по защите информации должны быть адекватны ценности данных.

Архитектура, построенная в соответствии с этими принципами, следует принципам модульного конструирования "непотопляемых отсеков". Разделяя архитектуру на модули, мы одновременно концентрируем в них определенную функциональность (рис. 2, см. вторую сторону обложки).

Средства очистки, преобразования и загрузки данных ETL (Extract, Transform, Load) обеспечивают полный, надежный, точный сбор информации из источников данных, благодаря сосредоточенной в ETL логике сбора, обработки и преобразования данных и взаимодействию с системами ведения метаданных и НСИ.

Система ведения метаданных является основным источником информации о данных в КХД. Система ведения метаданных поддерживает актуальность бизнес-метаданных, технических, операционных и проектных метаданных.

Система ведения НСИ позволяет устранить конфликты в кодировках исходных данных в системах-источниках.

Центральное хранилище данных (ЦХД) несет только нагрузку по надежному защищенному хранению данных. Структура данных в ЦХД оптимизирована исключительно в целях обеспечения эффективного хранения данных.

Средства выборки, реструктуризации и доставки данных (SRD) в такой архитектуре являются единст-

венным пользователем ЦХД, беря на себя всю работу по заполнению витрин данных и снижая нагрузку на ЦХД по обслуживанию запросов пользователей.

Витрины данных содержат данные в структурах и форматах, оптимальных для решения задач пользователей данной витрины.

Таким образом достигается удобная работа пользователей с необходимым объемом данных даже при отсутствии связи с ЦХД, возможность быстрого восстановления содержимого витрин из ЦХД при сбое витрины.

Достоинство этой архитектуры заключается в возможности раздельного проектирования, создания, эксплуатации и доработки отдельных компонентов без радикальной перестройки всей системы. Это означает, что начало работ по созданию КХД не требует сверхусилий или сверхинвестиций. Достаточно начать с ограниченного по своим возможностям программно-технического комплекса, и, следуя предложенным принципам, создать работающий и действительно полезный для пользователей прототип. Далее необходимо выявить узкие места и развивать соответствующие компоненты.

Связь архитектуры решения с рекомендованной архитектурой

Архитектура решения для системы сбора, хранения и анализа первичных данных, представленная на рис. 3 (см. вторую сторону обложки) переведена в термины рекомендованной архитектуры КХД и совмещена с ней.

Сбор данных осуществляется с помощью набора продуктов IBM Forms, который использует электронные формы для ручного ввода данных и позволяет передавать собранные данные в другие системы. Сервер приложений IBM Forms может быть в дальнейшем интегрирован с репозиториями структурированных и неструктурированных данных.

Единственным источником данных в проекте являются формы, заполняемые по строгим правилам, поэтому на текущем этапе нет необходимости в средствах ETL выборки, преобразования и загрузки данных (например, DataStage). Однако по мере развития проекта ожидается подключение других источников. Возможность использования средств ETL обеспечивает функциональную расширяемость системы без необходимости ее радикальной переработки.

Хранение данных реализовано с помощью IBM InfoSphere Warehouse. Анализ данных может быть выполнен средствами IBM InfoSphere Warehouse и IBM Cognos Business Intelligence (BI).

IBM InfoSphere Warehouse предоставляет следующие инструменты анализа данных: реализация аналитической обработки OLAP на основе инструментов Cubing Services и Alphablox, интеллектуальный анализ данных с помощью программных средств Miningblox и Alphablox, интеллектуальный анализ данных с привлечением программного обеспечения Intelligent Miner.

Интегрированный программный комплекс для управления корпоративной эффективностью IBM Cognos 10 BI помогает аналитикам интерпретировать данные, возникающие в процессе деятельности компании. Используя IBM Cognos 10 BI, любой сотрудник организации может построить графики, сравнить план и факт, создать различные типы отчетов и встроить их в удобный портал; создать персонализированные панели управления (*dashboard*), выполнить анализ данных и провести мониторинг событий и метрик в целях выработки эффективных бизнес-решений.

Аналитический инструментарий может быть дополнен высокопроизводительным средством корпоративного планирования IBM Cognos TM1, которое предоставляет полную среду планирования для подготовки персонализированных бюджетов и прогнозов.

Метаданные, полученные как побочный продукт согласования форм, и НСИ, как результат приведения данных к нормальной форме в реляционной базе данных КХД, являются прототипами будущих корпоративных систем управления метаданными и НСИ (рис. 4, см. третью сторону обложки).

Первые публикации о необходимости создания систем словарей-справочников появились в середине 80-х гг. прошлого века [9]. В 1995 г. была опубликована статья [10], в которой было указано, что для успешной интеграции данных необходимо организовать и поддерживать поток метаданных. Современная практика показывает, что это требование нуждается в уточнении, так как метаданные порождаются на всех этапах создания и эксплуатации информационных систем. Связь между данными, НСИ и метаданными подробно рассмотрена в работе [8], в которой было показано, что НСИ содержит бизнес-метаданные и технические метаданные.

Загрузка данных в КХД не может быть корректно выполнена без метаданных и НСИ, которые явно или неявно, но интенсивно используются на этом этапе. Очищенные и согласованные данные сохраняются, но метаданные и НСИ, как правило, игнорируются.

Создание репозиториев метаданных и НСИ значительно сокращает издержки на реализацию КХД, позволяет перейти от хранения несогласованных форм к хранению непротиворечивых показателей и повышает качество информационного обслуживания бизнес-пользователей [11].

Сравнение предлагаемого и существующих подходов

Для того чтобы ответить на вопрос, в чем отличие предлагаемого подхода от общепринятой практики, рассмотрим типичный пример проекта создания финансовой аналитической системы в банке [12].

Проектная команда исходила из того, что создание корпоративных мастер-данных является долгой, сложной и рискованной работой. Поэтому проект был ограничен решением локальной задачи реинжиниринга процессов планирования и прогнозирова-

ния, которые должны подготовить почву для создания системы банковской отчетности на основе интеграции всех основных банковских систем в целях использования более детальных данных, согласованных с главной бухгалтерской книгой.

Разработка корпоративного хранилища данных в глазах проектной команды была равносильна "большому взрыву", создавшему Вселенную. Проектная команда, избегая решений уровня предприятия, внедрила системы ведения метаданными и НСИ для одной специфической области деятельности. Поэтому репозиторий финансовых данных является узкотематической витриной для финансовой отчетности (рис. 5, см. третью сторону обложки).

В противоположность этому, корпоративное хранилище данных предоставляет согласованные корпоративные данные для широкого ряда аналитических приложений. Как показывает практика, единую версию данных способно обеспечить только КХД, взаимосвязанное с корпоративными системами ведения НСИ и метаданных.

Как видим, основная цель этого проекта — демонстрация быстрого маленького успеха. Многие из нас были поставлены в такую же ситуацию, когда было необходимо срочно продемонстрировать пусть маленькую, но работающую систему. Опытный разработчик понимает, что ему придется последовать совету Брукса [13] и выбросить эту первую версию. Причина заключается в том, что цена переработки созданных приложений и их включения в корпоративную инфраструктуру будет чрезмерно высока из-за отсутствия согласованных метаданных и НСИ.

Итоговая архитектура реализации существующих подходов

Коротко суммируем итоги выполненного анализа.

- Существующие подходы фактически внедряют разрозненные прикладные витрины данных. Данные в витринах могут иметь ценность внутри подразделений, но не для компании в целом, из-за невозможности их сведения вследствие разного смысла данных и их кодировки.
- Распространено убеждение, что создание КХД подобно смертельному трюку с непредсказуемыми последствиями, поэтому зачастую выбирается создание локальных витрин без учета развития КХД в целом.
- Требование мгновенных результатов вызывает стремление разработать и внедрить какое-нибудь ограниченное решение без связи с остальными задачами.

Следуя этим принципам, компании сначала внедряют разрозненные независимые витрины. Информация в витринах не согласована с данными из других витрин, поэтому на стол руководству ложатся противоречивые отчеты. Показатели в этих отчетах с одинаковыми названиями могут скрывать разные сущности, и наоборот, одинаковые сущности могут иметь разные наименования, могут вычисляться по разным алгоритмам, на основании разного набора данных, за разные сроки.

В результате пользователи независимых прикладных витрин говорят на разных языках бизнеса, и каждая витрина содержит собственные метаданные.

Другая проблема заключается в различии НСИ, используемой в независимых витринах данных. Разница в кодировке данных, в используемых кодификаторах, справочниках, классификаторах, идентификаторах, нормативах и словарях делает невозможным объединение этих данных без серьезного анализа, проектирования и разработки средств ведения НСИ.

Так в компании создаются несколько хранилищ несогласованных данных, что в корне противоречит самой идее создания КХД как единого и единственного источника очищенных, согласованных и непротиворечивых исторических данных. Отсутствие корпоративного ведения метаданных и НСИ (затенены на рис. 6, см. третью сторону обложки) еще более затрудняют возможность согласования данных между собой.

Понятно, что ни руководство, ни пользователи подобного хранилища не склонны доверять информации, содержащейся в нем. Поэтому на следующем этапе встает необходимость радикальной переработки, а по сути, создания заново хранилища, ориентированного на хранение не отчетов, а согласованных показателей, из которых будут собираться отчеты.

Таким образом, стремление к достижению сиюминутных результатов и к демонстрации быстрых успехов приводит к отказу от единого, сквозного управления метаданными и НСИ. Итогом такого подхода является наличие семантических островов, где сотрудники говорят на разных бизнес-языках. Полная переработка корпоративной архитектуры интеграции данных становится настоятельной необходимостью и приводит к повторному расходу средств и времени на создание полноценного КХД (см. рис. 6).

Тройная стратегия и планирование создания КХД

Предлагаемый подход основывается на тройной стратегии, рекомендованной архитектуре, на сформулированных принципах и лучших практиках построения КХД.

Как правило, от разработчиков требуется быстро продемонстрировать маленький успех по интеграции данных. В некоторых компаниях, напротив, необходимо разработать и внедрить корпоративную стратегию создания КХД. Как бы ни была поставлена задача, в обоих случаях нужно иметь перед глазами глобальную цель и достигать ее посредством коротких этапов. Роль "компаса", указывающего на глобальные цели, отводится скоординированной интеграции данных, метаданных и НСИ (рис. 7):

- интеграция НСИ, нацеленная на устранение избыточности и несовместимости данных;

• интеграция метаданных, направленная на обеспечение единого понимания данных и метаданных;

• интеграция данных, целью которой является предоставление конечным пользователям единой версии правды на основе согласованных метаданных и НСИ.

Как известно, большое путешествие начинается и заканчивается маленьким шагом. Создание централизованной среды управления данными, метаданными и НСИ является приоритетной задачей. Но бизнес-пользователи не видят мгновенной выгоды для себя от этой среды, и руководство предпочитает избегать длительных проектов без осязаемых результатов для основного бизнеса компании.

Поэтому на первой фазе следует выбрать 2—3 пилотных проекта. Основными критериями выбора этих проектов являются поддержка руководства и готовность пользователей и экспертов участвовать в постановке задачи. Проекты должны обеспечить минимально приемлемую функциональность будущего КХД.

В качестве условного примера на рис. 7 выбраны следующие пилотные проекты для реализации на первой фазе:

• интеллектуальный анализ данных (*data mining*) на основе Intelligent Miner (IM);

• многомерный анализ (OLAP) с помощью Cubing Services и Alphablox;

• анализ неструктурированного текста с использованием программных инструментов Unstructured Text Analysis Tools (UTAT).

Все перечисленные инструменты, используемые на первой фазе пилотных проектов, входят в состав IBM InfoSphere Warehouse.

Очень важно, чтобы в результате реализации этих коротких проектов пользователи ощутили реальные преимущества КХД. Совместно с пользователями необходимо выполнить анализ результатов внедрения проектов и определить, при необходимости, меры по изменению среды КХД и внести коррективы в мероприятия по интеграции данных, метаданных и НСИ.

На следующем этапе необходимо отобрать 3—4 новых проекта, которые бы продвинули компанию к созданию базовой функциональности перспективного КХД. Желательно, чтобы в отборе участвовали все заинтересованные стороны — руководство компании, пользователи, эксперты, проектная команда и специалисты по эксплуатации и поддержке КХД. Централизованная среда управления данными, метаданными и НСИ должна быть достаточна развита, чтобы соответствовать требованиям базовой функциональности КХД.

Допустим, что на второй фазе выбраны для реализации следующие проекты и инструменты:

• построение отчетов и исследование данных с помощью Cognos Business Insight Advanced и Report Studio;

• создание сложных интерактивных панелей управления (*dashboard*) на основе Cognos Business Insight;

• сценарный анализ с использованием Cognos TM1;

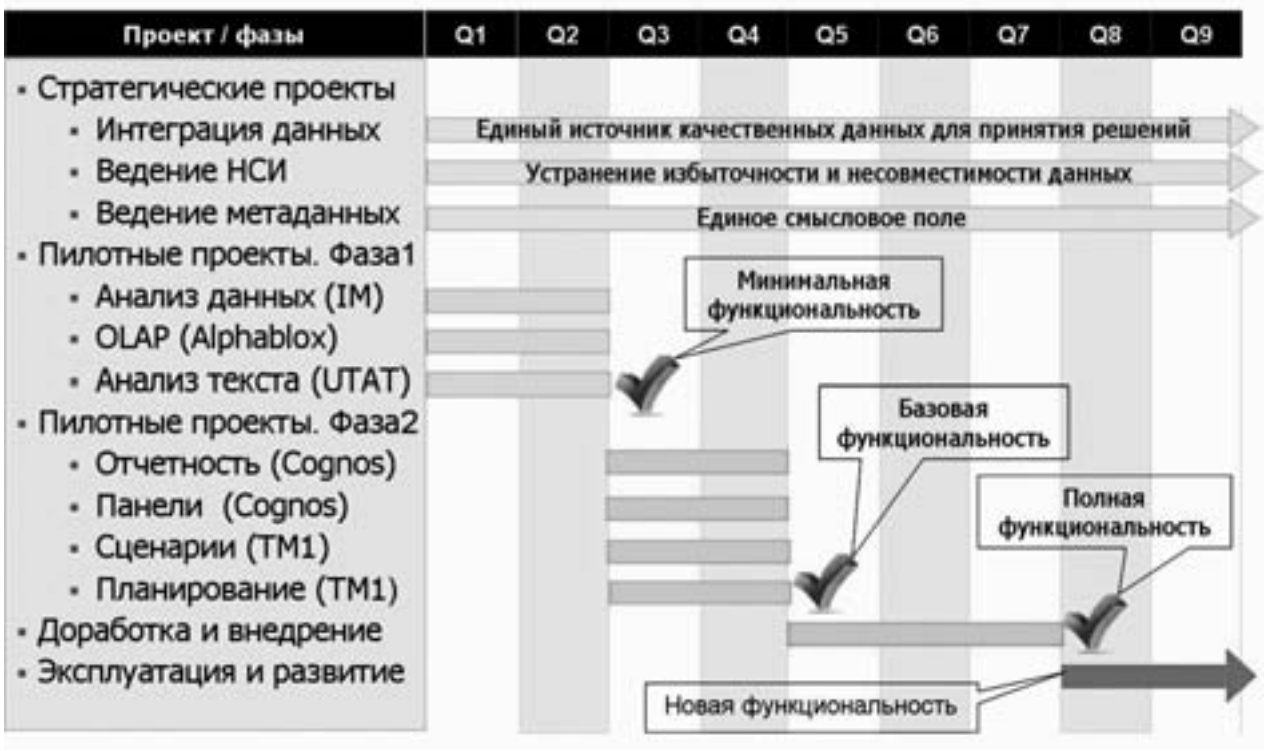

Рис. 7. План создания КХД (Q1—Q9 — время исполнения проекта в кварталах)

• корпоративное планирование с помощью Cognos TM1.

После завершения второй фазы пилотных проектов необходимо вновь выполнить анализ результатов выполнения проектов. Следующим шагом должно быть создание полнофункционального КХД, что невозможно без полноценной поддержки со стороны централизованной среды управления данными, метаданными и НСИ.

Итак, примерный план по созданию КХД может выглядеть следующим образом:

- Стратегические задачи:
  - скоординированная интеграция данных, метаданных и НСИ;
- Тактические задачи:
  - выбор 2—3 проектов для демонстрации преимуществ КХД;
  - создание централизованной среды управления данными, метаданными и НСИ;
  - анализ результатов и изменение среды КХД при необходимости;
  - внедрение 3—4 проектов с учетом полученного опыта;
  - в случае успеха — развитие КХД в масштабах компании;
  - промышленная эксплуатация КХД и создание новых задач, постановка и решение которых стали возможными благодаря накопленному опыту эксплуатации КХД.

Корпоративное хранилище данных должно развиваться вместе с предприятием. Жизнь не стоит на месте, появляются новые задачи, новые информационные системы. Если эти системы могут предоставить информацию, важную для анализа данных в масштабах предприятия, эти новые системы должны быть подключены к КХД. С тем, чтобы не создавать интеграционных проблем, желательно все новые системы создавать, основываясь на возможностях централизованной среды управления данными, метаданными и НСИ.

В свою очередь, централизованная среда управления данными, метаданными и НСИ должна изменяться и совершенствоваться с учетом новых систем и требований. Поэтому работы по интеграции данных, метаданных и НСИ должны осуществляться, пока существует предприятие и ее информационные системы, что на рис. 7 условно показано стрелками, выходящими за пределы графика работ.

Заключение

Корпоративное хранилище данных, построенное в результате скоординированной интеграции данных, метаданных и НСИ, обеспечивает более высокое качество информационно-аналитического обслуживания при снижении затрат и сокращении сроков разработки и предоставляет возможность принятия решений на основе более точной информации.

Предлагаемый подход обеспечивает эффективную работу систем управления данными, метаданными и НСИ, устраняет сосуществование модулей с близкой функциональностью, снижает совокупную стоимость владения и повышает доверие пользователей к данным КХД. Интеграция данных, метаданных и НСИ, выполняемая параллельно с развитием функциональности КХД, позволяет реализовать согласованные архитектуры, окружение, жизненные циклы и ключевые возможности для хранилища данных и систем ведения метаданных и НСИ.

Список литературы

1. **Асадуллаев С.** Система сбора и анализа первичных данных — I. Постановка задачи, сбор и хранение данных. 2011. URL: http://www.ibm.com/developerworks/ru/library/sabir/warehouse-1/index.html

2. **Асадуллаев С.** Система сбора и анализа первичных данных — II. Анализ первичных данных. 2011. URL: http://www.ibm.com/developerworks/ru/library/sabir/warehouse-2/index.html

3. **IBM** Forms documentation. 2010. URL: http://www.ibm.com/developerworks/lotus/documentation/forms/

4. **InfoSphere** Warehouse overview 9.7. 2010. URL: http://publib.boulder.ibm.com/infocenter/idm/v2r2/index.jsp?topic =/com.ibm.isw.release.doc/helpindex_isw.html

5. **Cognos** Business Intelligence. IBM. 2010. URL: http://publib.boulder.ibm.com/infocenter/cfpm/v10r1m0/index.jsp?topic=/com.ibm.swg.im.cognos.wig_cr.10.1.0.doc/wig_cr_id111gtstd_c8_bi.html

6. **Cognos** TM1. IBM. 2010. URL: http://www-01.ibm.com/software/data/cognos/products/tm1/

7. **Асадуллаев С.** Данные, метаданные и НСИ: тройная стратегия создания хранилищ данных. 2009. URL: http://www.ibm.com/developerworks/ru/library/r-nci/index.html.

8. **Асадуллаев С.** Архитектуры хранилищ данных — III. 2009. URL: http://www.ibm.com/developerworks/ru/library/sabir/axd_3/index.html.

9. **Leong-Hong B. W., Plagman B. K.** "Data Dictionary/Directory Systems", John Wiley & Sons, 1982. (Леонг-Хонг Б., Плагман Б. Системы словарей-справочников данных. М.: Финансы и статистика, 1986).

10. **Hackathorn R.** Data Warehousing Energizes Your Enterprise // Datamation, 1995. Feb. 1. P. 39.

11. **Асадуллаев С.** Управление качеством данных с помощью IBM Information Server. 2010. URL: http://www.ibm.com/developerworks/ru/library/sabir/inf_s/index.html

12. **Mastering** financial systems success. 2009. Financial Service Technology. URL: http://www.usfst.com/article/Issue-2/Business-Process/Mastering-financial-systems-success/

13. **Brooks F. P.** The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Professional; Anniversary edition, 1995. (Брукс Ф. Мифический человеко-месяц или Как создаются программные системы. М.: Символ-Плюс, 2010).

УДК 004.056.52 004.056.53

А. А. Иткес, мл. науч. сотр., НИИ механики МГУ им. М. В. Ломоносова,
e-mail: itkes@imec.msu.ru

Программный комплекс Nettrust для управления доступом к ресурсам распределенных информационных систем на основе отношений доверия

Логическое разграничение доступа является важным аспектом обеспечения безопасности информационных систем, в том числе, распределенных. Доступ субъекта к удаленному объекту в таких системах, как правило, осуществляется посредством другого субъекта, именуемого доверяющим субъекту, который осуществляет доступ. По этой причине для управления доступом к объектам распределенных систем можно использовать управление доверием между субъектами таких систем. Для этой цели разработан программный комплекс Nettrust, который подробно рассматривается в данной работе.

Ключевые слова: *информационная безопасность, логическое разграничение доступа, отношения доверия*

Введение

В течение последних лет сфера применения сложных, территориально распределенных информационных систем неуклонно расширяется. Одновременно с этой тенденцией возрастает и величина потенциального ущерба, который может быть причинен в результате некорректной работы подобных систем. Сбои в их работе могут возникать как случайно, не по злому умыслу, так и в результате намеренных действий как лиц, внешних по отношению к системе, так и администраторов, ответственных за те или иные аспекты сопровождения системы. Диапазон потенциальных нарушителей с точки зрения их целей и возможностей достаточно широк. Он простирается от подростков, ищущих приключений, до террористических и военно-политических организаций, имеющих целью на длительное время вывести из строя жизненно важные для страны информационные системы, причинив государству как можно больший ущерб [1]. Такой ущерб может иметь не только экономические, но и социальные, политические последствия. Отказы или сбои в работе современных информационных систем могут привести к экологическим или техногенным катастрофам, к человеческим жертвам и, учитывая тенденции развития общества и темпы развития телекоммуникаций, в будущем риски подобных событий увеличатся.

При анализе методов защиты сложных информационных систем часто упоминается "правило слабого звена". Оно заключается в том, что система защищена настолько надежно, насколько можно гарантировать защиту самого слабого его компонента. Во многом указанное утверждение справедливо. Однако не следует пренебрегать защитой системы в целом в том случае, если часть ее оказывается под контролем злоумышленников. Существует несколько причин для возникновения подобной постановки задачи. Первая из них состоит в том, что как бы хорошо ни были защищены компоненты распределенной информационной системы от внешней угрозы, злоумышленник может получить контроль над одним из них в каком-то смысле "случайно". Вторая причина заключается в том, что зачастую организации сталкиваются с нападениями не только внешних, но и внутренних нарушителей. Не следует также упускать из вида "комбинированные" угрозы, исходящие как извне, так и изнутри организации. Если противник — террористическая организация или крупная преступная группировка, то вполне вероятно, что перед нападением злоумышленники попытаются подкупить кого-либо из служащих организации, имеющей прямое отношение к процессу эксплуатации системы. Третья причина, по которой необходимо разрабатывать меры безопасности, предназначенные для их использования, в том числе и в условиях, когда некоторые из компонентов системы контролиру-

ются злоумышленником, состоит в том, что многие крупные распределенные информационные системы создаются путем объединения более мелких составляющих их компонентов. До объединения эти подсистемы могут иметь различные уровни защищенности. Некоторые из них, по крайней мере на начальном этапе их совместной работы, могут быть защищены недостаточно надежно.

С учетом изложенного, представляет практический интерес разработка способов построения политик безопасности сложно организованных, территориально распределенных информационных систем. Такие подходы и способы, с одной стороны, должны минимизировать взаимозависимость компонентов такой системы и уменьшать возможный ущерб от несанкционированного доступа к некоторым из них. С другой стороны, они призваны сделать компоненты достаточно сильно связанными для того, чтобы система в целом могла успешно выполнять свои основные функции. Необходимо уделить внимание защите гетерогенных систем, отдельные компоненты которых имеют различные, не согласованные друг с другом политики информационной безопасности. Однако пользователи этих систем по объективным причинам вынуждены участвовать в интенсивном обмене информацией посредством незащищенных каналов сетей связи общего пользования.

Важно отметить, что доступ субъекта к удаленному на сетевой среде объекту в распределенных системах, как правило, происходит посредством другого субъекта, который в контексте данной работы называется доверяющим субъекту, осуществляющему доступ. По этой причине для управления доступом субъектам к объектам распределенных информационных систем целесообразно использовать механизмы управления доверием между субъектами этих систем.

В настоящей работе представлен разработанный автором программный комплекс, называемый Nettrust, служащий для управления доступом к удаленным на сетевой среде объектам распределенных информационных систем на основе отношений доверия между их субъектами.

1. Модели и механизмы логического разграничения доступа

Настоящий раздел содержит описание некоторых важных в контексте целей работы свойств используемых на практике моделей и реализующих их механизмов логического разграничения доступа (в дальнейшем — ЛРД) субъектов к объектам распределенных информационных систем. Основное внимание в данном разделе уделено трем группам моделей ЛРД, которые в контексте настоящей работы будем также называть базовыми:

- ролевые модели ЛРД;
- модели ЛРД принудительной типизации;
- многоуровневые модели ЛРД.

Определения и некоторые свойства данных моделей ЛРД представлены в работе [3]. Для понимания дальнейших исследований, изложенных в настоящей работе, приведем краткие сведения об этих моделях.

При использовании **ролевой модели** ЛРД в подконтрольной информационной системе задаются множества P, R, U и S, называемые, соответственно, множествами привилегий, ролей, пользователей и субъектов. Каждой роли при этом соответствует множество привилегий, а каждому субъекту — множество ролей. Роли, таким образом, используются в качестве "промежуточного звена" между субъектами и привилегиями.

При использовании **модели ЛРД принудительной типизации** в подконтрольной информационной системе задаются следующие множества T, $D \subset T$, C и A, называемые, соответственно, множествами типов, доменов, классов и видов доступа. При этом каждый объект имеет свои тип и класс, каждый субъект — свой домен, а специальная таблица определяет права доступа субъектов каждого домена к объектам каждого типа и класса.

При использовании **многоуровневой модели ЛРД** в подконтрольной системе задается решетка (частично упорядоченное множество) L и каждому объекту o ставится в соответствие некоторый ее элемент $l(o)$, называемый уровнем этого объекта. Доступ субъекта s к объекту o на чтение разрешается в том и только том случае, если $l(s) \geqslant l(o)$, а на запись — в том и только том случае, если $l(s) < l(o)$. Данная модель является несколько менее выразительной по сравнению с двумя другими, однако она обладает повышенной устойчивостью к некоторым видам атак.

Кроме математических моделей логического разграничения доступа, в контексте целей данной работы интерес представляют программные механизмы, активно используемые в настоящее время для управления доступом к ресурсам информационных систем под управлением операционной системы (в дальнейшем — ОС) Linux. К их числу относятся базовые механизмы разграничения доступа, использующиеся в UNIX-подобных операционных системах, к которым относится Linux, а также ряд дополнительных программных средств с открытым исходным кодом для обеспечения безопасности информационных систем под управлением ОС Linux. К наиболее распространенным таким средствам относятся программные механизмы SELinux, GRSecurity и RSBAC.

Базовые механизмы ЛРД операционной системы Linux реализуют в подконтрольной информационной системе, так называемую **усеченную дискреционную модель ЛРД**. При использовании данной модели каждому объекту подконтрольной системы соответствует определенный пользователь-владелец, который имеет право изменять атрибуты безопасности этого объекта. При этом данная модель ЛРД регламентирует три вида доступа к объектам — чтение, запись и исполнение. Для каждого объекта определяется, какие из перечисленных прав доступа к этому объекту имеет его владелец, некоторая выделенная группа пользователей и остальные пользователи. Кроме того, в системе имеется так называемый "суперпользователь",

имеющий все права доступа ко всем объектам, а также право осуществлять ряд манипуляций с настройками системы, недоступных другим пользователям.

Выразительных свойств данной модели оказывается недостаточно для решения ряда актуальных в настоящее время задач безопасности. Например, она не позволяет разрешить какому-либо пользователю добавление информации к некоторому объекту, но может запретить модификацию информации, записанной в этот объект ранее. Кроме того, ограничения на права "обычных" пользователей приводят к необходимости запускать многие приложения от имени администратора, что приводит к дополнительным угрозам безопасности подконтрольной системы. В силу изложенных выше причин ОС Linux нуждается в более совершенных средствах обеспечения безопасности, реализующих более сложные модели ЛРД.

С этой целью в ОС Linux используется ряд дополнительных программных механизмов ЛРД, из которых три наиболее распространенных перечислены выше. Правила доступа субъектов к объектам имеют в этих трех системах различный и довольно сложный вид, однако, в следующем разделе данной работы отмечено, что модели ЛРД, реализуемые всеми тремя этими программными средствами могут быть представлены в терминах модели принудительной типизации.

2. Представление одних моделей логического разграничения доступа в терминах других

Как отмечено во введении, основной целью настоящей работы является создание программного комплекса, служащего для управления доступом к объектам распределенных информационных систем в соответствии с заданными моделями ЛРД с помощью отношений доверия. В связи с тем, что данный комплекс, представленный далее в разд. 4, для управления доступом к объектам в пределах локальных подсистем использует программные средства, перечисленные в разд. 1, для обоснования корректности его функционирования необходимо исследовать возможности представления моделей ЛРД, которые реализуются этими программными средствами, в терминах базовых математических моделей ЛРД. Кроме того, возможность представления одних базовых математических моделей ЛРД в терминах других позволит существенно упростить построение необходимых и достаточных условий возможности объединения этих моделей ЛРД, которые представлены в разд. 3.

Вопросы представления одних базовых математических моделей ЛРД в терминах других рассматриваются в работе [3], где доказано, что:

- любая многоуровневая модель ЛРД может быть представлена в терминах ролевой модели ЛРД;
- любая многоуровневая модель ЛРД может быть представлена в терминах модели принудительной типизации;
- любая модель принудительной типизации может быть представлена в терминах ролевой модели ЛРД.

Рассуждения, представленные в работе [3], активно используются в разд. 3 данной работы.

Рассмотрим возможности представления моделей ЛРД, реализуемых программными механизмами, перечисленными в разд. 1, в терминах базовых моделей ЛРД. Такое представление необходимо для того чтобы доказательно обосновать возможность объединения моделей ЛРД, реализуемых данными программными средствами. Начнем с усеченной дискреционной модели ЛРД, реализуемой традиционными механизмами ОС Linux.

В общем случае дискреционные модели ЛРД не могут быть выражены через модель принудительной типизации или ролевые модели ЛРД. Данное утверждение следует из того, что модель принудительной типизации предполагает изменение типов объектов только по определенным правилам, в то время как дискреционные модели ЛРД позволяют пользователям системы изменять атрибуты безопасности некоторых объектов по своему усмотрению. С учетом отмеченных обстоятельств, возникает необходимость в разработке новой математической модели ЛРД, которая, с одной стороны, являлась бы дискреционной в широком смысле, а с другой стороны, ее определяющие множества были бы аналогичны соответствующим множествам модели принудительной типизации. Отмеченные свойства позволили бы распространить на данную модель необходимые и достаточные условия возможности объединения моделей принудительной типизации, которые представлены в разд. 3.

Автором была разработана модель ЛРД, обладающая указанными выше свойствами, которая именуется моделью гибкой типизации. При использовании этой модели в подконтрольной информационной системе определяются такие же множества, что и при использовании модели принудительной типизации, а также множество U, называемое множеством пользователей. Каждому объекту при этом соответствует пользователь, называемый владельцем этого объекта. Владельцы всех объектов одного типа совпадают, и владелец каждого объекта имеет возможность изменять тип этого объекта в рамках заданных правил. Автором доказано, что любая усеченная дискреционная модель ЛРД может быть выражена в терминах модели ЛРД гибкой типизации. Данное утверждение используется для математического обоснования корректности функционирования программного комплекса Nettrust при объединении моделей ЛРД, реализуемых традиционными механизмами ОС Linux.

Для дальнейшего изложения необходимо также рассмотреть вопросы представления моделей ЛРД, реализуемых программными системами SELinux, GRSecurity и RSBAC, в терминах базовых математических моделей ЛРД. В работе [5] показано, что модели ЛРД, реализуемые программными средствами GRSecurity и RSBAC могут быть представлены в терминах модели ЛРД, реализуемой SELinux. Кроме того, автором данной работы доказано утверждение о том, что модель ЛРД, используемая SELinux, может

быть представлена в терминах модели принудительной типизации [6]. Из этих двух утверждений очевидным образом следует, что модели ЛРД, реализуемые всеми тремя программными механизмами SELinux, GRSecurity и RSBAC, могут быть представлены в терминах модели принудительной типизации.

Результаты, полученные в настоящем разделе, позволяют обосновать корректность взаимодействия программного комплекса Nettrust, разработке которого посвящен разд. 4, с традиционными механизмами ЛРД ОС Linux, а также с программными средствами SELinux, RSBAC и GRSecurity.

3. Отношения доверия

В настоящем разделе рассматривается разработанный автором метод объединения математических моделей логического разграничения доступа субъектов к объектам распределенных информационных систем на основе отношений доверия между субъектами этих систем. Понятие отношения доверия между субъектами необходимо для рассмотрения механизмов управления доступом субъектов к объектам распределенной на сетевой среде сложной информационной системы. Причина в том, что субъекты такой системы могут осуществлять доступ к объектам, находящихся в других компонентах этой системы только посредством других ее субъектов. Иными словами, доступ субъектов к удаленным на сетевой среде объектам распределенной информационной системы невозможен без использования механизмов доверия между субъектами данной системы. В связи с этим обстоятельством возникает необходимость в формализации и в дальнейшем исследовании понятия доверия между субъектами распределенной информационной системы.

Отношением доверия между информационными системами A и B называется подмножество $T_{A,\ B} \subset \subset S(A) \times S(B)$.

Смысл указанного отношения в том, что если пара (a, b) принадлежит $T_{A,\ B}$, то субъект a может получить доступ к объектам системы B посредством субъекта b. В этом случае будем считать, что субъект b доверяет субъекту a. Например, при загрузке файла по протоколу HTTP серверное приложение доверяет браузеру, поскольку браузер осуществляет доступ на чтение к удаленному объекту посредством сервера.

Кроме отношения доверия, важными для работы с моделями логического разграничения доступа субъектов к объектам распределенных информационных систем являются понятия привилегии и корректного множества привилегий. Привилегией при использовании многоуровневой модели ЛРД и модели принудительной типизации называется такое множество пар (*объект*, *вид доступа*), что каждый субъект подконтрольной информационной системы либо имеет право соответствующего доступа ко всем объектам из этого множества, либо не имеет ни к одному. Корректным множеством привилегий для ролевой модели ЛРД и модели принудительной типизации называется такое множество привилегий P, для которого некоторое множество субъектов может обладать всеми привилегиями, входящими в P и никакими другими.

С использованием понятия корректного множества привилегий в работе [3] доказано следующее утверждение.

Теорема:

Пусть информационные системы A и B имеют политики безопасности, основанные на ролевой модели ЛРД. Пусть система C состоит из объектов систем A и B, и также имеет политику безопасности, основанную на модели ролевой ЛРД. Пусть множество привилегий системы C имеет вид $P(C) = P(A) \cup P(B)$ *и ограничение модели ЛРД системы C на каждую из подсистем совпадает с локальной моделью ЛРД этой подсистемы. В этом случае объединение моделей ЛРД систем A и B в модель ЛРД системы C может быть выражено с помощью отношений доверия тогда и только тогда, когда для любой роли r системы C набор ее привилегий* $P(r)$ *имеет вид* $P(r) = P_A(r) \cup P_B(r)$, *где множества привилегий* $P_A(r) = P(r) \cap P(A)$ *и* $P_B(r) = P(r) \cap P(B)$ *корректны с позиций локальных моделей ЛРД систем A и B.*

Аналогичный факт имеет место и для модели принудительной типизации.

Теорема:

Пусть информационные системы A и B имеют политики безопасности, основанные на модели принудительной типизации. Пусть система C состоит из объектов систем A и B и также имеет политику безопасности, основанную на модели принудительной типизации.

При этом все три системы имеют одинаковые множества классов и видов доступа, причем класс каждого объекта в системе C такой же, как и в соответствующей этому объекту подсистеме. При этом пусть ограничение объединенной модели ЛРД на каждую из подсистем совпадает с локальной моделью ЛРД этой подсистемы. Подобное объединение моделей принудительной типизации может быть выражено с помощью отношений доверия тогда и только тогда, когда набор прав доступа каждого субъекта к объектам каждой из подсистем представляется корректным подмножеством привилегий этой подсистемы.

Таким образом, в большинстве случаев ролевые модели и модели принудительной типизации могут быть объединены с помощью отношений доверия в широком наборе случаев. Одновременно с этим, многоуровневые модели ЛРД могут быть объединены с помощью отношений доверия только в частных случаях. Данное утверждение строго сформулировано и доказано в работе [3].

Как отмечено в разд. 1, большинство информационных систем, функционирующих под управлением ОС Linux, используют для управления доступом субъектов к объектам дискреционные модели ЛРД, которые реализуются механизмами ЛРД UNIX-подобных ОС. По этой причине исследование возможностей объединения дискреционных моделей ЛРД имеет важное значение для обоснования корректности функционирования программных средств управления доступом субъектов к удаленным объектам в составе

распределенных информационных систем, в том числе для обоснования корректности функционирования программного комплекса Nettrust, результаты разработки которого представлены в разд. 4.

Для исследования вопросов объединения моделей ЛРД гибкой типизации необходимы понятия привилегии и корректного множества привилегий, аналогичные соответствующим понятиям, которые используются при исследовании возможности объединения моделей ЛРД принудительной типизации.

С использованием этих понятий сформулировано и доказано необходимое и достаточное условие, гарантирующее возможность объединения моделей ЛРД гибкой типизации с помощью отношений доверия, полностью аналогичное соответствующему условию для моделей принудительной типизации. Это условие служит обоснованием для анализа корректности функционирования, рассмотренного в разд. 4 программного комплекса Nettrust, в составе распределенных на сетевой среде информационных систем в случае, если компоненты этих систем используют для управления доступом субъектов к объектам модель гибкой типизации, частным случаем которой является усеченная дискреционная модель ЛРД UNIX-подобных ОС.

Отметим, что при использовании существующих на практике информационных систем, множества их субъектов изменяются во времени и крайне затруднительно обеспечивать своевременную модификацию настроек доверия при изменении, создании или удалении субъектов. По этой причине на практике в системе следует задавать отношения доверия не между субъектами информационных систем, а между группами субъектов. При этом субъекты следует разделять на группы таким образом, чтобы, с одной стороны, множества таких групп редко изменялись во времени, и, с другой стороны — доверие между всеми субъектами в рамках одной такой группы не снижало бы защищенность информационной системы.

Доменом называется такое множество субъектов d, что любые два субъекта из d имеют одинаковые права доступа к любому объекту. Множество доменов называется корректным, если его элементы не пересекаются и их объединение составляет все множество субъектов подконтрольной информационной системы. Отметим, что при использовании многоуровневой модели ЛРД доменом является множество субъектов, относящихся к одному уровню, а при использовании ролевой модели ЛРД — множество субъектов, имеющих одинаковый набор ролей.

Пусть A и B — информационные системы, $D(A)$ и $D(B)$ — корректные множества доменов этих систем. Контекстным отношением доверия между системами A и B называется подмножество $CT_{A,\,B} \subset D(A) \times D(B)$. При этом каждому контекстному отношению доверия $CT_{A,\,B}$ соответствует обычное отношение доверия $T_{A,\,B} = \{(a, b): a \in d_A,\ b \in d_B\ (d_A, d_B) \in CT_{A,\,B}\}$.

Понятие контекстного отношения доверия является важным при разработке программных средств управления доступом субъектов к объектам в распределенных информационных системах. Отмеченный факт имеет место в связи с тем обстоятельством, что множества субъектов информационных систем меняются во времени произвольным образом, в то время как множества доменов могут быть изменены только администратором системы при модификации политики безопасности. По этой причине при настройке сложной распределенной информационной системы более целесообразно указывать допустимые отношения доверия между доменами, а не между субъектами ее подсистем.

В заключение данного раздела следует определить еще одно понятие, называемое двухуровневым отношением доверия, которое является обобщением понятия отношения доверия, представленного в начале раздела. Использование двухуровневых отношений доверия позволяет формализовать частичное доверие между субъектами распределенных информационных систем, более распространенное на практике по сравнению с полным доверием. Двухуровневое отношение доверия определяется следующим образом.

Пусть A и B — информационные системы, $S(A)$ и $S(B)$ — множества субъектов этих систем, $D(B)$ — множество доменов системы B. Двухуровневым отношением доверия между системами A и B называется такое множество $T2_{A,\,B} S(A) \times S(B) \times D(B) \times \{0, 1, 2\}$, что для любых субъектов a и b систем A и B существует не более одной такой пары (d, k), что $(a, b, d, k) \in T2_{A,\,B}$. При этом число k называется уровнем доверия для пары субъектов a и b и обозначается как $TL(a, b)$.

Смысл двухуровневого отношения доверия в том, что, если $TL(a, b) = 0$, то b не принимает от a никаких запросов, если $TL(a, b) = 2$, то b доверяет a в обычном смысле. В случае, если $TL(a, b) = 1$, субъект b принимает запросы от субъекта a, однако не выполняет их, а порождает другой субъект b' с меньшими привилегиями, имеющий те же функциональные возможности, что и b, и доверяющий субъекту a в обычном смысле. При этом субъект b' получает такой домен d, что $(a, b, d, 1) \in T2_{A,\,B}$. Такой домен d называется доменом по умолчанию для субъектов a и b.

В дополнение к представленному определению, очевидным образом определяется понятие двухуровневого контекстного отношения доверия. Использование для интеграции моделей ЛРД двухуровневого контекстного отношения доверия в большей степени отвечает требованиям практики, чем использование других представленных в разд. 3 видов отношения доверия. Так, например, SSH-сервер при получении запроса от клиента порождает субъект с меньшими привилегиями, который и выполняет клиентские запросы, т. е. имеет место двухуровневое отношение доверия.

Результаты исследований, представленные в настоящем разделе и в разд. 2 служат обоснованием корректности функционирования программного комплекса Nettrust, который используется для управления логическим разграничением доступа к объектам распределенных информационных систем под управлением ОС Linux на основе доверия между субъектами этих систем. Архитектура этого программного средства рассматривается в следующем разделе.

4. Программный комплекс Nettrust

В настоящем разделе представлены результаты исследований, направленных на разработку дополнительного модуля к ядру ОС Linux, предназначенного для управления логическим разграничением доступа субъектов к объектам информационной системы, распределенной на сетевой среде, с помощью отношений доверия. К разрабатываемому программному комплексу предъявляются следующие основные требования:

- наличие строгой математической модели функционирования комплекса;
- поддержка большого числа различных моделей ЛРД;
- возможность эксплуатации комплекса в составе распределенной на сетевой среде информационной системы;
- возможность взаимодействия с внешними узлами, на которых Nettrust не используется;
- возможность изменения настроек доверия между узлами подконтрольной информационной системы без перезагрузки ее подсистем;
- возможность эксплуатации комплекса Nettrust без существенной модификации приложений в составе подконтрольной информационной системы;
- отсутствие центрального сервера управления настройками доверия, компрометация которого привела бы к компрометации всей распределенной информационной системы;
- возможность реализации двухуровневых отношений доверия с использованием программного комплекса Nettrust.

Математическое обоснование функционирования Nettrust представлено в первых трех разделах настоящей работы. Утверждения, описанные в этих разделах, в совокупности позволяют гарантировать, что Nettrust в сочетании с рассмотренными в разд. 1 программными средствами управления доступом субъектов к объектам информационных систем под управлением ОС Linux реализует в подконтрольной распределенной системе строгую математическую модель ЛРД.

Поскольку одним из перечисленных выше требований к комплексу является отсутствие центрального сервера управления доверием, для использования Nettrust соответствующее программное обеспечение должно быть установлено на каждом из компонентов подконтрольной распределенной информационной системы. При этом, в ходе разработки Nettrust принято решение о хранении в памяти каждого из узлов подконтрольной системы информации о том, каким субъектам других узлов доверяют субъекты данного узла. Очевидно, что для принятия модулем Nettrust решения о доверии, удаленному субъекту необходимо при установлении сетевого соединения каким-либо образом передать информацию о субъекте, инициировавшем соединение. При разработке комплекса автором было принято решение помещать метку безопасности субъекта, отправившего сетевой пакет, в поле опций заголовка IP. Данное поле имеет ограниченную длину, поэтому следует отказаться от передачи символьных имен меток безопасности (используемых, например, в SELinux) и оперировать только их целочисленными идентификаторами. Комплекс Nettrust использует специальные таблицы для преобразования идентификаторов доменов, используемых локальной подсистемой управления доступом, в метки, передаваемые по сети. При этом каждый из компонентов подконтрольной распределенной информационной системы может иметь несколько таблиц соответствия меток безопасности. Выбор таких таблиц для каждого конкретного сетевого соединения осуществляется исходя из IP-адреса взаимодействующего компонента. Таким образом, одна и та же подсистема может присваивать пакетам одного субъекта различные сетевые метки безопасности при взаимодействии с различными подсистемами. Данная возможность позволяет легко объединять сети, использующие различные соглашения о назначении сетевых идентификаторов доменов. Также возможно принимать решение о доверии в зависимости от используемого сетевого устройства, что дает подконтрольной системе некоторую защиту от атак с использованием поддельных IP-адресов.

Nettrust позволяет также использовать двухуровневые отношения доверия для контроля доступа субъектов к удаленным объектам. Для этого, однако, доверяющее приложение должно быть особым образом модифицировано и вызывать специальную функцию accept_nt, которая представляет собой аналог комбинации системных вызовов accept, fork и setuid/setgid. При этом в вызвавшем процессе дескриптор установленного соединения закрывается, так что субъект не может напрямую взаимодействовать с удаленным субъектом, которому доверяет не полностью. В настоящее время данная функция реализована в модуле ядра Nettrust в качестве системного вызова и работает только для TCP-соединений. Отметим, что реализация одноуровневых отношений доверия с помощью Nettrust не требует существенной модификации приложений в составе подконтрольной информационной системы.

В связи с тем обстоятельством, что комплекс Nettrust включает в себя модуль ядра ОС Linux, при разработке данного модуля необходимо кроме основных его функций, реализовать также механизмы, позволяющие администратору управлять поведением этого модуля. Для управления настройками доверия в ядре ОС используется протокол Netlink, который традиционно применяется в ОС Linux для управления настройками сетевой подсистемы ядра ОС. Данный протокол подробно рассматривается автором в работе [2]. Использование Netlink имеет несколько преимуществ перед синхронным обменом сообщениями. Одно из них — возможность выполнения приложением различных функций во время ожидания данных от ядра ОС. Еще одно преимущество заключается в том, что структура сообщений Netlink позволяет при модификации настроек не передавать ядру данные, не нуждающиеся в изменении, что позволяет не использовать выделенное значение, означающее "оставить без изменений".

Настройки доверия в ядре ОС представлены в виде трех групп таблиц и одного выделенного значения, определяющего тип локальной подсистемы контроля доступа субъектов к объектам. Первая группа содержит таблицы соответствия между доменами локальной подсистемы и их целочисленными идентификаторами, передаваемыми по сети. Вторая группа содержит информацию о смежных подсистемах, для взаимодействия с которыми используется Nettrust. Для каждой из них указываются такие данные, как диапазон IP-адресов удаленной системы, индекс сетевого устройства и номер используемой таблицы соответствия доменов. Третья группа данных содержит собственно информацию о доверии между субъектами. Записи этой группы включают идентификатор подсистемы-собеседника, к которой относится запись, а также уровень доверия и домена по умолчанию для данной пары доменов. Заметим, что домены как удаленного, так и локального субъекта в этой группе записей задаются значениями передаваемых по сети меток безопасности. Для локального субъекта такая метка хранится в таблице соответствия доменов, ассоциированной с данной подсистемой-собеседником.

Для управления Nettrust можно использовать как подсемейство Netlink, обозначаемое NETLINK_TRUST, так и подсемейство протокола Generic Netlink. Первый способ считается устаревшим и реализован в целях обратной совместимости с первым прототипом Nettrust. В обоих случаях для управления модулем Nettrust используется 14 команд, две из которых служат для изменения и получения типа используемой локальной подсистемы контроля доступа субъектов к объектам. Остальные двенадцать команд разделены на четыре группы. Первая группа команд используется для добавления, чтения или удаления целочисленного идентификатора домена. Вторая — для добавления, чтения или удаления информации о другом компоненте подконтрольной распределенной информационной системы. Команды из третьей группы служат для добавления, чтения или удаления информации о парах доверяющих друг другу субъектов. Команды четвертой группы используются для управления модулем преобразования транзитных пакетов, описанным далее.

Для удобного управления настройками доверия в ядре ОС были разработаны две прикладные программы. Одна из этих программ, называемая TrustML, используется для полной перезаписи настроек доверия. Данная утилита считывает новые настройки Nettrust из текстового файла, содержащего код на специально разработанном для данной цели языке, также называемом TrustML. Файл на языке TrustML состоит из четырех секций. Первая секция состоит из единственной команды, определяющей тип подсистемы, используемой для разграничения доступа субъектов к объектам на локальном узле. Вторая секция содержит определения используемых таблиц соответствия между локальными и сетевыми метками безопасности. Третья секция содержит определения используемых таблиц доверия между субъектами данного узла и субъектами других узлов распределенной информационной системы. В четвертой секции размещается некоторая информация о способах взаимодействия с узлами, не описанными в третьей секции. Вторая прикладная утилита называется TrustCTL и используется для проведения незначительных изменений в настройках доверия.

Кроме описанных выше прикладных программ, автором реализована также библиотека функций, называемая LibNettrust. Каждая из функций библиотеки LibNettrust преобразует свои аргументы в сообщение протокола Generic Netlink и посылает их ядру, либо получает пакет от ядра ОС, анализирует его структуру и возвращает некоторые из полученных значений.

Использование данной библиотеки лишает разработчиков приложений некоторых преимуществ использования Netlink перед синхронным обменом информацией. Однако, оно освобождает их от необходимости самостоятельно реализовывать обработку сообщений Generic Netlink, позволяя значительно упростить разработку некоторых несложных приложений.

Следует также отметить, что использование Nettrust вносит в подконтрольную распределенную информационную систему новые объекты, в числе которых таблицы соответствия между локальными и сетевыми идентификаторами доменов и таблицы отношений доверия. Этот факт означает, что для полноты используемой модели ЛРД необходимо задать права доступа субъектов к этим объектам. В настоящее время модификация таблиц соответствия между доменами и сетевыми идентификаторами доменов доступна только системному администратору, в то время как чтение указанной информации доступно всем пользователям. Настройки доверия между локальным и удаленным субъектом могут быть изменены, либо прочитаны только системным администратором, либо владельцем локального субъекта.

Отметим, что в случае, если подконтрольная распределенная информационная система является очень большой и сложно организованной, число таблиц соответствия доменов и таблиц отношений доверия должно быть очень велико и для их хранения требуется значительное количество памяти. Кроме того, в этом случае при изменении соглашений о назначении сетевых меток безопасности в одной из подсетей подконтрольной распределенной информационной системы, соответствующие изменения должны быть внесены в настройки отношений доверия всех узлов системы, что требует большого количества времени и увеличивает вероятность ошибки при настройке Nettrust. С учетом изложенного, необходимо включить в состав комплекса механизмы, позволяющие сократить стоимость перенастройки соглашений об использовании сетевых меток безопасности в подобных системах. С этой целью в Nettrust дополнительно реализована возможность преобразования меток безопасности при прохождении сетевого пакета через транзитный узел, соединяющий две подсети с различными политиками назначения сетевых меток безопасности.

В качестве иллюстрирующего можно рассмотреть следующий пример. Пусть некоторая организация полу-

чена объединением двух более мелких. Пусть в локальной сети первой организации учетным записям сотрудника Иванова соответствует сетевая метка безопасности, равная 100. Пусть в локальной сети второй организации учетным записям того же сотрудника соответствует сетевая метка безопасности, равная 1500. Пусть узел *A* подключен к первой сети, *C* — ко второй, а *B* — сразу к обеим. В этом случае узел *B* должен иметь учетную запись ivanov и две разные таблицы соответствия типов, используемые для взаимодействия с двумя разными подсетями. В первой из этих таблиц записи ivanov соответствует сетевая метка 100, а во второй 1500.

Функция преобразования меток безопасности в Nettrust заключается в том, что узел *B* при получении сетевого пакета от узла *A* к узлу *C* может изменить сетевую метку пакета по следующему правилу: сначала по исходной сетевой метке в первой таблице соответствия доменов ищется учетная запись, затем исходная метка заменяется на метку из второй таблицы, соответствующую данной записи. Таким образом, система *C*, подключенная только ко второй подсети, не имеет необходимости хранить две таблицы соответствия доменов, одна из которых использовалась бы для взаимодействия с узлами первой подсети, а вторая — с узлами второй подсети. При отсутствии функции преобразования меток транзитных пакетов у узла *B*, все узлы распределенной системы должны были бы иметь несколько таблиц соответствия доменов для корректной реализации модели ЛРД распределенной информационной системы.

Подводя итоги разработки программного комплекса Nettrust, следует еще раз отметить ряд ключевых моментов. К достоинствам данного программного комплекса следует отнести:

- наличие математически строгого обоснования корректности его функционирования;
- поддержку большого числа разных математических моделей и реализующих их программных механизмов управления доступом, в том числе в рамках одной распределенной информационной системы;
- наличие специальных механизмов, позволяющих упростить управление Nettrust в больших, сложно организованных распределенных информационных системах.

Разработанный комплекс прошел ряд тестовых испытаний, подтвердивших его соответствие всем требованиям, представленных в начале данного раздела.

В заключение перечислим некоторые направления дальнейшей доработки Nettrust. Первое из них — это создание автоматизированных средств согласования моделей ЛРД в пределах подконтрольной информационной системы. В настоящее время математический аппарат, лежащий в основе построения Nettrust гарантирует, что данный программный комплекс реализует в подконтрольной информационной системе строгую формальную модель ЛРД, однако, конкретный вид этой модели можно найти только аналитически. Создание автоматизированных средств согласования моделей ЛРД позволит значительно упростить настройку отношений доверия, снизив вероятность допустить при этом ошибку. Вторым направлением дальнейшей доработки Nettrust является создание механизмов криптографической защиты передаваемых по сети меток безопасности.

Заключение

В данной работе представлен прототип программного комплекса, служащего для управления доступом субъектов к удаленным на сетевой среде объектам распределенных информационных систем на основе доверия между субъектами этих систем. Указанный подход позволяет объединять модели разграничения доступа в сложных распределенных информационных системах, разные компоненты которых используют различные модели логического разграничения доступа из числа реализуемых программными средствами под управлением ОС Linux.

Автор выражает свою глубокую признательность доктору физико-математических наук, профессору Васенину Валерию Александровичу за постановку задач, многочисленные плодотворные обсуждения результатов исследования и помощь в подготовке статьи.

Список литературы

1. **Шапченко К. А., Андреев О. О., Савкин В. Б., Иткес А. А.** Специализированные дистрибутивы операционной системы Linux с повышенным уровнем защищенности / Критически важные объекты и кибертерроризм. Часть 2. Аспекты программной реализации средств противодействия. Под ред. В. А. Васенина. М.: МЦНМО. — 2008. — С. 168—216.

2. **Иткес А. А., Савкин В. Б.** Механизмы протокола Netlink для управления сетевой подсистемой ОС Linux // Информационные технологии, 2006. № 8. С. 26—32.

3. **Иткес А. А.** Объединение моделей логического разграничения доступа для сложно организованных распределенных информационных систем // Проблемы информатики. 2010. № 1. С. 85—95.

4. **Иткес А. А., Савкин В. Б.** К развитию механизмов разграничения доступа в распределенных информационных системах // Мат. конф. "Проблемы безопасности и противодействия терроризму". М.: МЦНМО, 2006. С. 349—367.

5. **Шапченко К. А.** К вопросу о средствах ОС Linux для управления доступом при использовании ролевых политик безопасности / Мат. конф. МаБИТ-2005. М.: МЦНМО, 2006.

6. **Иткес А. А.** Управление доступом к ресурсам распределенных информационных систем на основе отношений доверия: дис. ... на соискание ученой степени кандидата физ.-мат. наук.

УДК 519.683.5

А. В. Колганов, магистрант, Воронежский государственный университет,
e-mail: m@alexikolgariov.com

Эффективная реализация R-дерева для индексации часто меняющихся геопространственных данных

Анализируется использование индексной структуры "R-дерево" для построения программных систем, обрабатывающих часто меняющиеся геопространственные данные. Рассматриваются вопросы производительности структуры, а также возможные способы их разрешения. Для некоторых вариантов решения приводятся данные, рассчитанные в ходе работы. В результате проведенного исследования реализованы алгоритмы, позволяющие увеличить производительность структуры в некоторых частных случаях.

Ключевые слова: *индексация пространственных данных, часто меняющиеся данные, анализ производительности, R-деревья*

Введение

Задача индексации пространственных данных, т. е. данных с привязкой к координатам в определенной системе, активно изучалась в последние десятилетия. В качестве вариантов ее решения были выдвинуты различные структуры данных, такие как Quad-деревья [1] или R-деревья [2], которые в дальнейшем породили целое семейство древовидных структур, объединенных общими базовыми принципами. Эти структуры были подробно исследованы в соответствующих публикациях, а также описаны и резюмированы в работе [3]. К модификациям классического R-дерева относятся R^+-деревья [4], R*-деревья [5], cR-деревья [6] и т. д. [3]. Эти модификации увеличивают производительность всей структуры данных в целом:

- R^+-деревья направлены на увеличение скорости поиска точечных данных;
- R*-деревья используют более сложные эвристические алгоритмы, применяемые при построении дерева, что позволяет ускорить поиск данных;
- cR-деревья существенно изменяют способы расщепления узлов при вставке элементов, что также позволяет увеличить производительность структуры для поиска.

В ряде случаев классическое R-дерево и его модификации претерпевали изменения для обеспечения эффективной работы со специфическими задачами, возникающими при разработке современных систем [3]. К таковым задачам относится индексация пространственно-временных данных, т. е. пространственных данных, изменяющихся во времени [7—8]. Выполнение запросов, как по пространственным данным, так и по временным, широко используется при построении мобильных систем связи, геоинформационных систем (ГИС), систем мониторинга движения, а также в сервисах, основанных на местоположении (*Location-Based Services*). Такие системы предъявляют высокие требования к быстродействию индексных структур.

В некоторых случаях возникает необходимость индексировать сравнительно небольшие объемы пространственных данных, однако часто меняющихся во времени. При этом производительность обновления данных становится не менее важной, чем производительность их поиска. Такие задачи могут возникать в системах реального времени, где требуется отображать и/или анализировать часто изменяющуюся обстановку.

Настоящая статья исследует применимость классического R-дерева к индексации часто меняющихся данных и выдвигает возможные варианты увеличения производительности операции обновления данных. Для нескольких программных реализаций приведены численные результаты тестирования, а также графики, позволяющие прогнозировать работу в различных условиях. Также в этой статье численно уточняется вопрос улучшения структуры R-дерева при проведении большого числа последовательных обновлений данных и ее влияния на производительность.

1. Классическое R-дерево и его модификации

R-дерево — это структура, предложенная в работе [2] в целях ускорения поиска данных по их местоположению. Изначально Гуттман использовал его для работы со СБИС (сверхбольшими интегральными схемами). Впоследствии быстрый поиск прямоугольников нашел применение в обработке географических и, в более общем случае, пространственных данных, а также аудиовизуальной информации и информации, привязанной к временным интервалам [3].

R-дерево — древовидная сбалансированная структура данных, разбивающая пространство на множество иерархически вложенных прямоугольников. Оно поддерживает операции вставки, удаления, поиска данных и использует эвристические алгоритмы, нацеленные на минимизацию взаимного наложения прямоугольников, а также на уменьшение их площади.

В листьях дерева хранятся индексы данных вместе с минимальными габаритными прямоугольниками (МГП), им соответствующими. Каждый лист дерева может содержать не более M и не менее m индексных записей. Ветви дерева содержат указатели на дочерние ветви или листы, при этом их число также лежит в диапазоне $[m, M]$. Совместно с каждым указателем в нелистовом узле дерева хранится МГП всей ветви, на которую он указывает. МГП листа есть объединение всех МГП элементов, в нем находящихся. МГП ветви — объединение всех МГП ее дочерних ветвей или листов. Таким образом, можно ввести следующие формальные определения:

- Минимальным габаритным прямоугольником элемента x назовем прямоугольник $mbr(x)$ минимальной площади, содержащий все точки элемента x.
- Индексная запись I есть упорядоченная пара вида (r, i), где r — МГП записи, а i — ее идентификатор. При этом справедливо $mbr(I) = r$.
- Лист R-дерева L есть упорядоченная пара вида (r, υ), где r — МГП листа, а $\upsilon = (I_1 ... I_k)$, $k \in [m, M]$ — контейнер индексных записей. При этом справедливы соотношения: $mbr(L) = r$ и $r = \bigcup_{i=1}^{k} mbr(\upsilon_i)$.
- Ветвь R-дерева B есть упорядоченная пара вида (r, υ). Здесь r — МГП ветви; $\upsilon = (L_1 ... L_k)$, $k \in [m, M]$ или $\upsilon = (B_1 ... B_k)$, $k \in [m, M]$ — контейнер листьев или других ветвей. Справедливы равенства: $mbr(B) = r$ и $r = \bigcup_{i=1}^{k} mbr(\upsilon_i)$.
- R-дерево T-рекурсивная структура данных, представляющая собой:
  - пустую структуру в случае, если в ней нет индексных записей;
  - лист R-дерева в случае, если число содержащихся индексных записей не превышает M;
  - ветвь R-дерева в случае, если число содержащихся индексных записей превышает M.

В R-дереве нет реализации операции обновления объектов, эффективной с точки зрения затрат времени. Каждая операция обновления реализуется как последовательное удаление старого и добавление нового объекта. За счет таких манипуляций обновленный объект помещается в позицию дерева, наилучшим образом ему соответствующую.

Вставка в R-дерево аналогична вставке в B^+-дерево) [3], используемое для индексации одномерных данных. В дереве проводится поиск листа, который наилучшим образом подходит для добавления записи. Она вставляется в лист, после чего происходит иерархическое обновление всех узлов дерева от выбранного листа вверх по иерархии. При этом изменяются МГП всех узлов, содержащих данный лист. Если лист не может принять запись, так как уже является полным (содержит M записей), то проводится расщепление листа на два. При расщеплении используются некоторые эвристические алгоритмы, позволяющие выполнить его наилучшим образом, т. е. обеспечивая наилучшую структуру дерева после расщепления. В работе [2] предложено несколько таких алгоритмов различной сложности. Они работают за линейное или квадратичное время, если не принимать во внимание поиск наилучшего разбиения методом полного перебора. Оптимальные алгоритмы расщепления узлов исследовались в работах [9—10].

Следует обратить внимание, что при вставке элемента может проявляться цепная реакция, приводящая к расщеплению всех узлов дерева, начиная от заданного листа, заканчивая корнем. Таким образом, операция вставки в некоторых случаях может приводить к значительным затратам времени.

Удаление элементов R-дерева выполняется отлично от данной операции в B^+-дереве. Это отличие проявляется в подходе, применяемом для обработки случая незаполненных узлов. В B^+-дереве в этом случае проводится слияние двух соседних узлов, так как дерево хранит одномерную информацию и соседние узлы содержат последовательные записи. В R-дереве же данный подход неприменим вследствие многомерности хранимых данных. Слияние узлов дерева, находящихся на одном и том же уровне, возможно, но этот подход может привести к значительной потере производительности вследствие ухудшения структуры дерева. Поэтому в работе [2] предлагается использовать подход, основанный на вставке элементов. Этот подход заключается в удалении и последующей вставке элементов всех незаполненных узлов, которые встречаются при подъеме по иерархии от листа, в котором удалялась запись. У данного подхода есть следующие преимущества:

- операция вставки уже реализована и не требует модификации;
- операция вставки нацелена на поддержание хорошей структуры дерева, а значит, после нескольких последовательных операций удаления ее ухудшения происходить не будет.

Следует также учесть, что проявление цепной реакции возможно и во время удаления элемента. Эта реакция будет происходить в том случае, если все уз-

лы вверх по иерархии от того, из которого удаляется запись, содержат m записей. В худшем случае это приведет к удалению всех элементов дерева и, фактически, перестройке всей структуры.

Так как каждая операция обновления дерева подразумевает выполнение дорогостоящих операций удаления и вставки, то выполнение большого объема последовательных обновлений может приводить к большим потерям времени. Если в дереве обновляются все объекты и эти операции происходят достаточно часто, то выполнение последовательных операций обновления является менее эффективным, чем операция полной перестройки индекса. Последняя избавляет от необходимости удаления элементов по одному и сопутствующих этому затрат. Если же обновляются не все записи, но соотношение между их общим числом и числом обновляемых велико, то полная перестройка индекса подразумевает выполнение лишних операций вставки статичных элементов. Таким образом, чтобы ускорить обновление элементов дерева, нужно использовать подходы, не требующие использования операций удаления элементов или их вставки, и при этом не приводящие к вырождению структуры дерева.

Некоторые модификации R-дерева позволяют увеличить скорость вставки, удаления и обновления данных. Так гильбертово R-дерево [11] (по сути, ложная модификация B^+-дерева) позволяет ввести порядок узлов на каждом его уровне. Этот порядок дает возможность определить для каждого узла понятия его братьев и сестер, существование которых позволяет отложить использование операции разбиения узла при его переполнении: оно проводится только в случае, если все братья и сестры узла заполнены.

Следует отметить, что большинство исследований R-деревьев направлено на увеличение скорости поиска, улучшение структуры дерева или на достижение ее большей компактности. Это обусловлено актуальностью задач обработки больших объемов статических данных [3].

Механизмы работы с R-деревьями во внешней памяти были исследованы как в первой работе по этой тематике [2], так и в других, более специализированных (например [12]). Исследование [2] дает ответ на вопрос, какими должны быть параметры дерева M и m для обеспечения максимальной производительности при работе с деревом на диске. Отмечено, что при больших значениях m узлы дерева более подвержены расщеплению в силу большей их наполненности, что влияет на скорость вставки данных. При малых значениях m, напротив, наблюдается уменьшение числа операций расщепления, что, в свою очередь, увеличивает быстродействие вставки элементов. Также при малых значениях m наблюдается значительное увеличение производительности удаления данных, что связано с более редкими операциями "перевставки" элементов дерева. В итоге показано, что оптимальными с точки зрения быстродействия параметрами являются $M = 6$ и $m = \frac{M}{2} = 3$, обеспечивающие минимальное среднее время вставки и удаления записей.

Однако в случае систем реального времени, а также тонких прикладных ГИС, индексная структура и все данные обычно хранятся в основной памяти. В этом случае оптимальные значения параметров могут отличаться от указанных выше. Для того чтобы проверить быстродействие операций поиска и вставки в дереве при различных значениях m и M в оперативной памяти, были использованы следующие условия.

Пусть все объекты находятся на ограниченном рабочем поле размером a. Будем предполагать, что объекты являются материальными точками, т. е. их габаритные прямоугольники имеют высоту и ширину равные нулю. Общее число объектов, вставляемых в дерево, равняется c_{item}. Все объекты равномерно распределены по рабочему полю. Размер области поиска выражается отношением k его линейных размеров к размеру рабочего поля. Для каждой пары исследуемых параметров осуществляется вставка объектов в пустое дерево с замером затраченного времени. После того, как дерево заполнено, выполняется c_{search} итераций оконного поиска.

Для того чтобы получить более гладкие данные, описанные выше операции проводились несколько десятков раз для каждой пары (m, M). Также отметим, что варьировались лишь значения M, а значения m определялись по формуле $m = \frac{M}{2}$. Полученные результаты приведены на рис. 1.

Из этих данных следует, что оптимальными с точки зрения скорости вставки и поиска являются параметры $m = 4$ и $M = 8$. Эти параметры и были использованы при последующей работе с деревом.

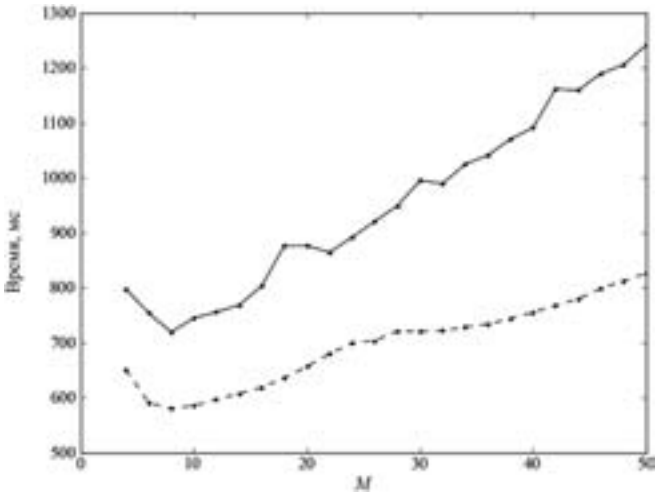

Рис. 1. Зависимость времени вставки и поиска данных от пар параметров (m, M): сплошная линия — время поиска, штриховая линия — время вставки. $a = 10^6$, $c_{item} = 10^5$, $c_{search} = 10^5$, $k = 0{,}1$

2. Оптимизация обновления в R-дереве

В общем случае оптимизировать операцию обновления дерева так, чтобы полностью избежать необходимости перемещения записи внутри него, не представляется возможным. Это обстоятельство объясняется тем, что иерархическая организация МГП подразумевает нахождение объектов, находящихся далеко друг от друга в пространстве, в разных ветвях дерева. Можно однако рассматривать эту операцию в контексте реальных данных, используемых в некоторых прикладных задачах, например, в отслеживании перемещения транспорта. В этом случае можно ввести операцию обновления таким образом, что она будет обеспечивать относительно высокую скорость модификации данных, не сильно снижая эффективность всей структуры при поиске.

Как было описано ранее, операция обновления в R-дереве состоит из двух последовательных операций: удаления элемента, а затем вставки обновленного элемента. Рассматривая алгоритмы подробнее, модификацию элемента в классическом дереве можно разделить на несколько этапов.

1. Поиск листа в дереве, в котором находится данный объект.

2. Удаление элемента из листа, затем выполнение операций по обновлению МГП, а также по обработке случаев незаполненных узлов.

3. Поиск листа, который наилучшим образом соответствует обновленному объекту.

4. Вставка элемента в лист, затем выполнение необходимых операций по обновлению МГП, а также обработке случаев переполненных узлов.

Первый подход к ускорению обновления дерева состоит в следующем. Предполагая, что работа проводится с точечными данными, можно смягчить требования к листьям R-дерева. Таким образом, если МГП обновленного элемента находится внутри МГП листа, то перевставка соответствующей записи выполняться не будет. Вместо этого проводится лишь ее модификация. Если МГП объекта после обновления не будет принадлежать листу в котором он находится, то для него вызывается обычная операция обновления дерева. Таким образом, габаритный прямоугольник в листьях дерева редко будет минимальным, что повлияет на скорость поиска данных, так как увеличит перекрытие областей МГП разных узлов. Однако так как МГП не будет увеличиваться, а после нескольких последовательных обновлений одного и того же элемента он будет вставлен в дерево заново, этот подход можно рассматривать как компромисс между производительностью поиска и обновления.

Второй подход представляет расширение первого. Если обрабатываемые данные разрежены по рабочему полю, то в листьях дерева можно хранить не МГП, а прямоугольник, полученный из МГП расширением границ на некоторую величину Δ_{MBR}. Если эта величина будет обеспечивать использование вышеописанного метода хотя бы при одном обновлении, то это позволит дополнительно увеличить производительность структуры в худшем случае (когда все объекты стремятся покинуть МГП листа). Однако подобный подход может существенно повлиять на общую производительность, так как он сознательно увеличивает МГП.

Также можно предложить третий подход к решению задачи: МГП расширяются не сразу, а только после того, как один из объектов действительно этого потребует. Причем расширение проводится на величину, суммарно не превосходящую Δ_{MBR}. Данный подход более эффективен с точки зрения поиска, но приводит к большим затратам во время операции обновления, поэтому его рассмотрение не входит в рамки данной статьи.

В итоге основные шаги алгоритма выглядят следующим образом.

1. Поиск в дереве листа, в котором находится данный объект.

2. Если новый МГП объекта лежит в МГП листа, обновление данных и выход из процедуры. Если нет — переход к п. 3.

3. Если $\Delta_{MBR_1} + \Delta_{MBR_2} \leqslant \Delta_{MBR}$, то расширение МГП листа, обновление МГП вверх по иерархии и выход из процедуры. Если нет, переход к п. 4. Здесь Δ_{MBR_1} — расширение, уже примененное к листу, а Δ_{MBR_2} — расширение, необходимое чтобы элемент остался в листе.

4. Осуществление обычной операции обновления дерева с учетом того, что лист, хранящий обновляемую запись, уже найден.

3. Модель данных

Следующая модель данных использовалась автором для упрощенного представления среды, в которой может проводиться мониторинг движения транспортных средств. Модель содержит как статичные объекты, так и движущиеся, предполагается, что объекты движутся почти прямолинейно, с заданной средней скоростью.

Предположим, что каждый объект данных является точечным, т. е. описывается парой координат (x, y). Назовем объект динамическим, если он будет изменяться при операции обновления. Отношение числа изменяемых объектов к числу неизменяемых выражается коэффициентом k_{dyn}. Каждый динамический объект имеет основное направление d и базовую скорость s. Значения этих величин выбираются случайным образом. Направление d равномерно распределено на отрезке $[0, 2\pi]$. Базовая скорость s распределена по нормальному закону с математическим ожиданием $M[s]$ и средним квадратическим отклонением $\sigma(s)$. При каждой итерации объект перемещается в основном направлении, отклоняясь от него на величину δ_d, распределенную равномерно на отрезке $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. Аналогично на каждой итерации объект отклоняется

от скорости на величину δ_s, распределенную по нормальному закону с математическим ожиданием $M[\delta_s] = 0$ и средним квадратическим отклонением $\sigma(\delta_s) = \frac{M[s]}{3}$.

Предположим, что рабочее поле представляет собой проекцию тора на плоскость. Такая интерпретация означает, что если объект уходит за одну из границ поля, он появляется на противоположной с тем же вектором перемещения. Таким образом, ни один из объектов при модификации не сможет покинуть рабочую область.

Для каждого случайного набора данных проводится c_u последовательных итераций обновления динамических элементов и затем c_s итераций поиска объектов по области.

4. Результаты замеров производительности и анализ

Замеры производительности программ осуществлялись для трех версий R-дерева: обычное дерево и деревья, описанные в первом и втором подходах при $\Delta_{MBR} = 50$ (см. разд. 2). Были выбраны значения $M[s] = 10$ и $\sigma(s) = 3$, что позволило оценить производительность структуры в лучшем для второго подхода случае (гарантирующем как минимум несколько итераций обновления без перемещения данных между узлами дерева). Для остальных параметров были выбраны значения $k_{dyn} = 0{,}75$, $c_u = 10^5$, $c_s = 10^5$. Замерялось время, затраченное на обновление и поиск данных. Так как данные выбираются случайно, полный процесс замера был многократно повторен, чтобы сгладить возможные колебания. Все значения,

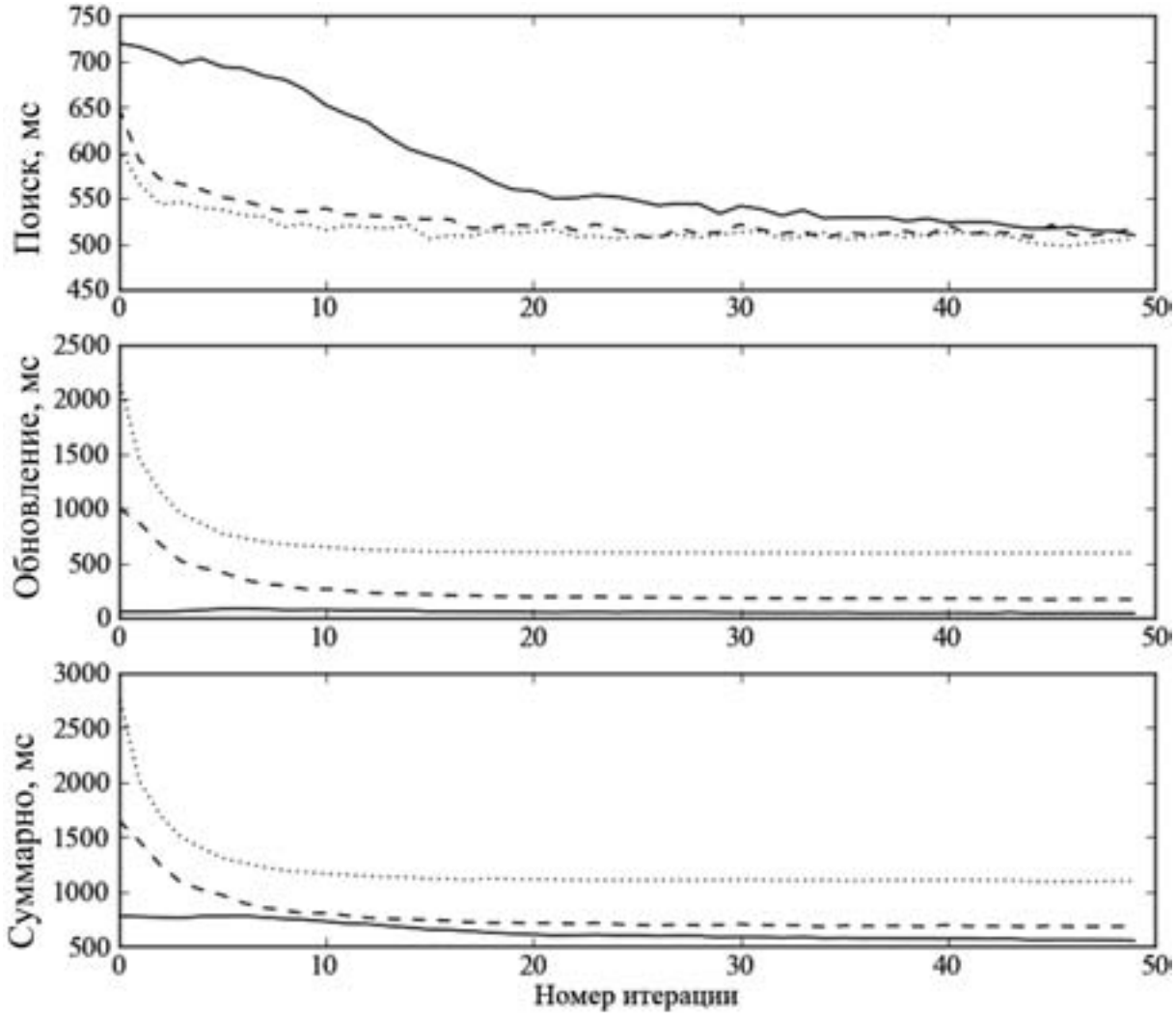

Рис. 2. Производительность контейнеров: пунктирная линия — R-дерево, штриховая — модификация первого подхода, сплошная — модификация второго подхода

Таблица 1

Числовые данные об изменении скорости поиска в дереве

Алгоритм	Время на первой итерации t_0, мс	Время на последней итерации t_1, мс	Улучшение времени r, %	Номер итерации N, когда было достигнуто значение 530 ± 3 мс
R-дерево	603	506	16	7
Подход 1	650	510	21	11
Подход 2	720	517	28	32

Таблица 2

Числовые данные об изменении скорости обновления в дереве

Алгоритм	Время на первой итерации t_0, мс	Время на последней итерации t_1, мс	Улучшение времени r, %
R-дерево	2205	595	73
Подход 1	1006	172	82
Подход 2	58	44	24

представленные ниже, выбраны как медианные для всего набора повторений.

На рис. 2 отображены зависимости времени поиска и обновления данных от номера итерации, а также суммы этих графиков.

Легко видеть, что при обновлениях R-дерево непрерывно улучшает свою структуру, что со временем обеспечивает лучшую скорость поиска и обновления данных. Данное утверждение справедливо для всех исследованных версий R-дерева. Из полученных данных также следует, что при описанных условиях время поиска для всех исследованных подходов будет слабо различаться после проведения значительного числа операций обновления данных. Результаты, свидетельствующие о повышении производительности поиска в дереве, показаны в табл. 1. Численные данные, подтверждающие увеличение производительности обновления, представлены в табл. 2.

Очевидно, второй подход дает большой прирост производительности операции обновления данных на каждой итерации, что выгодно отличает его от классического R-дерева. Первый подход также увеличивает производительность операции обновления, но его поведение во многом повторяет поведение классического дерева. Также он не дает равномерности времени операции обновления в течение первых итераций работы с деревом.

В целом при замере суммарного времени работы на каждой итерации второй подход дает ощутимый прирост производительности структуры данных, при этом обеспечивая лучшую устойчивость суммарного времени работы на первых итерациях. Это объясняется высокой скоростью обновления данных, что полностью оправдывает первоначально чуть более низкую скорость поиска, на практике обеспечивая лучшую суммарную производительность структуры.

На рис. 3 изображены зависимости времени поиска, обновления и суммарного времени работы структуры от $M[s] \in [0, 100]$ и $\Delta_{MBR} \in [0, 100]$. Из этих данных следует, что наибольшую производительность структура данных обеспечивает при значениях $\Delta_{MBR} \in [50, 70]$ и $M[s] \in [20, 40]$, т. е. при отношении $0{,}3 \leqslant \frac{M[s]}{\Delta_{MBR}} \leqslant 0{,}75$.

Отдельного пояснения требуют значения производительности поиска, наблюдаемые при скорости изменения объектов $M[s] \in [0, 10]$. Эти значения увеличивают вероятность того, что при нескольких итерациях обновления подряд объекты будут находиться внутри одного и того же узла, что препятствует повторной вставке объекта в дерево. Как следствие, это приводит к более медленному улучшению структуры дерева в сравнении со случаем больших значений параметра $M[s]$.

Заключение

В настоящей статье исследована производительность программного обеспечения, реализующего классические алгоритмы обновления R-дерева. Предложены новые подходы, позволяющие повысить скорость работы на определенном классе данных. Как видно из результатов, применение подхода 2 дает хорошую производительность на описываемой модели данных, что позволяет использовать его в прикладных задачах.

Описанная методология может найти применение при разработке программного обеспечения мобильной связи, геоинформационных систем, систем мониторинга движения и в других сервисах, обрабатывающих данные о местоположении объектов.

Предметом данной статьи не является исследование производительности предложенных решений на других моделях данных, этот вопрос может представлять интерес для дальнейших исследований. Аналогично применимость описанных подходов к более сложным модификациям R-дерева, таким как R*-дерево [5] и TPR*-дерево [13], а также вопросы быстродействия при работе с ними имеет смысл исследовать отдельно.

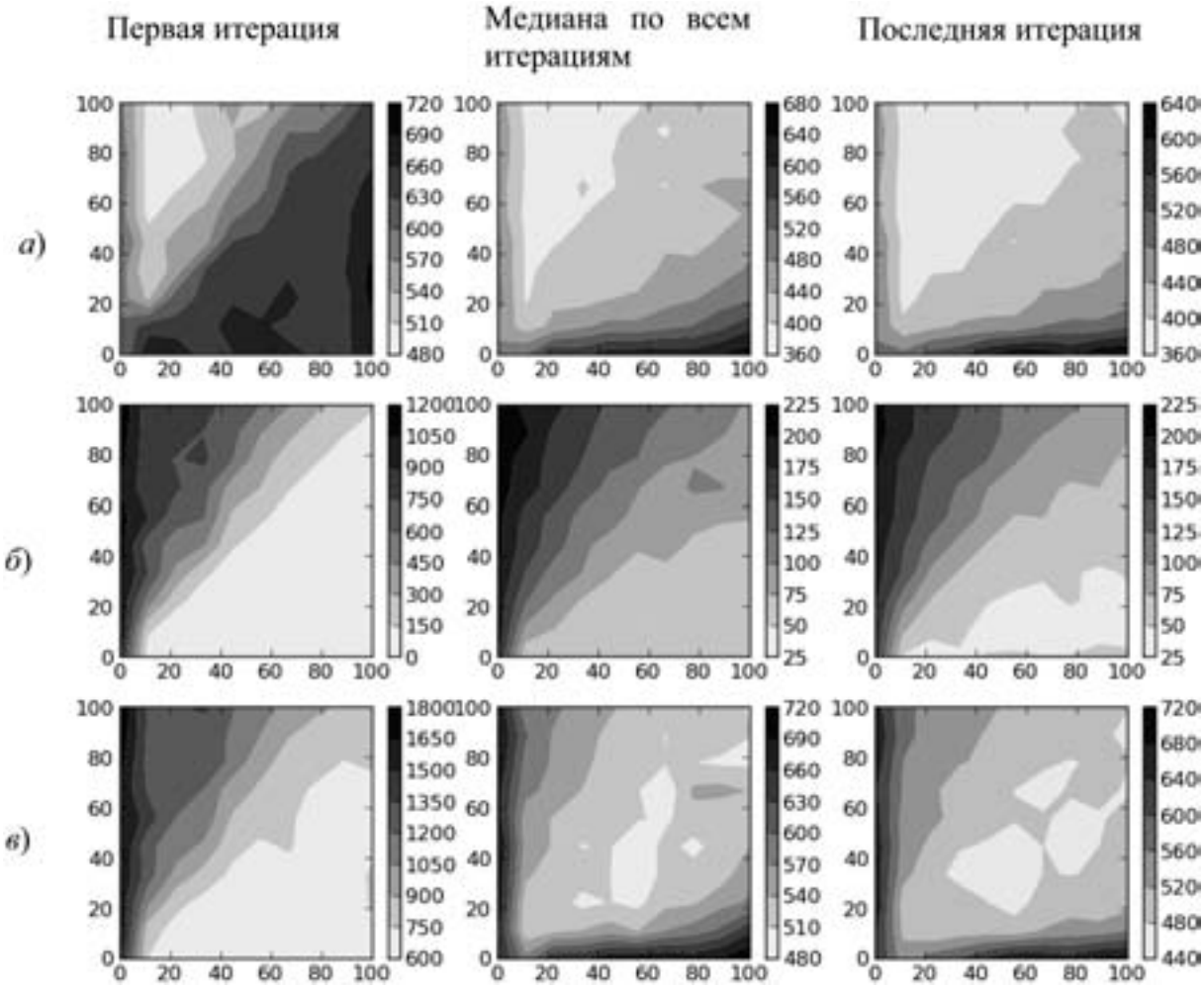

Рис. 3. Производительность операций поиска и обновления в зависимости от скорости объектов и расширения листьев дерева: *а* — обновление, *б* — поиск, *в* — суммарное время. По оси абсцисс — Δ_{MBR}, по оси ординат — $M[s]$

Список литературы

1. **Bentley R. A., Finkel J. L.** Quad-Trees: A Data Structure for Retrieval on Composite Keys // ACTA Informatica. 1974. Vol. 3, N 1. P. 1—9.

2. **Guttman A.** R-trees: a Dynamic Index Structure for Spatial Searching // Proc. of ACM SIGMOD Conf. on Management of Data. Boston. 1984. P. 44—57.

3. **Manolopoulos Y., Nanopoulos A., Papadopoulos A. N., Theodoridis Y.** R-Trees: Theory and Applications. London. Springer-Verlag, 2006.

4. **Sellis T., Roussopoulos N., Faloutsos C.** The R^+-tree — a Dynamic Index for Multidimensional Objects // Proc. of 13th International Conf. on Very Large Data Bases (VLDB'87). Brighton. 1987. — P. 507—518.

5. **Beckmann N., Kriegel H. P., Schneider R., Seeger B.** The R*-tree: an Efficient and Robust Method for Points and Rectangles // Proc. of ACM SIGMOD Conf. on Management of Data. AtlanticCity. 1990. P. 322—331.

6. **Brakatsoulas S., Pfoser D., Theodoridis Y.** Revisiting R-tree Construction Principles //Proc. of 6th East European Conf. on Advances in Databases and Information Systems. Bratislava. 2002. P. 149—162.

7. **Sistla A. P., Wolfson O. O., Chamberlain S., Dao S.** Modeling and Querying Moving Objects // Proc. of 13th IEEE International Conf. on Data Engineering (ICDE'97). Birmingham. 1997. P. 422—432.

8. **Wolfson O., Xu B., Chamberlain S., Jiang L.** Moving Objects Databases: Issues and Solutions // Proc. of 10th International Conf. on Scientific and Statistical Database Management (SSDBM'98). Capri. 1998. P. 111—122.

9. **Ang C. H., Tan T. C.** New Linear Node Splitting Algorithm for R-trees // Proc. of 5th International Symposium on Spatial Databases (SSD'97). Berlin. 1997. P. 339—349.

10. **Garcia Y., Lopez M., Leutenegger S.** On Optimal Node Splitting for R-trees // Proc. of 24th International Conf. on Very Large Data Bases (VLDB'98). — New York. 1998. P. 334—344.

11. **Kamel I., Faloutsos C.** Hilbert R-tree — an Improved R-tree Using Fractals // Proc. of 20th International Conf. on Very Large Data Bases (VLDB'94). Santiago. 1994. P. 500—509.

12. **Park M., Lee S.** Optimizing Both Cache and Disk Performance of R-trees // Proc. of 14th International Workshop on Database and Expert Systems Applications. Prague. 2003. P. 139—147.

13. **Tao Y., Papadias D., Sun J.** The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries // Proc. of 28th International Conf. on Very Large Data Bases (VLDB'02). Berlin. 2003. P. 790—801.

CONTENTS

ООО "Издательство "Новые технологии". 107076, Москва, Стромынский пер., 4

Дизайнер *Т.Н. Погорелова*. Технический редактор *Е.М. Патрушева*. Корректор *Е.В. Комиссарова*

Сдано в набор 12.05.2011 г. Подписано в печать 27.06.2011 г. Формат 60×88 1/8. Бумага офсетная. Печать офсетная. Усл. печ. л. 5,88. Уч.-изд. л. 6,94. Цена свободная.

Отпечатано в ООО "Белый ветер", 115407, г. Москва, Нагатинская наб., д. 54, пом. 4