ИЗДАЕТСЯ КОМПАНИЕЙ «ИНФОСИСТЕМЫ ДЖЕТ»

Jet Info

№ 7 (275)/2016

НАША «КУХНЯ» ЗАКАЗНОЙ РАЗРАБОТКИ

WWW.JETINFO.RU

Jet Info

Издается с 1995 года

Редакция:
Акимова Н.А., Дискина А.Л., Дмитриев В.Ю., Шедова Е.Л., Шубина В.А.

Дизайн и верстка:
Саблина М.А.

Корректура:
Макеева Е.И.

Иллюстрации:
www.shutterstock.com

Над номером работали:
Каткевич А.Г., Кичасов Р.Д., Кузнецова Н.Н., Нурутдинов А.А., Окунцев С.С., Романченко П.С., Тоботрас Б.В.

Издатель:
Компания «Инфосистемы Джет»

Контакты:
тел.: (495) 411-76-01
info@jet.msk.su,

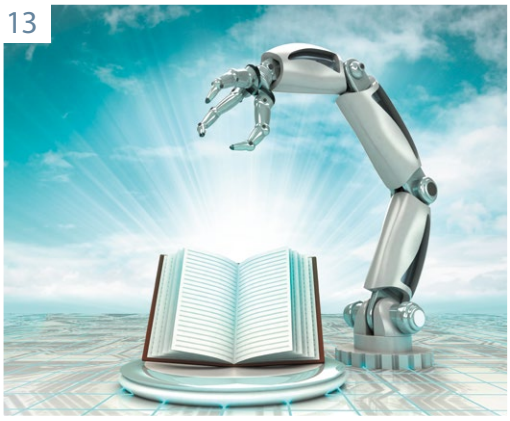
www.jetinfo.ru

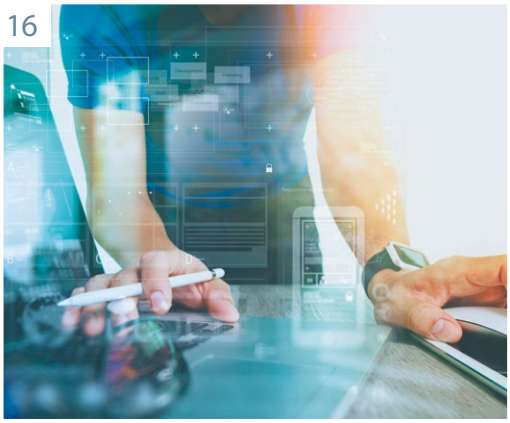
БОРИС ТОБОТРАС,
руководитель Центра
программных решений
компании «Инфосистемы Джет»

Вы открыли номер Jet Info, в котором мы делимся своим опытом по заказной разработке программного обеспечения. Мы занимаемся этим уже около 20 лет; за это время много раз менялись подходы и способы ведения проектов. Но все эти годы наши проекты по разработке объединяет одна основополагающая черта – выработанные нами методы решения самых разных задач, касающихся бизнес-анализа, собственно разработки, тестирования и управления такими проектами.

Этот номер даст вам возможность заглянуть на нашу «кухню» и увидеть самые интересные стороны этой работы. Интересного чтения!

Содержание





3 **От редакции**

6 **Новости**

7 **Тема номера**
Что может дать внешняя разработка
БОРИС ТОБОТРАС

10 **Тема номера**
«Фабрика разработки», или Как выглядит наш процесс разработки изнутри
ПАВЕЛ РОМАНЧЕНКО

13 **Тема номера**
Машинное обучение: как перейти от теории к практике
АЛЕКСЕЙ КАТКЕВИЧ

16 **Тема номера**
О технологическом цикле разработки кода на проекте в НСПК
СЕРГЕЙ ОКУНЦЕВ

Содержание

25

20 **Тема номера**
Искусство управления высокотехнологичными проектами
НАТАЛЬЯ КУЗНЕЦОВА

30

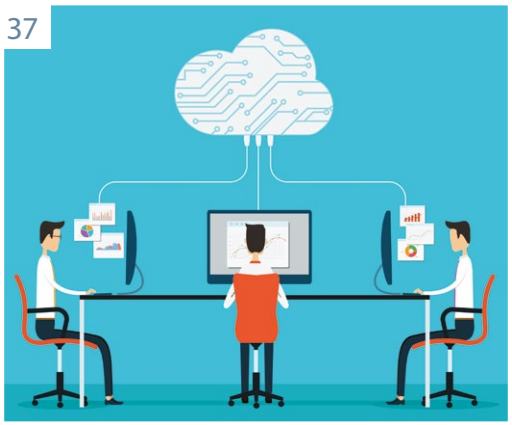
25 **Тема номера**
Создание модульных приложений на Java
РОМАН КИЧАСОВ

30 **Тема номера**
Разработка CPA-платформы для телеком-оператора
РОМАН КИЧАСОВ

34

34 **Тема номера**
Разработка ситуационных центров – взгляд изнутри
АЛЬБЕРТ НУРУТДИНОВ

37 **Тема номера**
Облачный подход к разработке
БОРИС ТОБОТРАС



37

СОЗДАНА СИСТЕМА ПРОТИВОДЕЙСТВИЯ МОШЕННИЧЕСТВУ В X5 RETAIL GROUP

Компания «Инфосистемы Джет» внедрила систему противодействия мошенничеству и гарантирования доходов в X5 Retail Group. Антифрод-система выявляет признаки реальных и потенциальных мошеннических схем, оперативно информирует о них специалистов по безопасности, а также позволяет формировать бизнес-ориентированную отчетность для операционных подразделений. Сегодня под антифрод-контролем находятся почти 8000 торговых точек ритейлера. По данным, полученным на фокусной группе из 20 магазинов, ритейлеру удалось снизить потери более чем на 1% от розничного товарооборота. Суммарные потери снизились на 0,76%, недостача уменьшилась на 0,36%.

Разработано 25 комплексных контрольных процедур, охватывающих кассовые операции и операции по картам лояльности, логистику товаров внутри магазинов (излишки, списания и пр.), товародвижение (объемы заказов, товарные запасы, прогнозы списаний и т.д.) – всего около 80% бизнес-процессов, существующих в торговых сетях.

Обследовав инфраструктуру заказчика и существующие бизнес-процессы, специалисты компании «Инфосистемы Джет» определили набор необходимых источников информации для антифрод-системы и состав загружаемых данных, инсталлировали и интегрировали ее с системами-источниками. К решению подключены центральные серверы баз данных по кассовым операциям торговой сети, процессинговый центр программы лояльности, модули ERP- и HR-системы, настроены механизмы автоматической загрузки данных из них. При этом сведения, необходимые для оперативного анализа ситуации, поступают в антифрод-систему в течение нескольких минут.

На основе собственной экспертизы в сфере борьбы с мошенничеством и с учетом опыта аналитиков X5 Retail Group специалисты компании «Инфосистемы Джет» разработали модели (комплексы алгоритмов) выявления неправомерных действий сотрудников. Эти алгоритмы позволяют автоматически определять подозрительные схемы, а также ошибки в процессах кассового обслуживания покупателей, выполнения транзакций по программам лояльности, управления товарной логистикой. Настроены механизмы формирования инцидентов и их назначения на проверку уполномоченным сотрудникам, а также контроля хода проверки.

Для сотрудников служб безопасности и руководителей операционных подразделений разработаны формы отчетов, содержащих статистику неправомерных действий и результаты расследования инцидентов, настроены средства визуализации. Формируемые отчеты позволяют бизнес-руководителям принимать обоснованные решения о необходимых мерах, в том числе по корректировке существующих процессов. Теперь ритейлер может держать под контролем различные категории потерь, наступивших из-за мошенничества или из-за нарушения установленных бизнес-процессов, по всей сети магазинов.

Антифрод-система построена на базе программного комплекса RAID компании WeDo Technologies. Сегодня это самое масштабное внедрение антифрод-решения в российском ритейле.

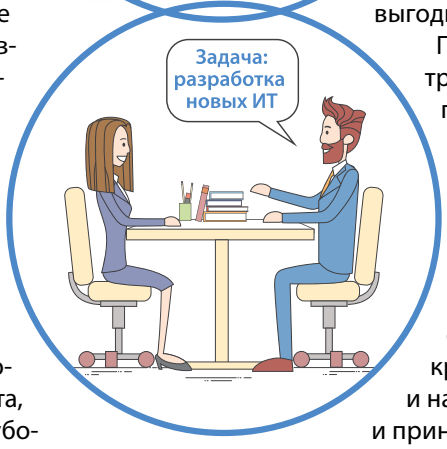
Специалисты системного интегратора осуществляют мониторинг работы системы, при необходимости выполняют доработку контрольных процедур, тонкую настройку параметров для снижения уровня ложных срабатываний и повышения эффективности процессов выявления мошенничества.

Запас производительности антифрод-системы позволяет заказчику расширять спектр контролируемых процессов в удобные для него сроки. В дальнейших планах ритейлера – увеличить полноту фрод-контроля за счет интеграции с системами Cash Control и видеонаблюдения, а также внедрить механизмы контроля мошеннических действий в системе закупок.

ЧТО МОЖЕТ ДАТЬ ВНЕШНЯЯ РАЗРАБОТКА

БОРИС ТОБОТРАС,
руководитель Центра
программных решений
компании «Инфосистемы Джет»

Сегодняшнее состояние дел вокруг заказной разработки отличают, пожалуй, 2 особенности. Первая – зрелость ИТ-индустрии, одним из свидетельств которой является большое количество готовых продуктов практически для всех задач, с которыми приходится сталкиваться типичному пользователю. В то же время наш основной корпоративный заказчик – пользователь нетипичный: его бизнес ставит перед ИТ задачи, обычно не решаемые в лоб установкой коробочного продукта, а требующие глубокой кастомизации или полноценной разработки новых информационных систем. Иногда необходимость замещения существующих продуктов диктуется внешними факторами, например, сегодня части наших заказчиков необходимо менять западные решения на отечественную разработку. В любом случае мы видим здесь вторую особенность: все более усиливается тенденция к осуществлению разработки своими силами, поддержке и развитию информационных систем с помощью собственного ИТ-подразделения – компании сосредотачивают в нем свою разработку ПО.



Какой мы видим свою роль как внешнего разработчика в таких условиях и какую ценность видят в нас наши заказчики даже при наличии внутренней разработки? Конечно, собственный департамент разработки позволяет решать большую часть ежедневных задач без потерь на подключение внешнего исполнителя – как финансовых, так и коммуникационных. Но есть целый ряд случаев, когда обращение к специализированной компании-разработчику несет зримые выгоды.

Первая и самая тривиальная выгода – решение одноразовых задач. Если компании необходимо разработать новую систему или доработать существующую, крайне неудобно и накладно искать и принимать на работу разработчиков, тестировщиков, аналитиков, выделять оборудование и покупать лицензии на средства разработки. Это тем более неудобно, что по завершении проекта придется так же резко сворачивать эту активность, переводя новую систему в режим поддержки. Здесь подключение внешнего исполнителя органично решает все проблемы.

Понятно, что ведение внутренней разработки позволяет собрать команду специалистов, глубоко понимающих собственные приложения и информационные системы и отлично ориентирующихся в используемых в них технологиях. Однако когда бизнес-требования диктуют необходимость разработки новых компонент, систем и комплексов, в некоторых случаях нужна экспертиза, далеко выходящая за рамки ежедневной поддержки существующих систем. И здесь проявляется наше второе отличие: мы обладаем широчайшим опытом разработки самых разных систем в различных отраслях – от банков и органов государственной власти до машиностроения и медицины. Центр программных решений выполняет десятки программных проектов каждый год, и этот опыт позволяет оперативно и качественно подходить к решению самых нестандартных задач заказчиков. Взаимодействие с различными вендорами, технологические стеки, требования и стили ведения проектов (то, что может представлять серьезные трудности для вну-

тренней команды разработки) – все эти задачи ежедневно решают наши сотрудники.

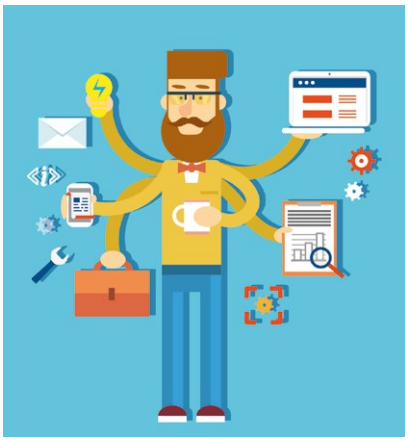
Чтобы представить еще один аспект, касающийся ценности внешнего разработчика, необходимо сделать философское отступление. Очень часто инженеры вообще и программисты в частности относятся к одному из двух типов характеров. Одни легко разбираются в новой тематике, энергично строят новые системы, но теряют к ним интерес, как только решены основные дизайнерско-конструкторские задачи. Другие не так любят изобретать новые подходы и архитектурные решения, зато эффективно и профессионально выполняют задачи по разработке и любят доводить работу до логического завершения. Для успеха любого проекта нужны люди обоих типов. Отметим также, что зачастую сразу после выхода системы в продуктив инженеры первого типа начинают присматриваться к другим проектам и в конце концов оставляют работу по этой системе или даже меняют место работы. В результате, когда нужно вернуться к разработке (как правило, получив новые требования от бизнеса), внутренней команде трудно найти нужных инженеров.

Мы одновременно ведем множество проектов для наших

> Все более усиливается тенденция к осуществлению разработки своими силами, поддержке и развитию информационных систем с помощью собственного ИТ-подразделения – компании сосредотачивают в нем свою разработку ПО.

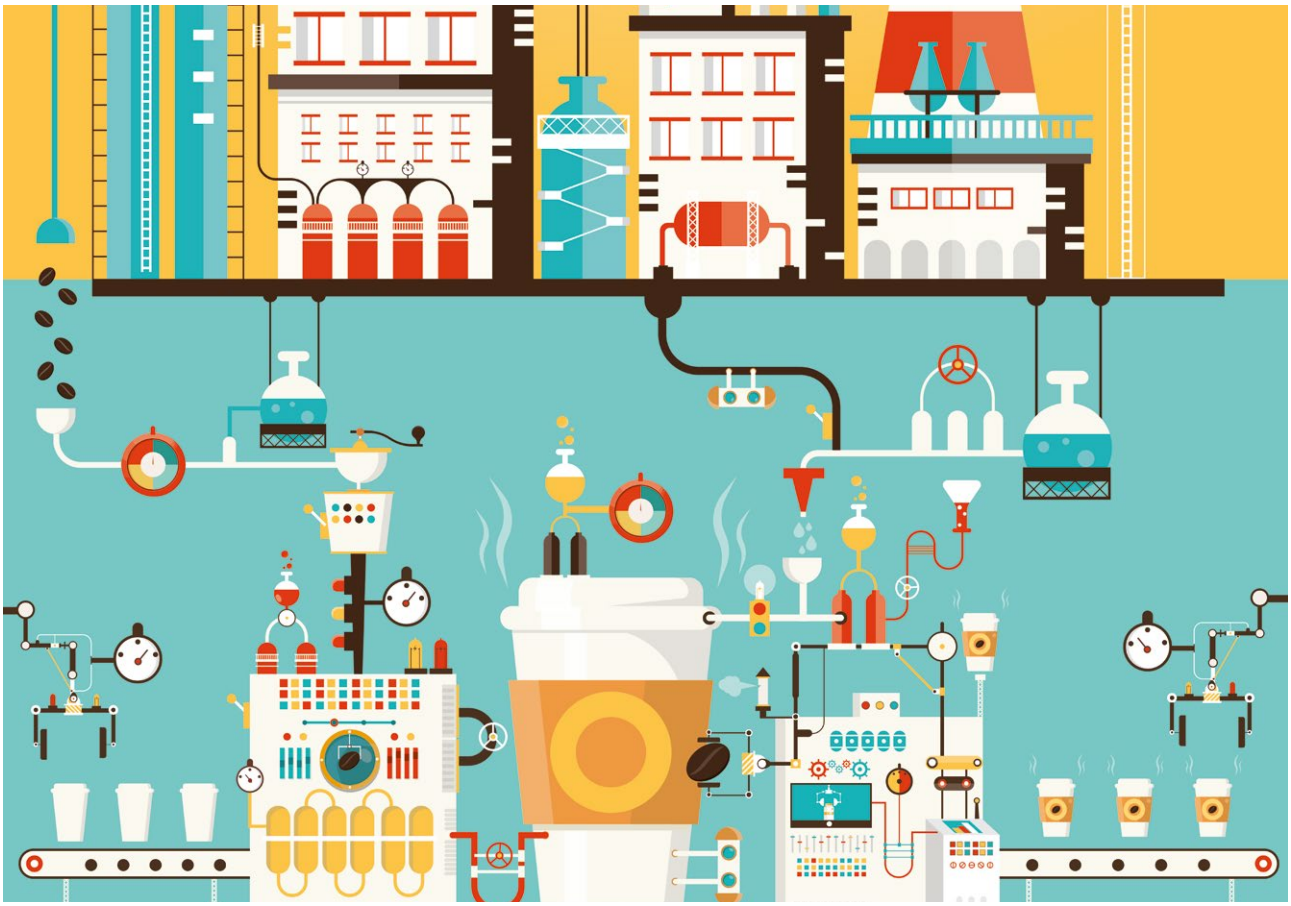
заказчиков. На их разных этапах нужны разнообразные специалисты, причем в различных количествах. Большой пул параллельных проектов позволяет нам гибко менять работу для наших специалистов, обеспечивая их возможностью постоянного профессионального роста. Кроме того, они избегают застоя в работе. Что это дает нашему заказчику? Возможность всегда обратиться к нам и реализовать проект, обеспечив себя командой, нужной для успеха всего дела.

Еще одна наша особенность – широкая специализация. В качестве примера можно привести нагрузочное тестирование. Одна из его задач – провести сайзинг оборудования, например, при смене поколений серверных систем или систем хранения данных под определенную ИС. Эта работа требует знания определенных методик и подходов к организации нагрузочного тестирования, умения строить тестовые модели данных, владеть инструментами нагрузки, способностью интерпретировать результаты измерений и не в последнюю очередь свободного «железа» для проведения тестирования. Конечно, все это любая крупная компания обычно способна организовать у себя самостоятельно. Но смена поколений оборудования обычно происходит раз в 3–6 лет, понятно, что конструкция, в которой необходимые для этого люди и «железо» будут востребованы так редко (а в остальное время будут простаивать), нежизнеспособна. Поэтому к нам часто обращаются именно с такими задачами: не типовыми, требующими узкой специализации, невозможной в условиях внутренней разработки.



Вследствие разнообразия наших проектов нарабатывается опыт использования самых разных технологий и продуктов различных вендоров. Нужно выбрать между RDBMS и NoSQL? Нужно решить, какую шину использовать – от IBM, Oracle или Red Hat? Нужно выбрать, строить: приложение на 3-звенной архитектуре с сервером приложений или на распределенной микросервисной архитектуре? Мы разберемся в стоящих перед заказчиком задачах и поможем сделать правильный выбор. JI

«ФАБРИКА РАЗРАБОТКИ», ИЛИ КАК ВЫГЛЯДИТ НАШ ПРОЦЕСС РАЗРАБОТКИ ИЗНУТРИ



ПАВЕЛ РОМАНЧЕНКО,
начальник отдела разработки
Центра программных решений
компании «Инфосистемы Джет»

Если рассматривать разработку как услугу, в ней можно выделить 2 основных направления – продуктовую разработку, т.е. создание продуктов «на рынок», и заказную, т.е. разработку уникального ПО под нужды заказчика. Мы поговорим о том, как у нас осуществляется процесс заказной разработки.

Опыт показывает, что обычно наименее подвержены изменениям сроки разработки, а наиболее гибкими частями являются предметная область и требования. Эти обстоятельства и диктуют правила, по которым ведется разработка. В ее ходе мы решаем несколько задач. Первая – это выявление требований заказчика, их структурирование и последующее описание предметной области, зачастую новой для нас. Вторая задача – это собственно разработка, которую можно разделить на проектирование и непосредственно программирование. При этом разрабатываемую систему необходимо интегрировать со смежными системами, разработку нужно вести с учетом языков и фреймворков, принятых у заказчика, придерживаться принятых им методик интеграции. И третья – обеспечение качества и тестирование. Это самая наглядная, наиболее видимая заказчику часть, так как именно команда тестирования дает добро на выдачу системы, обеспечивает качественное покрытие тестами, выявляет альтернативные сценарии, возможно, ускользнувшие от разработчиков и аналитиков. Таким образом, исходя из задач проектная команда обычно состоит из аналитиков, разработчиков и тестировщиков. Есть также группа управления, которая включает менеджера проекта и архитектора.

ТРИ КИТА И ОДНА ЧЕРЕПАХА

Разработка зиждется на трех китах: управление требованиями, разработкой и тестированием. Все это стоит на большой черепахе – на планировании.

Стоит сказать, что жизненный цикл проекта начинается задолго до старта программирования: еще на этапе пресейла мы выполняем предварительную аналитику, фиксируем основные задачи, которые необходимо решить.

В начале проекта аналитики погружаются в предметную область и описывают задачу с помощью так называемых «постановок» – описаний того, что нужно сделать. Наш основной инструмент на этом этапе – Atlassian Confluence. Это удобное средство для управления большим объемом информации. Говоря по-простому, Confluence позволяет создавать страницы с отформатированным текстом и совместно его редактировать. Для каждого документа ведется история изменений, для обсуждения предусмотрена система комментариев. Страницы можно иерархически организовывать, группируя, например, по функциональным областям проекта, по вехам и т.д.

Чуть позже стартует проектирование, архитектор (он же главный конструктор) определяет основные технические решения, выбирает платформу, фреймворки, библиотеки и составляет спецификации. Основными инструментами для него являются все тот же Atlassian Confluence и Visual Paradigm. В Confluence архитектор описывает принятые решения и проставляет прямые и обратные ссылки на постановки аналитиков, таким образом, всегда можно понять, как главный конструктор собирается реализовывать то или иное требование и какие требования затрагивает определенное

> Тесная коммуникация внутри проектной команды является одним из важнейших условий создания качественного продукта. Поэтому на каждый проект создается чат в Skype, почтовый список рассылки: мы обсуждаем новые требования заказчика, архитектурные решения и т.д.

проектное решение. Для проверки своих решений архитектор проводит Design Review: встреча объединяет участников его проектной команды, а также архитекторов и ведущих программистов других проектов, реализуемых у заказчика. Представление решения участникам имеет целью выявить неучтенные факторы, неоптимальные подходы и т.д.

После появления первых проектных решений мы приступаем к самой разработке. Проектные решения детализируются на спецификации, заводятся задачи в Atlassian Jira. Ссылки на задачи по разработке мы проставляем в проектные документы в Atlassian Confluence. Благодаря тому, что Confluence и Jira интегрированы, при открытии проектного решения мы всегда видим, в каком состоянии находятся задачи. Также стоит сказать, что для управления исходным кодом используется Git, в качестве централизованного сервера репозиториев – GitLab.

Каждую задачу мы разрабатываем в отдельной ветке Git, названной по имени задачи из Jira, что позволяет легко соотносить изменения с требованиями. Комментарии в commit также содержат номер задачи, чтобы коммиты отображались в Jira Worklog. Выполненные задачи переносятся (мержатся) в ветку development, а оттуда – в момент релиза – в ветку master (см. рис. 1).

Немаловажным фактором в достижении качественного результата разработки является тестирование, особенно автоматическое. Автоматические тесты у нас разрабатывают как программисты (unit-тесты), так и тестировщики (с использованием, например, Cucumber), в качестве сервера интеграции и непрерывной сборки мы используем Jenkins.

После сборки и тестирования артефакта он выкладывается в репозиторий Artifactory, откуда его забирает команда тестировщиков для проведения функциональных тестов.

Протестированный релиз, не содержащий критических и важных дефектов, выдается заказчику. Обязательный документ, сопровождающий каждый релиз, – это Release Notes. В нем мы указываем исправленные дефекты, найденные в предыдущих релизах, а также новые возможности системы.

Один из основных критериев успеха проекта – поставка в срок, поэтому мы планируем дату завершения фазы активной разработки на неделю-две (в зависимости от сложности проекта) раньше даты поставки и в это «промежуточное» время проводим стабилизацию – несколько раундов тестирования и исправления выявленных дефектов.

ВОПРОСЫ КОММУНИКАЦИИ

Стройный порядок вещей – аналитика, проектирование, разработка, тестирование, поставка – иногда может нарушаться по разным причинам. Самые тяжелые по последствиям случаи – это ошибки в аналитике со стороны как исполнителя, так и заказчика. Они тянут за собой перепроектирование, новую разработку, новые тест-планы, тестирование. Поэтому очень важны как можно более полное погружение аналитиков в предметную область, тщательное ведение реестра требований, отслеживание их изменений и отражение этих изменений в архитектуре приложения. Лет 7 назад на одном из проектов наш аналитик провел очень большую работу, предоставил user stories, описал предполагаемое поведение пользователей при работе с разрабатываемым решением, но при этом неверно проработал внутренние процессы (т.е. процессы, происходящие в системе, скрытые от конечного пользователя). Ошибка была замечена при презентации будущего решения заказчику. Правило представлять алгоритм наших будущих действий и общее видение системы представителям заказчика до начала самой разработки позволило избежать существенных проблем на проекте. Мы заново провели аналитику, изменили архитектуру решения и разработали систему в срок.

Тесная коммуникация внутри проектной команды является одним из важнейших условий создания качественного продукта. Поэтому на каждый проект создается чат в Skype, почтовый список рассылки: мы обсуждаем новые требования заказчика, архитектурные решения и т.д. Кроме того, проводятся ежедневные или еженедельные встречи – статусы, на которых команда определяет, что выполнено, ставит задачи на следующую неделю, корректирует планы. Но и в этих мероприятиях важно соблюдать баланс: например, в одном проекте менеджер устраивал ежедневные статусы по Skype на час-полтора, что в итоге стало тормозить рабочий процесс. По согласованию с заказчиком мы перешли на более эффективные встречи меньшими группами, общий статус собирался раз неделю.

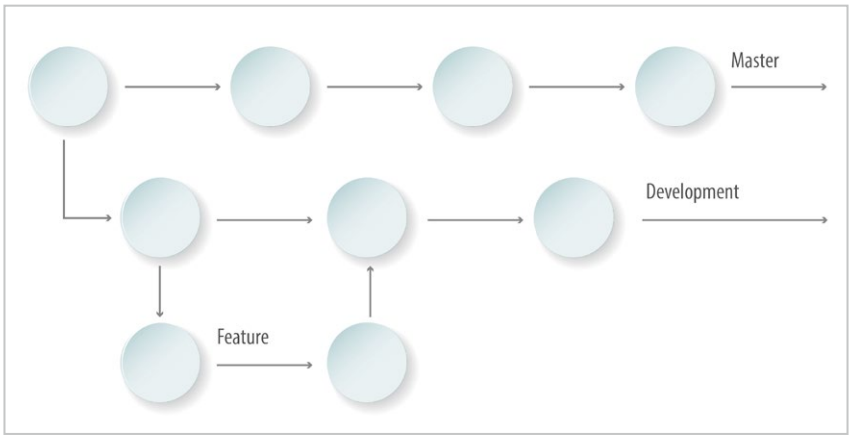
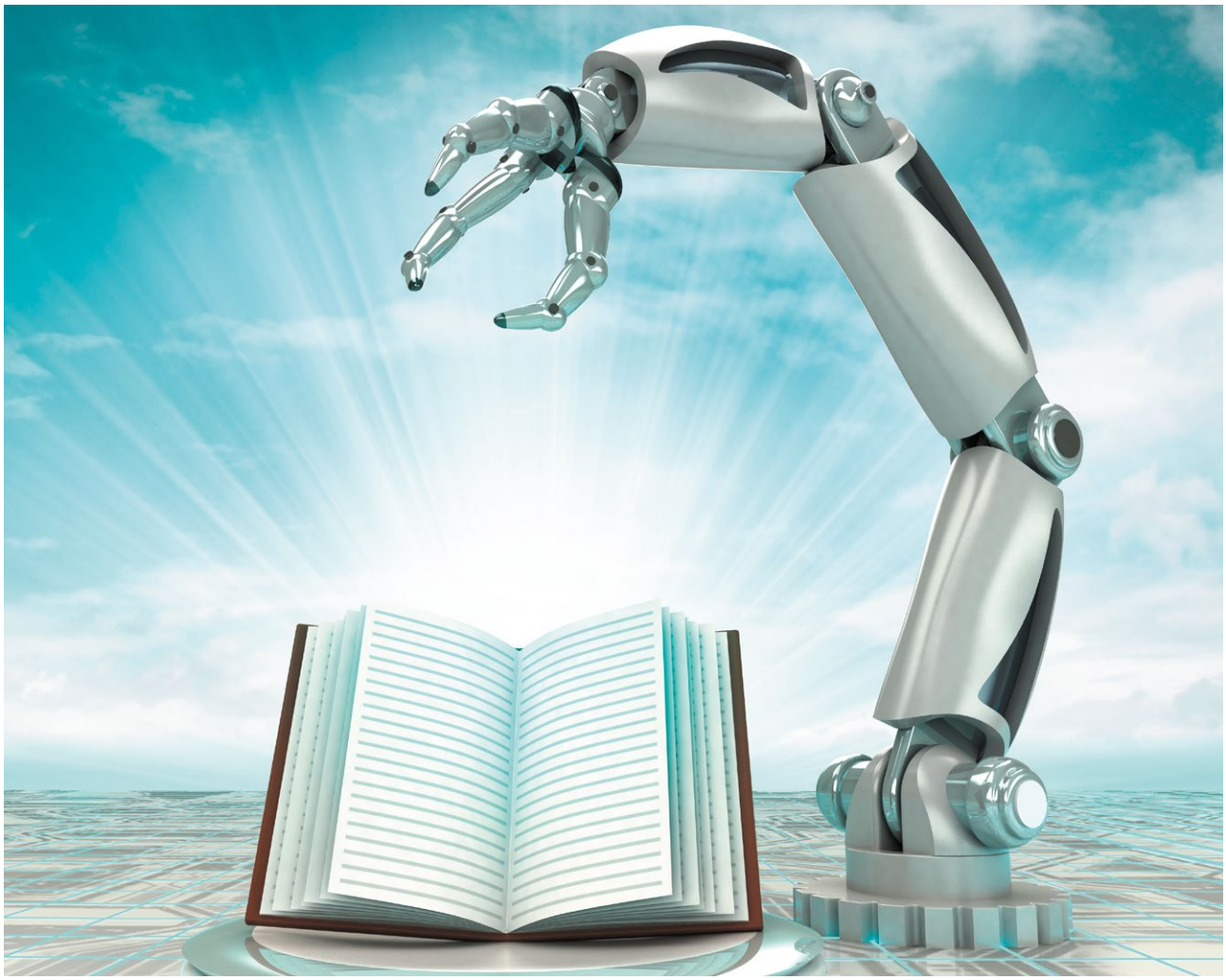


Рис.1. Модель ветвления Git

***

Мы используем описанные правила и подходы уже не первый год, и за это время они зарекомендовали себя в целом пуле наших проектов. В других статьях нашего номера мы подробно описываем примеры подобных кейсов.

МАШИННОЕ ОБУЧЕНИЕ: КАК ПЕРЕЙТИ ОТ ТЕОРИИ К ПРАКТИКЕ



АЛЕКСЕЙ КАТКЕВИЧ,
архитектор Отдела разработки
компании «Инфосистемы Джет»

Когда меня попросили написать статью для этого номера Jet Info о машинном обучении (МО), я, признаться, был озадачен. Писать о завершенных проектах на эту тему пока нечего – российский корпоративный заказчик только-только стал интересоваться возможностями использования алгоритмов машинного обучения в своем бизнесе. Говорить о текущих проектах можно далеко не всегда, да и сложно заранее предсказать, насколько успешными они окажутся в плане применения МО. Поэтому сейчас уместно говорить о том, что же такое машинное обучение и как его можно применять в суровом мире корпоративных систем.

Стоит сказать, что у ряда заказчиков, особенно если это люди от бизнеса, нет четкого понимания того, что такое МО, искусственный интеллект, самообучаемые системы. Иными словами, часто уже на стадии обсуждения вариантов решения их задач и подготовки ТКП возникают ситуации, когда заказчик, глядя на предлагаемое решение, интересуется: «А где здесь машинное обучение?». При этом нередко МО не нужно для решения его проблемы. Даже среди людей, которые имеют полное право именовать себя ИТ-шниками, далеко не все могут с уверенностью сказать, что такое машинное обучение, зачем оно нужно и как устроено. Все это сподвигло меня собрать и структурировать свои мысли в виде этой небольшой статьи.

ЧТО ЕСТЬ МАШИННОЕ ОБУЧЕНИЕ

Я специально не буду погружаться в описания различных алгоритмов, способов применения и уж тем более математики, лежащей под МО. Дело в том, что все это описано уже не по одному разу, причем специалистами, которые понимают в этом больше, чем я. Тем не менее мы не сможем обойтись без нескольких понятий.

Крайне простое, немного неточное, но при этом расставляющее все на свои места определение МО я прочел в свое время на классическом курсе от Andrew Ng на Couresra: «Машинное обучение – это область науки, описывающая способы обучения компьютера (создания алгоритма) решению какой-либо задачи без явного написания самого алгоритма».

В этом описании кроется одна немаловажная особенность: с одной стороны, есть задачи, для которых создать четкий формализованный алгоритм практически невозможно (например, распознавание голоса, компьютерное зрение, распознавание естественных текстов). С другой – нередко итоговый алгоритм, полученный в результате машинного обучения, достаточно прост, соответственно, затраты на его создание аналитическим путем могут быть в разы меньше, чем в случае применения МО. В качестве примера можно привести байку о том, как разработчик проходил собеседование и в ответ на просьбу написать алгоритм решения простой задачи FizzBuzz в шутку привел алгоритм создания, обучения и применения нейронной сети. К сожалению, пока еще встречаются прецеденты, когда в подобных случаях заказчик хочет применять «самообучающиеся» алгоритмы. В то же время существует широкий пласт задач, решить которые «классическим» способом если и возможно, то сопряжено с непозволительными трудозатратами.

Методы машинного обучения можно разделить на 2 большие группы. Первая – это обучение «с учителем», т.е. обучение алгоритма на данных с заранее известным ответом. Вторая – обучение «без учителя», когда все, что у нас есть, – это сырой набор данных. В каждой группе есть огромное количество подгрупп и методов, которые подходят для решения более узких задач, тем не менее их общая черта – это процесс применения. Любую модель, перед тем, как она сможет выдавать какой-либо результат на «боевых» данных, нужно обучить. И именно в процессе обучения кроется главная разница между двумя группами. Дадим «математическое» определение: при наличии «учителя» для обучения модели на входе есть множество данных *X*, множество правильных ответов *Y*, и все, что остается, выявить на основании этих данных функцию *F(X)*, которая вычисляет *Y* с достаточной точностью. После обучения у нас остается модель, содержащая *F(X)*. Ее можно применять к реальным данным для получения *Y*, когда он неизвестен.

При обучении «без учителя» на входе у нас имеется только массив данных *X*. И наша задача – найти такие *F(X)*, которые на выходе дают *Y*, имеющий какой-либо смысл. Естественно, смысл ищется исходя из заранее известных предположений. Собственно, поиск смысла в больших объемах неразмеченных данных – это отдельная область машинного обучения, которую раньше называли Data Mining, а теперь обычно именуют Data Science, хотя это и не совсем точно.

Если перенести эти определения на примеры из реальной жизни, можно сказать, что задачей для МО «с учителем» будет, например, прогнозирование цены квартиры в зависимости от ее расположения, метража, класса, ближайшего метро, района и т.п. на основании имеющихся сведений о ценах на рынке. Задача на первый взгляд не сложная, однако после того как количество оце-

ниваемых параметров перевалит за 10, решить ее аналитически станет гораздо сложнее. Классической задачей для обучения «без учителя» является сегментация и рекомендация товаров в онлайн-магазине. В этом случае у нас нет четких данных о том, что, например, покупатель 32 лет, мужского пола, с мобильного телефона на Android купит кофемолку. Однако мы можем по этим параметрам отнести его к определенной группе и в рекомендациях предложить товары, наиболее часто приобретаемые ее членами.

Возвращаясь к задачам, которые могут быть решены «классическим» способом: описанные выше примеры относятся как раз к таким. Иначе говоря можно сформировать огромный набор правил и фильтров, которые будут их решать, но чем больше параметров, от которых зависит решение, тем больше правил и тем сложнее они воспринимаются человеком. И самое главное: если что-то изменится в исходных данных, нам опять понадобится команда аналитиков и разработчиков, для того чтобы учесть эти изменения. В случае же с машинным обучением модель просто переобучится (в зависимости от выбранного подхода либо на лету, либо с помощью процесса обучения на новых данных).

ЗАЧЕМ ЭТОМУ УЧИТЬСЯ?

В России применение МО, особенно в корпоративном секторе, отстает от остального мира. Сложилась парадоксальная ситуация: в ТОП-100 лучших Data Scientists планеты входит достаточно много наших соотечественников, при этом компании, которые могут похвастаться реальными проектами, приносящими пользу в этой сфере, в России можно пересчитать по пальцам одной руки. Это говорит о том, что на рынке существует колоссальная нехватка специалистов, причем даже не гениальных ученых, а простых квалифицированных аналитиков, способных грамотно поставить задачу для машинного обучения, подготовить данные, осознано выбрать подходящую модель и проверить ее.

При этом сама идея применения методов машинного обучения все глубже проникает в мир корпоративных систем, порождая соответствующий поток задач. С учетом внушительных объемов информации, накопленных в системах, нужна не только аналитическая экспертиза для создания моделей, но и инженерные компетенции для построения эффективной и производительной архитектуры, способной их использовать. Все это стало предпосылкой для того, чтобы мы в Центре программных решений организовали свой центр компетенций, одной из областей знания которого должно стать машинное обучение. Мы планируем аккумулировать и делиться не только теоретическими знаниями, но и практическими аспектами использования инструментов типа Rapid Miner, IBM SPSS, Knime, Weka, SkyTree, технологий Big Data – Apache Spark MLib, Apache Ignite. Результатом работы центра компетенций должна стать разработка четких алгоритмов решения типичных задач компаний различных отраслей, связанных с Большими Данными и машинным обучением. Только подобные алгоритмы может позволят решать задачи заказчиков, «заточенные» под МО (таргетинг, выявление фрода и т.д.), быстро и качественно, а самое главное – эффективно. JI

✓ Как этому научиться

Прежде всего следует обратиться к дистанционному обучению типа Coursera. Собственно, на этой площадке располагается уже ставший классическим курс Стэндфордского университета от профессора Andrew Ng. Это отличная база для дальнейшего изучения, курс подходит даже тем, кто прогуливал в институте статистику и в принципе «не может в математику».

В случае если этого вам покажется мало и захочется углубить свои знания, существуют большое количество более длительных курсов. Из русскоязычных хотелось бы отметить лекции Школы анализа данных от Яндекса, которые с недавних пор можно свободно скачать с Яндекса, а также совместный курс МФТИ и Яндекса на все той же Coursera. Последний – это целая специализация с процессом обучения в течение полугода и дипломным проектом по окончании.

Если теоретическое обучение вы уже завершили или считаете, что вам не нужно глубокое понимание того, как работает математика «под капотом», можно попробовать учиться на очень хороших tutorials, которые лежат на сайте Kaggle.com. Это своеобразная социальная сеть для людей, объединенных любовью к машинному обучению и Data Science. Классический пример решаемых ими задач, – «Титаник», где на входных данных о реальных пассажирах печально известного корабля показаны все этапы применения машинного обучения: подготовка данных, определение параметров, обучение и оценка модели, ее применение.

О ТЕХНОЛОГИЧЕСКОМ ЦИКЛЕ РАЗРАБОТКИ КОДА НА ПРОЕКТЕ В НСПК

СЕРГЕЙ ОКУНЦЕВ,
руководитель отдела тестирования
Центра программных решений
компании «Инфосистемы Джет»

Год назад мы приняли участие в разработке платежной системы «Мир» – нам нужно было в кратчайшие сроки создать систему клиринга. К слову, аналогичные решения годами разрабатываются и тестируются у таких столпов рынка, как MasterCard и Visa. Перед нами стояла задача амбициозная и ответственная, требующая, помимо высококвалифицированных специалистов, эффективной организации процессов разработки и тестирования. Ниже описаны примененные на проекте практики, которые сократили его длительность и позволили к сроку создать качественный продукт. Сразу обозначим состав команды со стороны нашей компании, в нее входили 7 разработчиков, 4 инженера по функциональному тестированию, 3 инженера по автоматизации, 3 аналитика и 1 проектный менеджер.

Не секрет, что тестирование обычно отстает от разработки по времени: пока у тебя нет готового решения, ты не можешь полноценно начать тестирование. Что уж говорить об автоматизации этого процесса. Но если качественно проработаны требования к создаваемой системе и правильно спроектирована архитектура автотестов, тестирование может идти параллельно с разработкой или обгонять ее. Более того, когда появится готовое решение, мы в кратчайшие сроки сможем перепроверить, насколько оно соответствует требованиям. Когда тесты пишутся до разработки кода, то такой подход называется TDD – Test Driven Development (разработка через тестирование). Стоит сказать, что на проекте в НСПК мы немного отошли от канонов: TDD предполагает, что тесты пишет разработчик на уровне модулей, у нас этим занимался отдел автоматизации тестирования на уровне системы в целом. Для части функционала мы разработали тесты раньше, чем был готов код его реализующий. У TDD-подхода есть ряд преимуществ. Он позволяет находить дефекты на самых ранних этапах разработки. В то же время команда разработчиков заранее думает о том, чтобы решение было тестопригодно, соответственно, код становится более поддерживаемым.

Кроме того, тесты позволяют производить рефакторинг кода без риска его испортить: при внесении изменений в хорошо протестированный код риск появления новых ошибок значительно ниже. Если новая функциональность приводит к ошибкам, тесты сразу же это покажут.

Дополнительным плюсом является тот факт, что тесты могут использоваться в качестве документации (об этом мы скажем ниже). И наконец, несмотря на то что при разработке через тестирование требуется написать большее количество кода, общее время, затраченное на разработку, обычно оказывается меньше. Поскольку тесты защищают от ошибок, время, уходящее на отладку, многократно снижается. Рецепт быстрой разработки звучит так: большое количество тестов помогает уменьшить количество ошибок в коде.

При этом у TDD есть слабые места. Подход не позволяет автоматически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами. В силу того, что UI очень часто меняется и поведение поль-

> Непрерывная интеграция – это практика как можно более частой сборки всех модулей и изменений системы вместе. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. И она может задержать окончание работ. Переход к непрерывной интеграции снижает трудоемкость процесса и делает интеграцию более предсказуемой.

зователя формализовать сложнее, чем протокол обмена данными, в этой области TDD практически неприменим – временные затраты на реализацию возможности его использования превысят общее время, отведенное на разработку.

ИСПОЛЬЗУЕМЫЕ ИНСТРУМЕНТЫ

Для описания тестов мы использовали Cucumber – инфраструктуру тестирования, которая дает возможность устранить коммуникативный разрыв между членами команды (в нее могут входить тестировщики, аналитики, разработчики, представители бизнеса). Тесты разрабатываются в строго определенной нотации, о которой договариваются все участники, чтобы одинаково понимать процесс.

Поведением разработки при этом можно управлять в стиле Given/When/Then (условия/операция/результат). Контрольные тесты записываются в файлы функций, охватывающие один или несколько сценариев тестирования. Cucumber интерпретирует тесты на указанном языке программирования. В нашем случае тесты преобразуются в Java-код.

Конечно, это не что-то сверхновое, грамотные разработчики давно взяли на вооружение разделение кода в своих тестах в следующем виде и последовательности: подготовка к тесту, тестируемое действие, проверка результата выполнения этого действия. В нотации Given/When/Then эти этапы немного по-другому обозначены и подняты на уровень всей команды. Например, в Given описывается контекст выполнения сценария тестирования (как вариант, точки вызова сценария в приложении), а также содержатся любые необходимые для выполнения сценария данные. When говорит об операциях, результат или поведение которых мы тестируем. Then определяет ожидаемый результат. Например:

> **Feature:** Первый презентмент
>
> **Scenario Outline:** Проверка корректности первого презентмента
> **Given** Клиринговая система получает файл с первым презентментом
> **When** Клиринговая система обрабатывает файл с первым презентментом
> **Then** Файл «результат» должен появиться в выходной папке системы

Git (система хранения и управления версиями кода)

Вопрос организации хранения и доступа к коду в команде разработки уже давно не является сакральным знанием. Нет ни одного серьезного проекта, в котором бы весь код хранился локально у одного разработчика. Код должен быть доступен всем участникам проекта в любое время, независимо от других людей в команде. При этом было бы неплохо также иметь доступ к его ранним версиям. Более того, в больших командах есть практика веток, когда та или иная функциональность разрабатывается и тестируется в отдельной ветке, а затем сливается в основной ствол. В качестве системы хранения и управления версиями кода на проекте в НСПК мы использовали Git.

Code Review (инспекция кода)

Code Review (CR) – инженерная практика в терминах гибкой методологии разработки. Это анализ (инспекция) кода, позволяющая выявить ошибки, недочеты, расхождения в стиле написания кода, в соответствии написанного кода поставленной задаче. Команда разработки на проекте в НСПК у нас состояла из 7 разработчиков. Как только один разработчик заканчивал свой кусок функциональности, он отправлял запрос на изменение. Как только любой другой разработчик освобождался, он анализировал кусок кода коллеги на предмет единого оформления, правильной реализации логики и эффективности использования кода.

У такого подхода есть ряд преимуществ:

1. Выявление ошибок на ранних этапах, позволяет экономить время тестирования.
2. Распространение знаний. Во многих командах у каждого программиста есть кусок кода в проекте, за который он ответственен. Он и только он сфокусирован на этом коде. Пока коллеги не поломают что-то, связанное с этим кодом, они не обращают на него внимания. Побочный эффект от этого состоит в том, что только один человек полностью понимает, как работает определенный компонент. Если этот человек не укладывается в сроки или заболел, мало кто сможет оперативно помочь ему и взять часть работы на себя. В случае с CR, как минимум два человека будут знакомы с кодом, – автор и reviewer. Reviewer не знает код так, как его знает автор, но он знаком со структурой и архитектурой кода, что очень важно.
3. Единый стиль написания кода у всей команды.

> Подход TDD не позволяет автоматически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами. Так как UI часто меняется и поведение пользователя формализовать сложнее, чем протокол обмена данными, в этой области TDD практически неприменим.

При этом Code Review требует наличия у разработчиков соответствующего опыта, поэтому на первых этапах работы процесс может идти медленно.

Unit Tests

Разработчики не защищены от влияния человеческого фактора, поэтому следующим этапом после инспекции кода должно стать проведение модульных тестов, которые будут проверять разработанный код на соблюдение контрактов с другими модулями.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной или критичной для бизнеса функциональности. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к деградации системы, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Unit Tests повышают надежность кода и позволяют обнаруживать ошибки на ранних этапах разработки. При этом они не позволяют отловить все ошибки. К тому же Unit Tests, как и обычный код, нужно поддерживать.

Continuous Integration (CI, непрерывная сборка)

Непрерывная интеграция – это практика как можно более частой сборки всех модулей и изменений системы вместе. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. И она может задержать окончание работ. Переход к непрерывной интеграции снижает трудоемкость процесса и делает интеграцию более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий.

Преимущества:

1. Быстрое выявление и исправление проблем интеграции.
2. Немедленный прогон модульных тестов (Unit Tests).
3. Постоянное развертывание целевого решения на тестовом стенде или на стенде разработчиков, что позволяет не тратить время участников проектной команды.
4. Прогон интеграционных автотестов.

Из минусов стоит назвать относительно большие временные затраты на поддержку непрерывной интеграции и потенциальную необходимость в выделенном сервере под эти нужды.

У нас на проекте в НСПК сборка модулей происходила минимум один раз в день. Тесты в горячую пору запускались до 5–9 раз в день.

Нужно понимать, что вышеприведенные практики очень плохо работают поодиночке. Только в симбиозе они позволяют получить сильный прирост к эффективности и качеству кода на проекте. Последовательность всех указанных практик представлена на рис. 1.

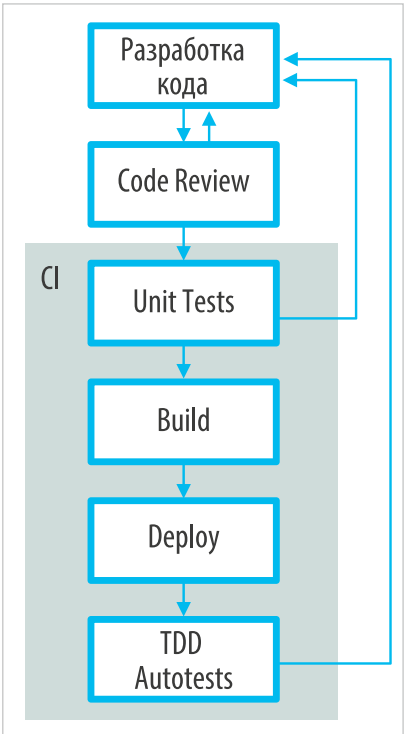


Рис. 1. Последовательность практик тестирования кода

И напоследок приведем цифры и результаты, полученные нами на проекте. Мы поставили все сборки в срок, при этом сократили Time-To-Market в 15 раз: без автоматизированных тестов, включенных в цепочку CI, проведение полного цикла регресса заняло бы 30 дней, а мы в общей сложности уложились в 2 человеко-дня. Проект содержит около 1 млн строк, 70% функционала было покрыто автотестами. До передачи в продуктив (за полгода) мы выявили более 1000 ошибок. В продуктиве же было обнаружено около 50 важных ошибок. JI

ИСКУССТВО УПРАВЛЕНИЯ ВЫСОКОТЕХНОЛОГИЧ-НЫМИ ПРОЕКТАМИ

НАТАЛЬЯ КУЗНЕЦОВА,
старший менеджер проектов
Центра программных решений
компании «Инфосистемы Джет»

В этой статье мы отойдем от общепринятого подхода, который рассматривает управление проектами в разрезе групп процессов, и поговорим об этом с точки зрения управления целями проекта, проектной командой и выбора технологий.

ЦЕЛИ

Сначала озвучим критерии успешного проекта:

- ✓ работы должны быть выполнены в запланированные сроки;
- ✓ бюджет не превышен;
- ✓ результат удовлетворяет заказчика и востребован им.

Чтобы проект соответствовал этим критериям, на его начальном этапе менеджер должен задать себе несколько вопросов:

- Кто является заказчиком проекта?
- Очевиден ли для всех результат проекта?
- Нужно ли рассматривать промышленные или государственные ограничения?
- Имеет ли проект обоснованные ограничения по времени?
- Выделены ли средства для реализации проекта?
- Есть ли аналоги в отрасли?
- Влияют ли на проект смежные системы?
- Нет ли других вариантов?

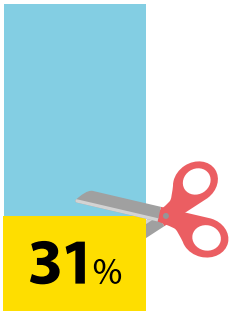
Формирование цели проекта

Определившись с видением проекта, следует перейти к его цели. Она должна быть руководящим посылом для всех участников команды и содержать информацию о том, что именно и в какие сроки необходимо сделать. Каждый член команды должен знать цель и четко ей следовать, при этом она должна исходить из объективных предпосылок.

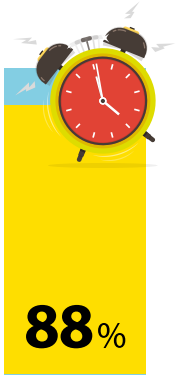
При формировании цели проекта нужно учитывать вполне определенные вопросы: подходят ли существующие Hardware и ОС для работы с разрабатываемой системой, необходимо ли переучивать пользователей или достаточно ознакомить их с новым решением, потребуется ли измененять бизнес-процессы работы конечных пользователей после внедрения системы и т.д. Все эти вопросы необходимо задать в самом начале проекта при постановке цели.

На срок завершения проекта могут влиять и другие факторы, например, бизнес-циклы компании. Так, во многих крупных банках, являющихся нашими заказчиками, существует мораторий на введение в эксплуатацию систем в декабре, есть понятие «высокого сезона», к наступлению которого необходимо завершить внедрение. Кроме того, не исключено изменение сроков закупки необходимого оборудования, это может быть связано, например, со сроками выплаты налогов компанией. При планировании получения виз от согласующих лиц (стейкхолдеров, спонсоров) необходимо учитывать планы по обучению, командировкам и отпускам сотрудников.

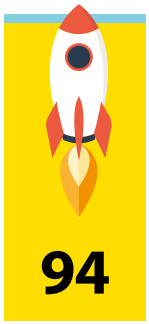
По данным Standish Group, менеджмент ИТ-проекта не всегда бывает успешным:



проектов прерываются до своего завершения



проектов не укладываются в сроки



проекта приходится начинать заново из 100 начатых

средняя стоимость превышается на 189%

среднее время выполнения превышается на 222%

Определив цель проекта, можно перейти к созданию его устава – официального и подробного изложения цели, согласованного с общим направлением развития компании. В документе прописываются все сроки, риски, ограничения, допущения, планы коммуникаций проекта.

Конфликт проектов по целям

В крупной компании с большим количеством одновременно реализуемых проектов нужно своевременно определять их приоритет, внимательно отслеживать вероятность их пересечения, следить за единством технологических решений, не допускать взаимоисключающих проектов в разных департаментах. Также необходимо согласовывать стратегии и перспективные планы подразделений, сводить вместе менеджеров различных подразделений и проектов, а также формировать механизмы и правила разрешения конфликтов между проектами. В компаниях эти цели обычно выполняет проектный офис: предоставляет услуги по менеджменту проектов для всей компании и при этом является руководящей структурой для всех проектов. Наличие PMO обеспечивает сбалансированный и взвешенный подход при разрешении конфликтных ситуаций, выделении бюджета, распределении ресурсов.

А вот его отсутствие может привести к нежелательным последствиям. Например, в одном крупном российском банке «железо», уже планируемое к закупке в рамках проекта, служба эксплуатации компании отказалась принимать на поддержку из-за неудовлетворительных результатов его опытно-промышленной эксплуатации в ходе другого внедрения. Проблемы можно было бы избежать при наличии единого архитектора и скоординированной работы проектного офиса.

На крупных проектах обычно формируется совместная команда из представителей исполнителя и заказчика. При этом не исключена ситуация, когда в нее могут попасть люди, не заинтересованные в успешном завершении проекта, не стремящиеся к максимальному участию в коллективной работе. Роль менеджера заключается в том, чтобы вовремя поставить «диагноз», идентифицировать и заинтересовать этих людей, либо вывести их из проекта/свести их влияние на результат к минимуму.

Отметим, что в ИТ-индустрии зачастую недостаточно добиться указанных в уставе целей проекта. Важно грамотно выстроить команду и в дальнейшем обеспечивать ее слаженную работу (в том числе и на других, смежных проектах), потому что люди – это 90% успеха. Ниже описаны принципы работы проектной команды и ответственность каждой группы ее участников.

КОМАНДА

Менеджер проекта

Менеджер проекта по разработке ПО должен одновременно обладать качествами шахматиста, тренера футбольной команды, режиссера, командира подлодки и даже предсказателя будущего. Как шахматисту ему необходимо просчитывать свои действия на много ходов вперед, как режиссеру – работать со «звездами», как тренеру – вести команду к победе, как командиру корабля – знать обо всем, что происходит на борту, какими бы незначительными эти события ни казались на первый взгляд, как предсказатель он должен предвидеть возможные проблемы.

Ему необходимо не только сформировать и держать карту проекта у себя в голове, но и донести эту информацию до всех членов команды. Нарисовать такую карту и найти точный и короткий путь к цели помогут интервью с людьми, принимающими решения в компании, с конкретными пользователями, со всеми

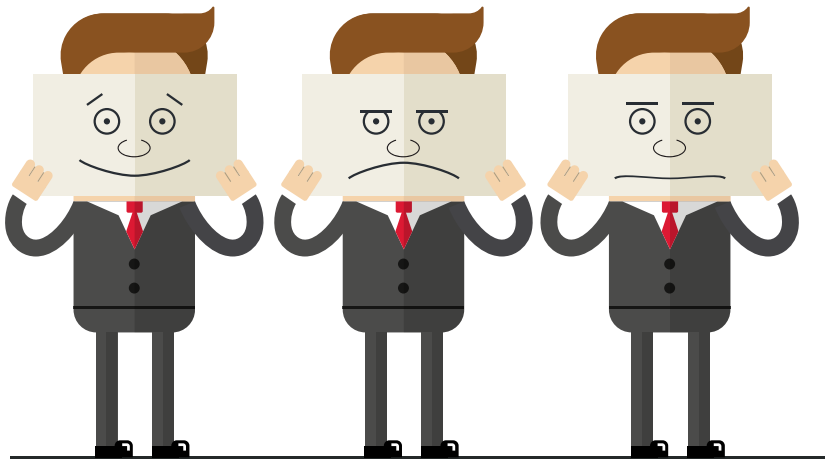
сотрудниками со стороны заказчика, участвующими в проекте, со стейкхолдерами, с членами проектной команды.

Например, в проекте по построению единого интеграционного решения для компании «Миг Кредит» мы провели более 50 интервью с сотрудниками различных функциональных подразделений заказчика для выявления потребностей бизнеса. Благодаря такому скрупулезному подходу мы выстроили единый конвейер, который обеспечил работу всей организации от входящего звонка клиента до работы с ним после погашения займа.

Аналитики

В подавляющем большинстве случаев проект начинается со списка требований, изложенного на 1–2 страницах или озвученного во время встречи. Здесь первую скрипку играют бизнес-аналитики: именно они проводят интервью с пользователями, фиксируют функциональные требования к системе, встречаются с представителями службы эксплуатации заказчика и определяют нефункциональные требования. Совместно с архитектором и ИТ-подразделением компании определяется технологический стек проекта, разрабатывается и согласовывается с заказчиком техническое задание. Это действительно масштабная работа, поскольку заказчик не всегда до конца представляет, что именно должно стать результатом проекта, и очень редко – как система должна работать в деталях.

После окончания этапа аналитики архитектор и менеджер проекта могут оценить его бюджет и сроки и начать этап проектирования системы. На этой стадии солирует архитектор, однако один в поле не воин, поэтому аналитики также серьезно вовлечены в процесс проектирования.



Стоит сказать, что члены команды могут быть оптимистами, пессимистами или реалистами, менеджер же должен совмещать в себе все 3 ипостаси: быть оптимистом с командой, чтобы мотивировать людей, пессимистом втайне от других, чтобы учитывать наихудший сценарий развития событий, и реалистом, чтобы в каждый конкретный момент видеть всю картину целиком, без прикрас и эмоций.

Архитектор

Его основная задача – спроектировать архитектуру приложения в соответствии с выявленными требованиями, разбить систему на компоненты, выбрать протоколы, фреймворки. При проектировании учитываются нефункциональные требования по производительности, обеспечению надежности и т.д. Так же, как архитектор согласовывает техническое задание на выполнимость, аналитик согласовывает архитектуру приложения, описываемую в технорабочем проекте, на соответствие требованиям. В процессе разработки архитектор декомпозирует принятые решения на задачи, назначает их разработчикам, контролирует выполнение задач. Именно он является лицом, принимающим решения в вопросах, касающихся сложных технических проблем.

Разработчики

Помимо непосредственно программирования разработчики участвуют в проектировании. Высококлассный специалист может взять на себя целый компонент или подсистему, спроектировать его, декомпозировать на задачи и раздать их другим разработчикам.

Разработчик постоянно должен быть в тонусе – изучать новые технологии и практики, уметь быстро разобраться в новом для него фреймворке. Обязательное умение – написание поддерживаемого кода. Известно, что поддерживать даже свой код через несколько месяцев становится

трудно, если заранее не озаботиться подробными комментариями, чистотой кода (правильными именами переменных и т.д.).

Тестировщики

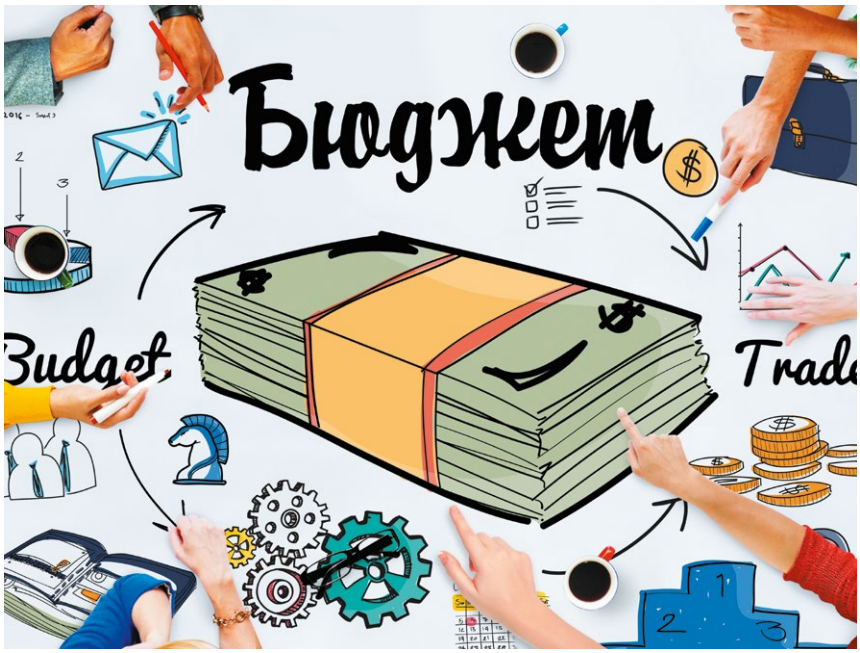
Помимо навыков программирования, тестировщику потребуются знания, которые обычно ассоциируют с архитекторами, системными аналитиками и администраторами сетей. «Сломать» систему не так трудно, гораздо сложнее понять, в чем проблема и как ее можно решить. Например, чтобы правильно протестировать систему, содержащую базу данных, сначала нужно разобраться в принципах построения БД; для проверки телеком-решений нужно знать протоколы, которые в них используются, бизнес-процессы, архитектуру решений.

Тестировщики в ряде случаев работают «классическим способом» – только руками и головой – и не пользуются средствами автоматизации. Огромную роль в таком тестировании играет человеческий фактор: эксперту нужно уметь ставить себя на место пользователя, анализируя при этом поведение программы.

Технологии

Менеджмент ИТ-проектов – это способность одновременно учитывать технологические и бизнес-пожелания заказчика, возможности и интересы членов команды, сроки и бюджет проекта. Найти этот баланс зачастую очень нелегко.

Если выбранная под проект технология новая, впечатляет заявленными возможностями, но пока не имеет широкого распространения, во многих случаях лучше подождать, пока ее обкатают, а пока выбрать более известный вариант. Слишком часто технология служит оправданием всего остального, что приводит к непредвиденным проблемам на проекте. Менеджер проекта должен быть готов сказать «нет» или, по крайней мере, «мы вернемся к этому проекту позже», грамотно и подробно обосновав свою позицию. Приведем пример: один крупный российский банк выбрал для себя известное на рынке решение, которое уже стоит в крупнейшем западном банке. Референс-визиты, довольный заказчик – на первый взгляд, все замечательно. Однако при его внедрении становится очевидно, что система не соответствует бизнес-процессам заказчика и не обеспечивает требования российских регуляторов. Специалистам пришлось приложить массу усилий для «заточки и подгонки» решения для удовлетворения потребностей конечных пользователей.



Отметим, что ИТ-проекты имеют опасную «привычку» быстро разрастаться, терять управляемость и превышать планируемый размер и бюджет. Иногда возникают ситуации, когда бюджет проекта не утвержден даже после начала работы проектной команды, и менеджер в таких случаях работает над сокращением второстепенных дополнений к решению, препятствующих соответствию проекта желаемым рамкам (он занимается так называемым «срезанием углов»).

***

Как показывает наша практика, правильное формирование целей проекта, грамотное управление проектной командой, выстраивание продуктивных отношений между всеми ее участниками и взвешенный подход к выбору технологий являются залогом успеха практически любого проекта по разработке ПО. JI

СОЗДАНИЕ МОДУЛЬНЫХ ПРИЛОЖЕНИЙ НА JAVA

РОМАН КИЧАСОВ,
ведущий архитектор
Центра программных решений
компании «Инфосистемы Джет»

Общеизвестно, что в языке программирования Java отсутствует поддержка модульности. Эта проблема была замечена давно, предпринимались неоднократные попытки исправить ситуацию. Речь в первую очередь идет о проекте Jigsaw[1]. Сначала планировалось добавить поддержку модульности в Java 7 (2011 год), потом в Java 8 (2014 год), наконец было анонсировано, что Jigsaw станет частью Java 9 (2017 год). На момент подготовки статьи 3 последние заметки в персональном блоге главного архитектора Java Марка Рейнхолда были посвящены внесению модульности в Java.

Так что же такое модульность, какие преимущества дает модульная архитектура ПО и что делать до появления Java 9? Модульная архитектура программного обеспечения – это подход к разработке, при котором программа разбивается на независимые компоненты, которые могут взаимодействовать между собой посредством четко описанных контрактов. Эти компоненты – модули – должны обладать несколькими признаками. Первый – инкапсуляция (они должны скрывать свою реализацию от внешнего окружения), второй – слабая связность (модули взаимодействуют только с помощью заранее оговоренных контрактов), третий – динамичность (возможность замещаться «на лету», без необходимости остановки всего приложения).

OSGi

Что делать, когда стоит задача разработать по-настоящему модульную архитектуру на языке Java? На помощь приходит OSGi (Open Services Gateway Initiative) – спецификация динамической модульной системы и сервисной платформы для Java-приложений, разрабатываемая OSGi Alliance. Спецификация описывает такую модель разработки ПО, при которой компоненты обладают всеми тремя описанными выше признаками. Основой концепции OSGi являются 2 сущности: наборы (Bundles) и сервисы (Services).

Наборы

При разработке ПО на Java, как правило, используются сторонние библиотеки. В мире Java библиотеки упаковываются в специальные файлы с расширением JAR (Java ARchive). Это обычный ZIP-архив, в котором находятся Java-классы (файлы с расширением .class). При этом использование библиотек может быть сопряжено с определенными сложностями.

Как только вы начинаете использовать какую-либо библиотеку в своем приложении, вам, как правило, становятся доступны все классы в этой библиотеке. Дело в том, что разработчики библиотек не имеют возможность спрятать классы, которые используются для реализации их внутренней логики. Таким образом, если вы используете код, который не предполагался для применения вне библиотеки, вы можете столкнуться с несовместимостью при использовании новой версии библиотеки или нарушить ее корректное функционирование.

Еще одной проблемой является так называемый JAR Hell. Многие разработчики сломали об него немало копий. Суть состоит в том, что как только у вас в проекте начинают использоваться разные версии одной и той же библиотеки (как правило, это большие проекты, которые эволюционируют со временем), вы можете столкнуться с ситуацией, когда один и тот же класс имеет разные методы в разных версиях библиотеки. Java же устроена так, что будет использована первая версия библиотеки, которую найдет загрузчик классов. Тем самым, обратившись в коде к какому-либо классу во время выполнения программы, вы получите ошибку, что метод, к которому вы обраща-

> Используя Fuse Fabric, мы можем создавать микросервисную архитектуру, которая поддерживает простоту настройки и развертывания, изолированное выполнение сервисов в рамках своих JVM-процессов (со своими специфичными настройками, например, разными настройками сборщика мусора), при этом сервисы взаимодействуют между собой по сети с использованием легковесного протокола.

[1] Jigsaw puzzle – составная картинка-загадка (головоломка, в которой нужно сложить отдельные кусочки, чтобы получилась картинка). Каждый кусочек (модуль) соединяется с другими (посредством интерфейсов), для того чтобы получилась законченная картинка (программное обеспечение).

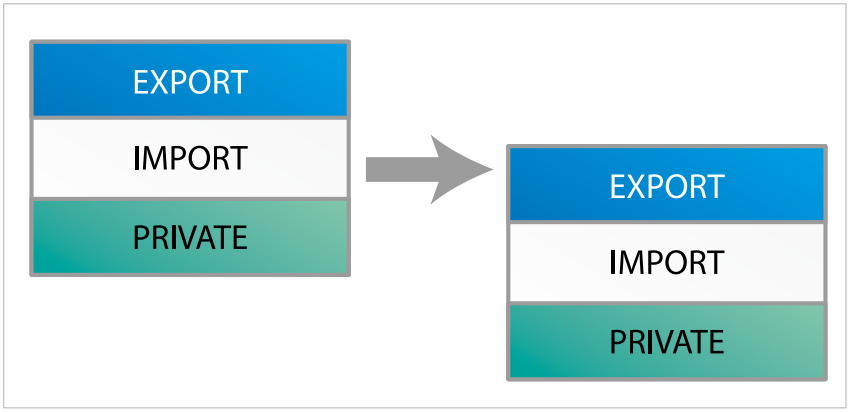


Рис. 1. Импорт/экспорт пакетов в OSGi-наборах

етесь, не существует. Связано это с тем, что на этапе выполнения Java ничего не знает о версии библиотеки, которая должна использоваться в том или ином случае.

Стоит отметить, что разработчики OSGi решили не изменять структуру JAR-файлов для обеспечения модульности, а просто добавили в них дополнительную информацию, которая используется средой OSGi. Более того, эта дополнительная информация никак не влияет на использование JAR-файлов в обычных Java-приложениях. Итак, чтобы JAR-файл стал OSGi-набором, в него добавляются данные, которые определяют, какие пакеты данного набора доступны для использования вне его (Export-Package) и какие пакеты других наборов требуются для работы этого набора (Import-Package). При этом можно задать как версию API, которую набор предоставляет для других наборов, так и версию или диапазон версий API, которые набор требует для своей работы от них же. Все классы набора, которые не находятся в его экспортируемой секции, не доступны вне набора (Private). Таким способом OSGi-набор выполняет требование слабой связности.

Сейчас большинство Java-библиотек уже являются OSGi ready, т.е. содержат информацию для возможности выполнения в OSGi-контейнере. Также существует множество инструментов и утилит, которые позволяют создать из обычных JAR-файлов модули для OSGi.

Так как зависимости между наборами и интерфейсы, предоставляемые этими наборами, имеют четкие версии, среда выполнения

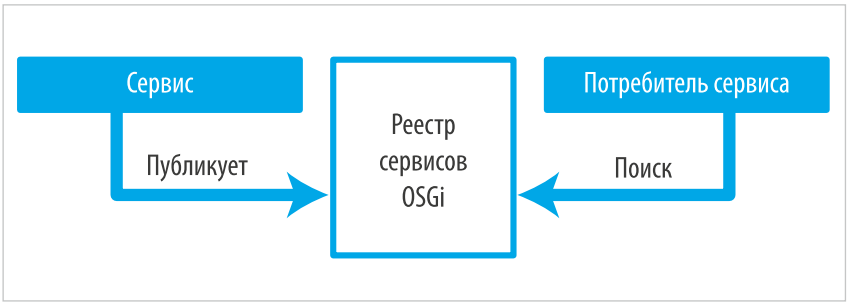


Рис. 2. Публикация OSGi-сервиса в реестр сервисов. Получение опубликованного OSGi-сервиса из реестра сервисов

OSGi позволяет легко избегать ситуаций JAR Hell, которые мы описали выше.

Таким образом, немного расширив описание обычного JAR-файла и добавив поддержку этого описания в среду выполнения, OSGi-сообщество решило проблему создания модулей в Java. Среда выполнения OSGi позволяет динамически загружать и выгружать новые наборы во время выполнения. Более того, OSGi отслеживает зависимости между наборами и динамически разрешает их.

Сервисы

Итак, имея OSGi-наборы, мы можем разрабатывать модульные приложения, которые взаимодействуют посредством интерфейсов (API). Возникает вопрос: где взять класс, который реализует требуемый интерфейс? Добавить такой класс в API набора – плохое решение, так как в рамках модульной архитектуры мы договорились не использовать внутренние классы наборов вне этих наборов. Можно было бы использовать шаблон «фабрика» для реализации интерфейса и добавить его в API набора, но разрабатывать каждый раз новый класс для сокрытия реализации интерфейса тоже кажется не лучшей идеей.

OSGi решает задачу поиска реализации интерфейса посредством реестра сервисов. Набор может зарегистрировать реализацию с описывающим ее интерфейсом в реестре сервисов. Набор, который использует интерфейс из другого набора, может выполнить поиск в реестре требуемой реализации нужного ему интерфейса. Как правило, наборы регистрируют сервисы при запуске в OSGi-контейнере. Более того, в реестре сервисов могут быть зарегистрированы одни и те же интерфейсы с разными реа-

лизациями и дополнительными идентификационными данными. При этом набор, который ищет в реестре требуемый ему сервис, может выбрать наиболее подходящий (например, используя фильтрацию).

МИКРОСЕРВИСНАЯ АРХИТЕКТУРА В ВИРТУАЛЬНОЙ МАШИНЕ JAVA

Некоторое время назад микросервисная архитектура наделала много шума в ИТ-сообществе. Особенно бурно ее стали обсуждать после того, как признанный гуру Мартин Фаулер в соавторстве с Джеймсом Льюисом опубликовал статью «Microservices», в которой подробно рассмотрел преимущества разбиения монолитной архитектуры на набор независимых модулей. Каждый из таких модулей представляет отдельное приложение. Каждое приложение разрабатывается и развертывается независимо от других. Взаимодействие между модулями происходит посредством четко определенных интерфейсов с использованием легковесных протоколов (REST, Protocol Buffers, MQ и т.д.). По факту каждый модуль – это микросервис, который выполняет одну определенную задачу и, как правило, содержит минимальное количество кода. Преимуществами такого подхода к разработке ПО являются:

- Легкость (при разработке микросервиса выполняется реализация всего одной конкретной части функциональности. При этом нет нужды беспокоиться о других частях программы).
- Простота замены (если вы решите, что в текущей реализации сервис не справляется со своей задачей, вы легко можете переписать его и заменить на новую версию без влияния на другие части программы. Более того, без остановки текущего ПО).

> Модульная архитектура программного обеспечения – это подход к разработке, при котором программа разбивается на независимые компоненты, которые могут взаимодействовать между собой посредством четко описанных контрактов.

- Повторное использование (так как микросервис выполняет одну маленькую задачу, он может многократно использоваться там, где это необходимо).

Как видно из описания выше, разработчики модульных приложений с применением OSGi давно пользуются всеми преимуществами микросервисной архитектуры, но только в рамках виртуальной машины Java. Каждый набор с OSGi, который публикует сервис в реестр, является микросервисом внутри Java Virtual Machine, JVM (в мире OSGi такие микросервисы называются µServices).

Red Hat JBoss Fuse

Мы в Центре программных решений используем все преимущества OSGi при разработке ПО для телеком-операторов, страховых, процессинговых компаний. Для этого мы применяем продукт Red Hat JBoss Fuse, а конкретно – его конфигурацию Fuse Fabric. Платформа JBoss Fuse предоставляет гибкую OSGi-среду для выполнения модульного приложения.

Сегодня к ПО предъявляются такие требования, как непрерывность работы, возможность легкого горизонтального масштабирования, простота развертывания и наличие средств управления кластером для кластерного ПО. Fuse Fabric решает эти задачи. Технология позволяет развернуть несколько экземпляров Red Hat JBoss Fuse и объединить их в кластер, а также предоставляет централизованный инструмент управления получившейся средой.

В рамках Fuse Fabric существуют следующие абстракции:

Фичи (features) – совокупность OSGi-наборов, которые реализуют какой-либо функционал.

Профили (profiles) – совокупность фич, которые должны выполняться в рамках одного контейнера, и конфигурационные настройки для наборов, которые входят в фичу.

Контейнеры (containers) – отдельные JVM-процессы, которые выполняются на конкретном узле кластера Fuse Fabric под управлением контейнера Red Hat JBoss Fuse.

Таким образом, любое ПО, которое строится на технологии Fuse Fabric, состоит из OSGi-наборов, которые группируются в фичи и развертываются в рамках какого-либо отдельного JVM-процесса на одном или нескольких узлах кластера в соответствии с заранее заданным профилем.

Среда Fuse Fabric содержит множество инструментов, которые позволяют легко управлять полученным кластером. Мы можем создавать профили, на основе профилей создавать контейнеры, создавать/удалять/запускать/останавливать контейнеры на любом хосте, который входит

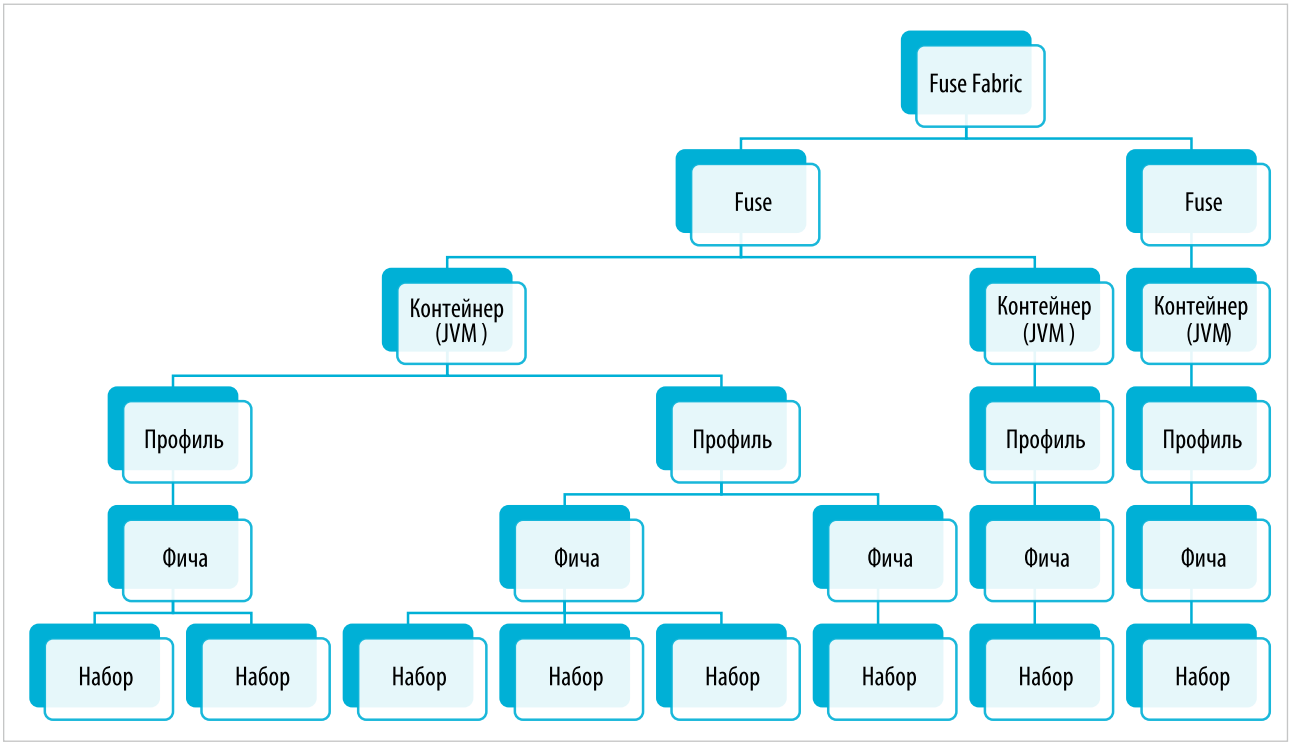


Рис. 3. Иерархия сущностей в Fuse Fabric

в кластер, подключать новые узлы кластера и т.д. И все эти действия мы можем выполнять в режиме online, т.е. без прерывания функционирования остальных узлов кластера. Среда поддерживает возможность хранения нескольких версий профилей, фич, OSGi-наборов.

В полученной среде OSGi-наборы могут взаимодействовать не только в рамках одного контейнера, но и вызывать друг друга в рамках разных контейнеров (даже в рамках различных хостов). Эту возможность обеспечивает технология Distributed OSGi. Дополнительно – при минимальных затратах на разработку – каждый сервис, который предоставляет OSGi-набор, можно превратить в REST/web-сервис, который можно вызывать из внешних систем.

Таким образом, используя Fuse Fabric, мы можем создавать микросервисную архитектуру, которая поддерживает простоту настройки и развертывания, изолированное выполнение сервисов в рамках своих JVM-процессов (со своими специфичными настройками, например, разными настройками сборщика мусора), при этом сервисы взаимодействуют между собой по сети с использованием легковесного протокола.

> Для того чтобы JAR-файл стал OSGi-набором, в него добавляются данные, которые определяют, какие пакеты данного набора доступны для использования вне его (Export-Package) и какие пакеты других наборов требуются для работы этого набора (Import-Package).

Так как продукт Red Hat JBoss Fuse основан на Open Source стеке технологий Apache ServiceMix, в нашем распоряжении есть такие мощные технологии, как Apache ActiveMQ (реализация спецификации JMS), Apache Camel (реализация шаблонов проектирования корпоративных приложений), Apache CXF (библиотека для разработки REST/web-сервисов), Blueprint (предоставляет возможности внедрения зависимостей), Spring и т.д. Поскольку указанные технологии бесшовно интегрируются между собой в Red Hat JBoss Fuse, это значительно снижает время разработки требуемого функционала. JI

РАЗРАБОТКА CPA-ПЛАТФОРМЫ ДЛЯ ТЕЛЕКОМ-ОПЕРАТОРА

РОМАН КИЧАСОВ,
ведущий архитектор
Центра программных решений
компании «Инфосистемы Джет»

В этой статье мы опишем пример использования модульного подхода в одном из наших проектов. Мы реализовали модульную архитектуру CPA-платформы (Content Provider Access) для телеком-оператора на Fuse Fabric. Мы также скажем о том, почему приняли такое решение, а не стали использовать стандартный стек технологий J2EE для создания монолитного приложения.

CPA – это платформа для организации взаимодействия с контент-провайдерами, т.е. партнерами, которые предоставляют какие-либо услуги абонентам телеком-оператора. В качестве простейшего сценария можно рассмотреть ситуацию, когда абоненту требуется узнать погоду на завтра:

1. абонент отправляет SMS-сообщение на короткий номер для получения прогноза погоды;
2. SMS-сообщение поступает в SMS-центр телеком-оператора;
3. SMS-центр отправляет сообщение в CPA-платформу;
4. CPA-платформа определяет услугу, привязанную к короткому номеру, определяет стоимость услуги, проверяет наличие на счету абонента достаточного количества средств, запрашивает у партнера, предоставляющего услугу прогноз погоды, списывает средства со счета абонента и протоколирует бизнес-операции, передает контент в SMS-центр для отправки абоненту;
5. SMS-центр отправляет абоненту SMS-сообщение с погодой на завтра;
6. абонент получает SMS-сообщение.

Даже в таком сценарии можно увидеть базовые модули, которые участвуют во взаимодействии: модуль для взаимодействия с SMS-центром, каталог услуг, каталог партнеров, модуль работы с биллинговой системой, модуль протоколирования бизнес-операций, модуль взаимодействия с партнером и т.д.

При этом мы рассмотрели лишь простейший сценарий, а у телеком-операторов их могут быть десятки и в них могут быть задействованы разные системы. Например, запрос может поступить с USSD-центра, IVR-центра, web-портала и т.д. Либо это может быть не разовый запрос на получение услуги, а подписка на услугу (в том числе в рассрочку), отписка от нее и т.д.

Когда мы начинали проект по созданию CPA-системы, то рассматривали несколько вариантов архитектуры платформы. При проектировании приложений такого масштаба в первую очередь нужно учитывать следующие моменты:

- Нагрузка на систему будет очень большой, т.е. она должна обрабатывать сотни достаточно сложных бизнес-сценариев в секунду. К этому следует добавить возможность простого горизонтального масштабирования системы при увеличении нагрузки.
- Класс систем такого типа – это business critical или mission critical, т.е. время простоя системы должно быть минимальным даже при внесении каких-либо изменений в нее (например, добавление новой функциональности или исправление каких-либо дефектов).
- Система должна легко и централизованно управляться. Должны быть четкие метрики, говорящие о ее текущем состоянии.
- Компоненты системы должны легко заменяться: в мобильной связи изменения происходят так быстро, что сегодня вы взаимодействуете с одной системой по одному протоколу, а завтра с другой – по другому.

> Со временем сложился стереотип, что для разработки сложных корпоративных систем следует использовать J2EE и традиционные серверы приложений. В мире разработки принято разрабатывать монолитные приложения, и отсутствие массовых решений на технологии OSGi (по сравнению с J2EE) вызывает сомнения в ее применимости.

- Время разработки нового функционала и его тестирования должно быть минимальным, так как решающую роль играет такая вещь, как time to market.

Мы быстро осознали, что создание традиционной монолитной J2EE-архитектуры нам не подходит, так как не дает требуемой гибкости и не закрывает все потребности заказчика. Наш выбор остановился на модульной архитектуре на основе OSGi – Red Hat JBoss Fuse в конфигурации Fuse Fabric.

Нам нужно было построить такую архитектуру, в основе которой лежало бы взаимодействие различных автономных модулей на основе четко определенных интерфейсов, т.е. создать микросервисную архитектуру в текущем понимании этого термина (правда, на момент начала работ по созданию CPA-платформы слово

«микросервисы» не было таким модным, как сейчас).

Отметим, что неправильно выбранная архитектура для систем такого уровня буквально может привести к катастрофическим последствиям как на этапе разработки и эксплуатации, так и на этапах дальнейшей доработки системы для расширения ее функциональных возможностей. Нам требовалось как можно раньше показать заказчику так называемый proof of concept выбранной идеи и ее жизнеспособность. И здесь на помощь пришла поддержка модульности в OSGi.

В основу проектирования системы мы положили API модулей, из которых она должна состоять, и BPM-процессы (бизнес-процессы, описанные с помощью стандартной нотации BPMN). Последние описывали бизнес-сценарии, которые выполняет CPA-платформа, в виде вызовов API модулей. Иными словами на начальном этапе мы имели модули (OSGi-наборы), которые, по сути, представляли собой микросервисы в виде «заглушек», выполняющие примитивные операции, и BPM-процессы, которые максимально приближенно к бизнес-сценариям телеком-оператора описывали последовательность вызовов OSGi-сервисов для достижения требуемого результата.

При этом мы понимали, что создаваемые нами OSGi-наборы будут вызываться как в BPM-сценариях, так и из других мест. Так, например, сервис для работы с каталогом услуг не только непосредственно вызывается из BPM-процессов, но и используется в back-office платформы CPA для обеспечения возможности контент-менеджерам заводить, удалять, редактировать услуги. Также API сервисного каталога используется для предоставления списка услуг на сайте телеком-оператора. Сервис работы с биллинговой системой сначала использовался только в CPA-платформе, а сейчас благодаря тому, что он также доступен в виде REST-сервиса, его услугами пользуются и другие системы в инфраструктуре телеком-оператора.

Разработка архитектуры в виде модулей с четко специфицированным API позволила нам в кратчайшие сроки увидеть, что создаваемая архитектура целостна и работоспособна.

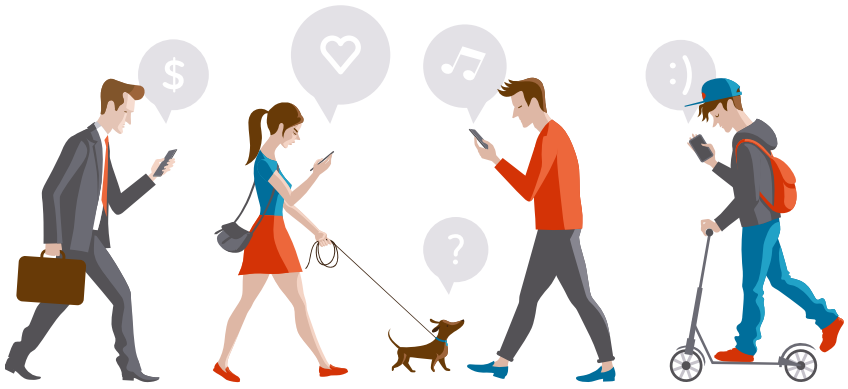
Так как API наборов был согласован и зафиксирован, это дало нам возможность распараллелить процесс разработки каждого OSGi-набора, т.е. отдельные команды могли заниматься разработкой какого-либо OSGi-набора. При этом одна команда разработки никак не влияла на другую, так как работала со своей кодовой базой и предметной областью. Все, что должно было быть соблюдено, – это ненарушение согласованного API-набора. Одна команда могла сосредоточиться на разработке партнерского каталога, другая – сервиса бизнес-логирования и т.д. При этом архитектор следил за целостностью системы на основе согласованных API.

Для облегчения процесса отслеживания изменений API мы взяли на вооружение семантическое версионирование наборов OSGi. Этот подход предлагает использовать версии компонент в виде *X.Y.Z*, где *X* – мажорная версия, *Y* – минорная версия, *Z* – патч-версия. При этом версии увеличиваются по следующим правилам:

- мажорную версию увеличиваем, когда сделаны обратно несовместимые изменения API;
- минорную версию увеличиваем, когда добавляется новый функционал, не нарушающий обратной совместимости;
- патч-версию увеличиваем, когда добавляются обратно совместимые исправления, не затрагивающие API.

Это позволило нам видеть изменения API сервисов в рамках всей системы и управлять ими при выстраивании совместной работы сервисов.

При разработке сложных систем также требуется разрабатывать тесты на реализованный функционал. Так как взаимодействие компонентов в среде OSGi организовано путем вызова интерфейсов, это в значительной степени упрощает модульное тестирование. Если какой-либо сервис в нашем приложении обращается к другому сервису, на этапе тестирования тот сервис, к которому происходит обращение, мы можем легко заменять



«заглушкой» для тестов. При этом при тестировании каждого модуля мы сосредотачиваемся именно на его специфике работы.

В процессе разработки мы понимали, что любой сервис в зависимости от характера нагрузки может стать узким местом всего приложения. Поэтому каждый вызов API какого-либо OSGi-набора был обернут метриками, которые были доступны по JMX-интерфейсу. Это позволило нам видеть максимальное время вызова какого-либо метода сервиса, усредненное за промежуток времени значение и характер изменения метрики. Данные метрики позволили нам сразу идентифицировать узкие места приложения во время нагрузочного тестирования, кроме того, они дали возможность наблюдать за продуктивной системой в режиме online и при деградации производительности оперативно принимать какие-то решения по стабилизации системы. Например, мы можем перенести сервис на другой узел среды выполнения или вынести его в отдельную виртуальную машину Java. При этом OSGi позволяет нам выполнять все эти действия на работающей системе, не влияя на остальные части приложения.

При добавлении нового функционала в существующую систему у нас появилась возможность явно отслеживать изменившуюся кодовую базу (новые наборы или наборы, у которых увеличилась версия) и сократить время на тестирование. Если мы видим, что в очередной поставке не изменился номер версии компонента, на него не требуется разрабатывать новых тестов и повторно тщательно тестировать. Лучше уделить внимание новому функционалу, который был добавлен в систему.

Конечно же, никто не застрахован от ошибок, которые могут проявиться в продуктивной среде. Но OSGi позволяет нам оперативно вносить изменения в работающую систему. Достаточно исправить ошибку в компоненте, увеличить его патч-версию и установить в среду исполнения OSGi. Это становится очевидным плюсом, когда время простоя системы в значительной степени влияет как на качество предоставляемого сервиса, так и на прибыль компании.

Также стоит отметить, что, в противоположность модульной архитектуре, при обнаружении ошибки в монолитной архитектуре приходится пересобирать все приложение и перезапускать его. Перезапуск традиционного J2EE-приложения, как правило, требует значительного количества времени, а значит, может обернуться потерями для компании.

***

Однако модульный подход не обходится и без минусов. Непросто преодолеть сложившийся стереотип, что для разработки сложных корпоративных систем следует использовать J2EE и традиционные серверы приложений. В мире разработки принято разрабатывать монолитные приложения, и отсутствие массовых решений на технологии OSGi[1] (по сравнению с J2EE) вызывает сомнения в ее применимости.

Нам кажется, что плюсы, которые дает OSGi, могут кардинально упростить как разработку ПО для корпоративного сектора, так и дальнейшее его сопровождение и развитие. Не стоит также забывать, что Java 9 уже идет к поддержке модульности (хотя и не в полном объеме, который уже реализован в OSGi), а плюсами микросервисной архитектуры пользуются такие успешные компании, как Netflix, Amazon, The Guardian и др.

> Нам нужно было построить такую архитектуру, в основе которой лежало бы взаимодействие различных автономных модулей на основе четко определенных интерфейсов, т.е. создать микросервисную архитектуру в текущем понимании этого термина (правда, на момент начала работ по созданию CPA-платформы слово «микросервисы» не было таким модным, как сейчас).

К сожалению, в рамках одной статьи невозможно рассказать обо всех преимуществах OSGi. Но одно очевидно – эта технология является хорошей альтернативой традиционным монолитным приложениям, которые на данный момент правят бал в корпоративном секторе. JI

[1] Наиболее известным и массовым решением на базе технологии OSGi является проект Eclipse, с которым знаком почти каждый Java-разработчик. Но не все знают, что в своей основе Eclipse использует реализацию спецификации OSGi под названием Equinox.

РАЗРАБОТКА СИТУАЦИОННЫХ ЦЕНТРОВ – ВЗГЛЯД ИЗНУТРИ

АЛЬБЕРТ НУРУТДИНОВ,
менеджер по развитию бизнеса
Центра программных решений
компании «Инфосистемы Джет»

В этой статье мы рассматриваем особенности разработки информационно-аналитических систем (ИАС) для ситуационных центров (СЦ). Для начала следует отметить, что создание СЦ – это сложный комплексный проект, в котором задействованы специалисты совершенно разных областей, начиная от аналитиков и методологов и заканчивая проектировщиками архитектурно-строительных решений и инженерных систем. Все это накладывает свой отпечаток на процесс разработки ИАС. СЦ – вещь строго иерархичная, поэтому нужно следовать принципу максимальной технологической согласованности проекта, принимая во внимание горизонтальные и вертикальные связи этого комплекса. Каждый уровень СЦ обеспечивает функционирование вышестоящего уровня, поэтому их рассмотрение по отдельности может привести к потере целостности решения.

Управление подобным проектом, а также архитектурный и технический надзор – основные вызовы, с которыми сталкивается проектная команда в ходе создания СЦ. В свою очередь, главная задача менеджера проекта и архитектора – это комплексная интеграция методологических, программных, технологических, инженерных и строительных решений, применяемых на каждом этапе создания комплекса, умелое привлечение и оркестровка профильных компетенций членов команды. Причем без потери леса за деревьями.

Теперь перейдем к сложностям, с которыми мы сталкивались в подобных проектах, и вариантам их решения. Один из наиболее сложных этапов при внедрении ИАС – этап сбора требований. Реализовав не один проект в этой сфере, мы пришли к выводу, что, к сожалению, накопленные компетенции наших аналитиков сложно переиспользуются в каждом новом СЦ. К примеру, мы создавали 2 на первый взгляд похожих ситуационных центра, причем в одной компании. Один из них автоматизировал управление кризисными ситуациями и оперативное реагирование, другой был предназначен для управления комплексной безопасностью. И эти СЦ в конечном итоге отличались так сильно, что порой опыт предыдущего проекта не помогал, а наоборот, играл против нас. Дело в том, что любой СЦ – это чаще всего верхушка определенной сферы деятельности заказчика, и от проекта к проекту деятельность, которую мы представляем посредством ситуационного центра, существенно отличается. Это как если бы вы строили заборы, но каждый раз это нужно было бы делать на разных планетах. Банковский аналитик без труда сориентируется в любом новом банке, а аналитик по СЦ каждый раз погружается в новую тематику.

> Наши аналитики, приступая к работе над новым ситуационным центром, готовы к тому, чтобы оставить за дверью большую часть накопленного на предыдущих проектах багажа знаний и погрузиться новую предметную область.

Разумеется, нельзя сказать, что нет ничего общего между модулями ИАС, функционирующими в разных СЦ. Модули анализа, планирования, прогнозирования, обеспечения взаимодействия в том или ином виде присутствуют везде, однако в каждом случае к ним предъявляются разные требования, да и смысл, вкладываемый заказчиком в тот или иной модуль, зачастую отличается.
К тому же деятельность в рамках самого СЦ – это почти всегда не детерминированный процесс, а кейс-менеджмент и поддержка принятия решений, которые крайне сложно поддаются моделированию и описанию (не существует блок-схем, способных с достаточной точностью описать процесс мозгового штурма). Поэтому наши аналитики, приступая к работе над новым ситуационным центром, готовы к тому, чтобы «оставить за дверью» большую часть накопленного на предыдущих проектах багажа знаний и погрузиться в новую предметную область.

При этом, как уже было отмечено выше, СЦ включает не только софт – это и комплекс помещений, и специализированные мультимедийные системы, и видеоконференцсвязь, и телефония, и т.д. Причем у каждого заказчика свои требования к подобным технологическим системам: кто-то смотрит на СЦ в первую очередь как на ИАС, кто-то привык идти от оборудования, которое в нем используется, а для кого-то ситуационный центр – это люди и регламенты. Например, органы государственной власти при построении СЦ чаще определяют его как помещение, оснащенное оборудованием, с которым работают люди, использующие различные программы. И ИАС в данном случае является лишь одной из составляющих СЦ. Но есть и другие

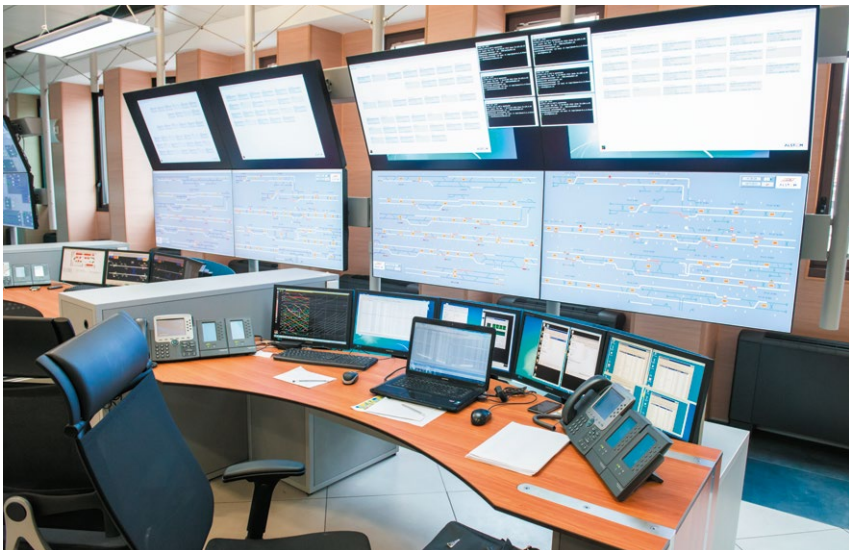
примеры. Так, в нашем проекте для крупной нефтяной компании информационная система (а именно подсистема поддержки принятия решений) являлась с точки зрения заказчика главным компонентом СЦ.

Так или иначе наши аналитики неизбежно расширяют кругозор, одновременно углубляясь в технические, методологические и программные аспекты. Мы пришли к выводу, что лучшим рецептом для грамотной реализации этой стадии проекта является как можно более раннее погружение аналитиков в деятельность компании, формирование на старте команды, которая на 100% погружена в проект и плотно взаимодействует с представителями заказчика.

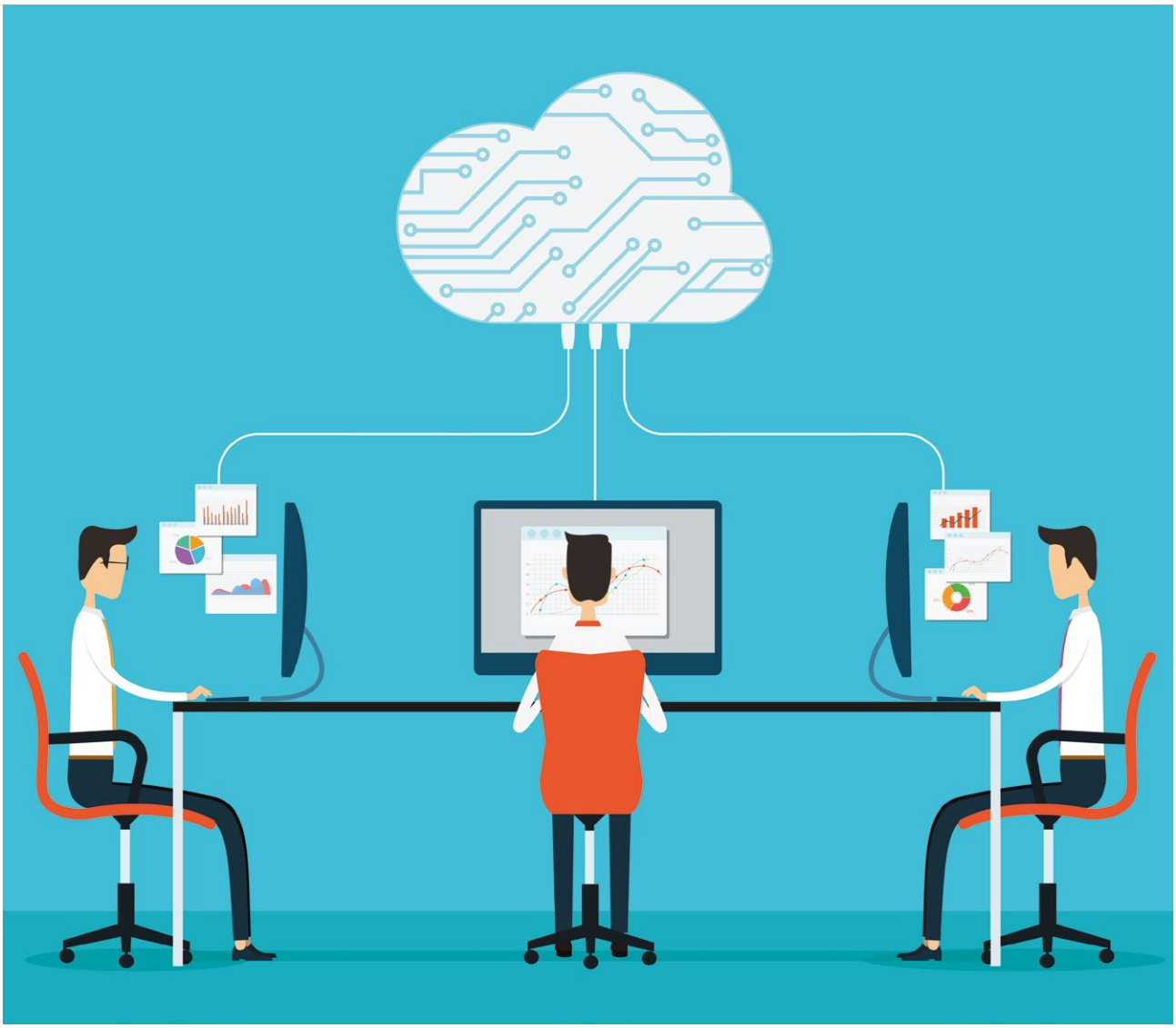
Стоит подчеркнуть, что тема СЦ – достаточно новая для нашей страны, по ней сформировано не так много best practices. Порой это приводит к тому, что заказчик не до конца понимает, что именно должна делать ИАС, его представление об этом может быть довольно размытым. Например, один из наших заказчиков на старте проекта хотел, чтобы система «думала», имея ввиду поддержку принятия решений, но, к сожалению, идеи не всегда были сформулированы достаточно четко для того, чтобы наши программисты смогли воплотить их в жизнь. В подобных ситуациях мы, опираясь на опыт и экспертизу, предлагаем заказчику уже наработанные практические кейсы и конкретику, даже рискуя «не попасть». Практика показала, несмотря на то что не все предложенные кейсы принимаются, сам процесс наталкивает заказчика на рассуждения в практической плоскости и генерацию более конкретных требований, которые мы впоследствии формализуем и реализуем.

Иногда на подобных проектах возникает дрейф требований. Ведь в их ходе проходит большое количество интересных и в то же время непредсказуемых с точки зрения планирования развития проекта мозговых штурмов. И это может повлечь за собой сложности, особенно когда на завершающих стадиях реализации прикладного ПО возникают требования, не реализуемые на архитектурном уровне. Обычно мы снижаем подобные риски двумя способами. Первый – это прототипирование. Еще до того как разработчики приступают к написанию первой строчки кода, мы показываем специалистам заказчика подробные описания сценариев работы, максимально приближенные к реальности макеты интерфейсов системы (до полей и кнопок), экранные формы отчетов (нередко в виде прототипа BI-системы с тестовыми данными). Вместе с итеративным процессом разработки это позволяет на ранних стадиях выявлять неучтенные (либо изменившиеся) требования.

Второй способ: мы используем в качестве ядра построения ИАС разработанную нами платформу-конструктор справочников. Эта платформа позволяет достаточно гибко модифицировать объекты и логику работы системы зачастую без необходимости привлечения разработчиков (либо с минимальным привлечением). Например, на одном проекте наш заказчик захотел серьезно поменять логику учета ЧС/происшествий. Требовалось, чтобы происшествие могло относиться сразу к нескольким типам одновременно (пожар + ДТП), при этом к нему применялись бы бизнес-правила обоих типов. Другой случай: потребовалось, чтобы параметры происшествия (в карточке происшествия их было более 200) могли вноситься не только сотрудниками самого СЦ, но и смежными подразделениями, причем независимо, с сохранением истории и возможностью динамически менять как состав полей, так и ответственных за их заполнение. В обоих случаях нам удалось практически без доработок реализовать новые пожелания, несмотря на то что они возникли уже по ходу проекта. Так что в подавляющем большинстве случаев мы готовы изменить систему даже под те требования, реализация которых в обычных условиях привела бы к принципиальному изменению программной архитектуры модулей.



ОБЛАЧНЫЙ ПОДХОД К РАЗРАБОТКЕ



БОРИС ТОБОТРАС,
руководитель Центра
программных решений
компании «Инфосистемы Джет»

Основные черты современных информационных систем (ИС) – гибкость и динамика. Программно-определяемые сети, системы хранения данных, ЦОД – все они ставят своей целью сделать ИС максимально адаптируемыми под нужды пользователя, создавая эластичную и оптимальную инфраструктуру.

Однако инфраструктура не нужна сама по себе: ее разворачивают ради прикладных решений, приносящих реальную пользу компании. Несмотря на поток новых технологий, фундаментальные основы построения прикладных решений в последние десятилетия менялись мало. И вот мы являемся свидетелями того, как новые технологии в инфраструктуре – программно-определяемые ЦОД и облачные сервисы – начинают трансформировать подход разработчиков к построению ИС.

МАСШТАБИРУЕМОЕ ПРИЛОЖЕНИЕ – ЧТО ТАКОЕ И ЗАЧЕМ

Сегодня термины «облака» и «облачные технологии» встречаются буквально на каждом шагу. Каждый понимает их по-своему: кто-то считает, что это очередные громкие слова, за которыми ничего не стоит, а кто-то – что это технологии, перевернувшие ИТ-индустрию. Не претендуя на охват темы целиком, мы предлагаем пощупать этого слона с одной стороны: как облака могут изменить архитектуру прикладных информационных систем?

Есть ли способы определить, что конкретная система может выиграть от применения в ней облачных технологий? Есть ли какие-то характерные признаки того, что именно вашу систему стоит развивать в облаке? Придется ли что-то менять в существующей системе или в подходе к разработке новой, чтобы получить отдачу от облака? В нашей статье мы ищем ответы на эти вопросы. Но для начала нужно разобраться с тем, что же такое cloud-среда.

Общепринятое определение облачного сервиса принял Американский национальный институт стандартов и технологий. Сервис должен обладать следующими основополагающими признаками:

1. Самообслуживание по требованию.
2. Универсальный доступ по сети.
3. Объединение ресурсов для обслуживания большого числа потребителей в единый пул для динамического перераспределения мощностей.
4. Эластичность услуг.
5. Учет потребления.

Для разработчика системы в первую очередь это означает, что теперь можно динамически менять «размер» ИС: добавлять новые ресурсы, в первую очередь, вычислительные (процессор/память) и ресурсы хранения (дисковое пространство или сервисы хранения) или отказываться от тех, что стали не нужны.

Облака впервые позволили решить старую и больную проблему масштабирования приложений. Типичный пример ее появления выглядит так: предположим, вы создаете коммерческий сервис, обслуживающий пользователей. В сервис вы заложили архитектуру, позволяющую легко масштабировать его горизонтально, например, добавлением серверов приложений с балансированием нагрузки. Но сколько серверов вам, собственно, нужно развернуть при запуске сервиса? Вы решаете, что, пока пользователей мало, достаточно двух, а дальше, по мере роста пользовательской базы, вы будете добавлять машины. Вы запускаете сервис, к вам приходят первые пользователи, все идет прекрасно. О вас даже написали на Хабре, Slashdot'е или Википедии. Через два часа вместо первой сотни пользователей к вам наведываются сто тысяч. И 99 из них, включая журналистов, потенциальных инвесторов или реальных заказчиков, увидят вместо сайта с вашим сервисом тайм-аут, или внутреннюю ошибку, или еще что-нибудь столь же неприятное.

> Типичный исходный текст программы 1990-х годов выглядит примерно так: зовем функцию, если что-то пошло не так, пишем сообщение об ошибке. Типичная современная программа, к сожалению, не может позволить себе подобный подход, потому что отказ не является исключительной ситуацией. Вы не можете вывести пользователю баннер, когда он пришел к вам через поисковый запрос Google, «у меня произошла ошибка на одном из тысячи серверов, попробуйте, пожалуйста, позже».

Наивная альтернатива этому сценарию – заранее продумать максимальную нагрузку и за-

пускать сервис на аппаратной платформе, рассчитанной под нее: например, вместо двух серверов – 50 или 500. И все они 99% времени будут проедать электричество и деньги на вашем хостинге, простаивая в ожидании того самого пикового всплеска посетителей, который может и не случиться.

Если Хабрахабр или Википедия кажутся вам надуманными источниками нагрузки, возьмем другой пример – интернет-магазин по продаже цветов. 360 дней в году его нагрузку выдержит один сервер, со всем front-end'ом, с приложением, базой данных и аналитикой. А оставшиеся дни – накануне 1 сентября и 8 марта – понадобится пять серверов. Или пятьдесят. Причём плановое подключение-отключение серверов – тоже не панацея: можно ошибиться, во-первых, с оценкой сайзинга, во-вторых, с планированием нагрузки (например, бывает еще – неожиданно – 14 февраля).

Даже если вы успели заметить рост нагрузки и готовы оперативно подключать/отключать новое оборудование, все равно «оперативно» здесь измеряется в лучшем случае часами. Реальному миру свойственны проблемы с логистикой.

С появлением облачных сервисов эту проблему можно легко решить. Ваш сервис должен масштабироваться по горизонтали и позволять отслеживать свою загрузку, а также реагировать на ее изменение. Тогда можно снабдить его несложной логикой вида «если в последние 30 секунд загрузка серверов приложений выше пороговой, добавить еще один сервер; если в последние 2 минуты загрузка серверов приложений ниже пороговой, убрать один сервер».

Слово «легко» в предыдущем абзаце, конечно, надо воспринимать с изрядной долей юмора. Чтобы это стало действительно легким, ваша система должна быть с самого начала спроектирована с учетом возможности переезда в облако. В первую очередь, это означает – используйте слабо связанные сервисы везде, где только можно. Один из первопроходцев облачных технологий компании Amazon Джефф Безос, говорят, в определенный момент издал всем командам разработки указ, смысл которого сводился к следующему:

- все команды публикуют свои данные и функциональность через сервисные интерфейсы;
- все команды обязаны взаимодействовать через эти интерфейсы;
- не допускается никакой другой формы межпроцессных коммуникаций: никакой прямой линковки, никакого прямого чтения из хранилища данных другой команды, никакой разделяемой памяти, никаких других «черных входов». Единственный разрешенный способ – сетевые вызовы сервисных интерфейсов;
- не имеет никакого значения, какую технологию использует та или иная команда;
- все сервисные интерфейсы без исключения обязаны проектироваться так, чтобы их можно было публиковать для использования разработчиками во внешнем мире.

Заканчивался указ следующим: «Любой, кто этого не сделает, будет уволен. Благодарю вас, удачного дня!».

И посмотрите, где сейчас бывший книжный интернет-магазин Amazon.

НАДЕЖНЫЕ СЕРВИСЫ НА ПРИНЦИПИАЛЬНО НЕНАДЕЖНОЙ ИНФРАСТРУКТУРЕ

Вынесение сервисов в облако имеет еще одно фундаментальное отличие от привычной моде-

ли разработки. Это отношение к отказам. Старый подход заключается в том, что мир вокруг разработчика, в общем-то, безоблачен и безотказен. А отказ, если он все же случается, – это повод выбросить белый флаг и сдаться: «невозможно установить соединение; извините, повторите ваш запрос позже». Новый подход состоит в том, что отказы случаются постоянно и с ними нужно уметь работать так же, как мы работаем с разными значениями входных данных. Например, поиск Google работает на колоссальных размеров вычислительной ферме, часть которой (кажется, процентов десять) не функционирует (сломано, отключено, в плановом обслуживании). Google принципиально использует самые дешевые аппаратные компоненты и легко заменяет их на новые при отказе. Что нужно, чтобы это работало? Программные компоненты должны стоически переносить эти отказы.

Для этого разработано множество технологий: кластеризация, обеспечивающая работу системы, пока поддерживается минимальный кворум сервисов; службы каталогов, дающие возможность динамически искать в сети работающие сервисы; репликация данных, позволяющая иметь избыточное количество доступных данных на случай отказа.

Но самое главное – мысль об обработке отказов должна все время быть в голове у разработчика. Эфемерные экземпляры сервисов могут создаваться и удаляться подсистемой мониторинга из-за изменения нагрузки на систему. Сервисы могут просто отказывать и переставать работать. Они могут становиться недоступными (а потом снова доступными) из-за проблем с сетевой связностью. Недоступность сервиса, обрыв соединения, внутренняя ошибка – это одинаково вероятные события, реакцией на которые должно быть не бросание исключения («все сломалось!»), а, например, повторный выбор узла с сервисом и еще одна попытка вызова.

> В случае использования SOA приложение представляет собой набор независимых сервисов. Каждый компонент программы предоставляет другим компонентам определенные услуги и одновременно выступает в роли потребителя услуг «соседей». Границы между сервисами явно видны даже конечным пользователям программы. Вместо того чтобы возводить монолитное громоздкое здание из бетона, мы берем и соединяем отдельные подвижные блоки.

Технологии масштабирования

Для того чтобы ваша система могла масштабироваться, должны быть разработаны простые и надежные механизмы. Назовем некоторые из них.

Мониторинг и метрики

Как решить, в какие моменты пора разворачивать/сворачивать сервис? На этот вопрос отвечают метрики, за которыми следит система мониторинга. Поэтому для начала она должна у вас присутствовать, и каждая компонента должна уметь отвечать на запрос о своем здоровье. Это могут быть счетчики длин внутренних очередей, загрузки процессора/памяти/дисков, количество запросов в секунду, за последнюю секунду.

Балансировка

Простым добавлением виртуальной машины (ВМ) в ваше облако производительность системы не поднять. Все ваши компоненты должны уметь автоматически развертываться на новой машине: установка пакетов, настройка и конфигурация, репликация данных – все это должно существовать в виде скриптов, запускаемых после создания новой виртуальной машины.

Кроме того, свежесозданный экземпляр сервиса должен автоматически начать получать свою долю нагрузки. Для этого можно пользоваться разными механизмами, самыми простыми из них являются различные балансировщики нагрузки и Round Robin DNS.

Кэширование

Время доступа к сетевому сервису на порядки превышает время доступа к локальным данным. Поэтому сразу после того, как вы вынесли доступ к данным в отдельную службу, вы начинаете думать о том, как кэшировать их в остальных службах, где они нужны, со всеми сопутствующими проблемами в виде дисциплины инвалидирования и обновления данных и консистентности кэшей

между экземплярами служб. Неприятно, зато кэширование позволит сохранить изолированность компонент при сохранении оперативной доступности данных.

Облачные службы

Провайдеры облачных услуг обычно предоставляют огромное количество готовых компонент, необходимых для создания масштабируемых приложений. Наиболее востребованные из них:

- облачный хостинг: виртуальные машины;
- облачное объектное хранилище: служба распределенного хранения прикладных объектов;
- облачная база данных: обеспечивает наличие реплицированной базы данных (реляционной и/или типа «ключ-значение»);
- облачный мониторинг и автомасштабирование: предоставляет услуги сбора статистики с узлов облака, мониторинга использования ресурсов и автоматического добавления/удаления сервисов;
- CDN (Content Delivery Networks): обеспечивает доставку статического контента географически и топологически распределенным пользователям системы.

НОВЫЕ – ОБЛАЧНЫЕ – НАВЫКИ ПРОГРАММИСТА

«Облачный» подход несет в себе несколько принципиально новых вещей, с которыми нужно учиться работать. Ниже перечислены несколько основополагающих советов для программистов, которые помогут адаптироваться в новых условиях жизни.

Во-первых, поскольку система становится сетевой, к ней уже нельзя относиться как к локальной. Система является распределенной, и у вас будут происходить разнообразные отказы, «странности» в сети, например, вы будете видеть внезапную блокировку/изменения файлов.

> Когда разработчик переходит на облачный формат, он начинает думать об инфраструктуре, мыслить в нетрадиционных для себя ИТ-терминах. Можно зайти с другой стороны – программировать начинают администраторы. То есть классическая граница между специалистом, разрабатывающим приложение, и сетевиком, настраивающим сеть, стирается. Приложение, которое автоматически разворачивается на виртуальных машинах, должно так же автоматически настроить себе сеть и storage. Иначе говоря, разработчику теперь нужно думать о том, на какой инфраструктуре работает его приложение. Он не может, как это было 10 лет назад, вручить дистрибутив приложения администратору со словами: «Вот требования к инфраструктуре, создай соответствующий вычислительный комплекс». Разработчик активно участвует в развертывании приложения.

Во-вторых, все, что вы делаете и чем пользуетесь в работе, должно представлять собой сервисы. Вам как программисту придется привыкать к XML, создавать каталоги сервисов и работать с ними.

В-третьих, вас ждет параллельное программирование. Вам нужно будет изучать свободные от блокировок структуры данных, учиться разрабатывать высококонкурентные модули.

Еще один принципиальный и неприятный момент связан с обработкой ошибок. Типичный исходный текст программы 1990-х годов выглядит примерно так: зовем функцию, если что-то пошло не так, пишем сообщение об ошибке. Типичная современная программа, к сожалению, не может позволить себе подобный подход, потому что отказ не является исключительной ситуацией. Вы не можете вывесить пользователю баннер, когда он пришел к вам через поисковый запрос Google, «у меня произошла ошибка на одном из тысячи серверов, попробуйте, пожалуйста, позже». Вам нужно работать с этой ситуацией: все алгоритмы, которые вы создаете, в качестве основного условия должны включать в себя вариант отказа – отсутствие доступа к серверу, к хранилищу данных, к другому сервису. Выйти из строя может все, вплоть до процесса чтения конфигураций. Иными словами приложение нужно строить таким образом, чтобы отказ рассматривался как органичная часть процесса функционирования приложения.

Кроме того, вам нужно думать о балансировке нагрузки – о том, как оптимально построить балансировщик, о метриках, о максимизации производительности, об обеспечении безопасности.

На последнем остановимся более подробно.

ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ С САМОГО НАЧАЛА

Еще один важный аспект облачного приложения – безопасность. В условиях, когда каждый компонент вашей системы – отдельный сервис, очень важно, чтобы все его аспекты, связанные с безопасностью, были продуманы на самом раннем этапе. Модели данных, роли пользователей, модель нарушителя должны быть согласованны между собой компонентами.

Не нужно забывать о готовых инструментах обеспечения безопасности – PKI, SSL, шифровании данных – зачастую уже предоставляемых используемой инфраструктурой. Пожалуй, худшее, что можно придумать в этой области, – разработка своих собственных оригинальных механизмов и алгоритмов, не подвергшихся тщательному многолетнему изучению.

ГДЕ СТОИТ ПРИМЕНЯТЬ

Принципы работы облачных вычислений целесообразнее всего использовать в 2 случаях: при разработке сервис-ориентированной архитектуры (SOA) и для решения задач Big Data.

Прежде чем рассмотреть первый из озвученных вариантов, нужно сказать несколько слов об отличии типичной, стандартной архитектуры приложения от микросервисной. Типичное приложение с точки зрения устройства представляет собой «черный ящик» для всех, кроме его автора. Допустим, программа перестает справляться с нагрузкой, при этом ее эксплуатант не может самостоятельно определить, какая из «внутренних запчастей» дала сбой и почему. Замене подлежит только все приложение целиком. Это, мягко говоря, несколько неудобно.

Ответом стало появление концепции SOA: в этом случае приложение представляет собой набор независимых сервисов. Каждый компонент программы предоставляет другим компонентам определенные услуги и одновременно выступает в роли потребителя услуг «соседей». Границы между этими сервисами явно видны даже конечным пользователям программы. Иначе говоря, вместо того чтобы возводить монолитное громоздкое здание из бетона, мы берем и соединяем отдельные подвижные блоки. Пользователь может распределять эти блоки в облачной инфраструктуре в зависимости от их потребностей в мощностях, добавлять/уменьшать вычислительные ресурсы под каждый сервис при необходимости. Приложение более прозрачно, оно легко поддается мониторингу и управлению. При этом если нужно изменить его функциональность, можно доработать отдельные сервисы, не затрагивая остальные (ими в этот момент будут продолжать пользоваться сотрудники компании). И если в случае монолитного приложения релиз-цикл подразумевает выпуск новой версии тяжелого, большого продукта не чаще одного раза в полгода–год, то модульная архитектура несет с собой непрерывность релиз-цикла: сервисы обновляются так часто, как это необходимо.

Таким образом, если говорить о «жизни» приложения в облачной инфраструктуре, то в качестве единицы размещения в облаке выступают как раз те самые сервисы, из которых оно состоит. Такой формат в первую очередь избавляет от проблем с масштабированием ресурсов, необходимых приложению.

> Технологии Big Data актуальны для тех компаний, которым необходимо обрабатывать данные о большом количестве своих клиентов/пользователей. Это телеком-операторы, финансовые организации, ритейлеры. Речь идет о работе скоринговых систем, программ лояльности, систем, прогнозирующих поведение пользователей, и т.д.

Облако также оптимально подходит как инфраструктурная основа для технологий Больших Данных. Что по своей сути представляют Big Data? На наш взгляд, самое полезное и простое определение – это данные, которые не может обработать один вычислительный узел, каким бы мощным он ни был. Типичный пример – работа поисковой машины Google. Поисковику при запросе пользователя фактически нужно сгрести в одно место весь Интернет. Над решением этой задачи трудятся сотни тысяч серверов: по ним «тонким слоем» разнесена обработка данных, после ее завершения нужная информация «сбегается» к пользователю в виде результатов поиска. Эта технология, с которой, собственно, начались Big Data. Облачные вычисления гарантируют гибкость, эластичность, необходимую для задач Больших Данных.

Jet Info
ИЗДАЕТСЯ КОМПАНИЕЙ «ИНФОСИСТЕМЫ ДЖЕТ»

Главный редактор Дмитриев В. Ю.

Россия, 127015, Москва, Б. Новодмитровская, 14/1,
тел. (495) 411 76 01, факс (495) 411 76 02, e-mail: Jetinfo@jet.msk.su, www.jetinfo.ru

Полное или частичное воспроизведение материалов, содержащихся в настоящем издании, допускается только по согласованию с издателем