## Тестирование программного обеспечения

**Тестирование программного обеспечения** — процесс исследования, испытания программного продукта, имеющий две различные цели:

* продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям;
* выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Существующие на сегодня методы тестирования программного обеспечения не позволяют однозначно и полностью выявить все дефекты и установить корректность функционирования анализируемой программы, поэтому все существующие методы тестирования действуют в рамках формального процесса проверки исследуемого или разрабатываемого программного обеспечения.

Такой процесс формальной проверки, или верификации, может доказать, что дефекты отсутствуют с точки зрения используемого метода. (То есть, нет никакой возможности точно установить или гарантировать отсутствие дефектов в программном продукте с учётом человеческого фактора, присутствующего на всех этапах жизненного цикла программного обеспечения.)

Существует множество подходов к решению задачи тестирования и верификации программного обеспечения, но эффективное тестирование сложных программных продуктов — это процесс в высшей степени творческий, не сводящийся к следованию строгим и чётким процедурам или созданию таковых.

Качество программного обеспечения можно определить как совокупную характеристику исследуемого ПО с учётом следующих составляющих:

* надёжность
* сопровождаемость
* практичность
* эффективность
* мобильность
* функциональность

Существует несколько признаков, по которым принято производить классификацию видов тестирования. Обычно выделяют следующие:

## По объекту тестирования

* Функциональное тестирование
* Тестирование производительности
* Нагрузочное тестирование
* Стресс-тестирование
* Тестирование стабильности
* Конфигурационное тестирование
* Юзабилити-тестирование
* Тестирование интерфейса пользователя
* Тестирование безопасности
* Тестирование локализации
* Тестирование совместимости

## По знанию системы

* Тестирование чёрного ящика
* Тестирование белого ящика
* Тестирование серого ящика

## По степени автоматизации

* Ручное тестирование
* Автоматизированное тестирование
* Полуавтоматизированное тестирование

## По степени изолированности компонентов

* Модульное тестирование
* Интеграционное тестирование
* Системное тестирование

## По времени проведения тестирования

* Альфа-тестирование
* Дымовое тестирование (англ. smoke testing)
* Тестирование новой функции (new feature testing)
* Подтверждающее тестирование
* Регрессионное тестирование
* Приёмочное тестирование
* Бета-тестирование

## По признаку позитивности сценариев

* Позитивное тестирование
* Негативное тестирование

## По степени подготовленности к тестированию

* Тестирование по документации (формальное тестирование)
* Интуитивное тестирование (англ. ad hoc testing)

## Уровни тестирования программного обеспечения

## Уровни тестирования

* **Модульное тестирование** — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Часто модульное тестирование осуществляется разработчиками программного обеспечения.
* **Интеграционное тестирование** — тестируются интерфейсы между компонентами, подсистемами или системами. При наличии резерва времени на данной стадии тестирование ведётся итерационно, с постепенным подключением последующих подсистем.
* **Системное тестирование** — тестируется интегрированная система на её соответствие требованиям.
* **Альфа-тестирование** — имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приёмочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться программа.
* **Бета-тестирование** — в некоторых случаях выполняется распространение предварительной версии (в случае проприетарного программного обеспечения иногда с ограничениями по функциональности или времени работы) для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Часто для свободного и открытого программного обеспечения стадия альфа-тестирования характеризует функциональное наполнение кода, а бета-тестирования — стадию исправления ошибок. При этом, как правило, на каждом этапе разработки промежуточные результаты работы доступны конечным пользователям.

## Статическое и динамическое тестирование

Описанные ниже техники — тестирование белого ящика и тестирование чёрного ящика — предполагают, что код исполняется, и разница состоит лишь в той информации, которой владеет тестировщик. В обоих случаях это динамическое тестирование.

При статическом тестировании программный код не выполняется — анализ программы происходит на основе исходного кода, который вычитывается вручную, либо анализируется специальными инструментами. В некоторых случаях анализируется не исходный, а промежуточный код (такой как байт-код или код на MSIL). Также к статическому тестированию относят тестирование требований, спецификаций, документации.

## Регрессионное тестирование

Регрессионное тестирование (англ. regression testing, от лат. regressio — движение назад) — собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода. Такие ошибки — когда после внесения изменений в программу перестаёт работать то, что должно было продолжать работать, — называют регрессионными ошибками (англ. regression bugs).

После внесения изменений в очередную версию программы, регрессионные тесты подтверждают, что сделанные изменения не повлияли на работоспособность остальной функциональности приложения. Регрессионное тестирование может выполняться как вручную, так и средствами автоматизации тестирования.

## Тестовые сценарии

Тестировщики используют тестовые сценарии на разных уровнях: как в модульном, так и в интеграционном и системном тестировании. Тестовые сценарии, как правило, пишутся для проверки компонентов, в которых наиболее высока вероятность появления отказов или вовремя не найденная ошибка может быть дорогостоящей.

## Тестирование белого ящика и черного ящика

В зависимости от доступа разработчика тестов к исходному коду тестируемой программы различают «тестирование (по стратегии) белого ящика» и «тестирование (по стратегии) чёрного ящика».  
При тестировании белого ящика (также говорят — прозрачного ящика), разработчик теста имеет доступ к исходному коду программ и может писать код, который связан с библиотеками тестируемого программного обеспечения. Это типично для модульного тестирования, при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции — работоспособны и устойчивы, до определённой степени. При тестировании белого ящика используются метрики покрытия кода или мутационное тестирование.

При тестировании чёрного ящика, тестировщик имеет доступ к программе только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов, с уверенностью в том, все ли идёт правильно, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши.

Как правило, тестирование чёрного ящика ведётся с использованием спецификаций или иных документов, описывающих требования к системе. Обычно в данном виде тестирования критерий покрытия складывается из покрытия структуры входных данных, покрытия требований и покрытия модели (в тестировании на основе моделей).

При тестировании серого ящика разработчик теста имеет доступ к исходному коду, но при непосредственном выполнении тестов доступ к коду, как правило, не требуется.

Если «альфа-» и «бета-тестирование» относятся к стадиям до выпуска продукта (а также, неявно, к объёму тестирующего сообщества и ограничениям на методы тестирования), тестирование «белого ящика» и «чёрного ящика» имеет отношение к способам, которыми тестировщик достигает цели.

Бета-тестирование в целом ограничено техникой чёрного ящика (хотя постоянная часть тестировщиков обычно продолжает тестирование белого ящика параллельно бета-тестированию). Таким образом, термин «бета-тестирование» может указывать на состояние программы (ближе к выпуску чем «альфа»), или может указывать на некоторую группу тестировщиков и процесс, выполняемый этой группой. То есть, тестировщик может продолжать работу по тестированию белого ящика, хотя программа уже «бета-стадии», но в этом случае он не является частью «бета-тестирования».

## Тестирование чёрного ящика или поведенческое тестирование

**Тестирование чёрного ящика или поведенческое тестирование** — стратегия (метод) тестирования функционального поведения объекта (программы, системы) с точки зрения внешнего мира, при котором не используется знание о внутреннем устройстве тестируемого объекта. Под стратегией понимаются систематические методы отбора и создания тестов для тестового набора. Стратегия поведенческого теста исходит из технических требований и их спецификаций.

## Тестирование по стратегии белого ящика

**Тестирование по стратегии белого ящика** — тестирование кода на предмет логики работы программы и корректности её работы с точки зрения компилятора того языка, на котором она писалась.

Тестирование по стратегии белого ящика, также называемое техникой тестирования, управляемой логикой программы, позволяет проверить внутреннюю структуру программы. Исходя из этой стратегии тестировщик получает тестовые данные путем анализа логики работы программы.

Техника Белого ящика включает в себя следующие методы тестирования:

* 1. покрытие решений;
  2. покрытие условий;
  3. покрытие решений и условий;
  4. комбинаторное покрытие условий.

# Покрытие кода тестами

Покрытие кода, по своей сути, является тестированием методом белого ящика. Тестируемое программное обеспечение собирается со специальными настройками или библиотеками или запускается в особом окружении, в результате чего для каждой используемой (выполняемой) функции программы определяется местонахождение этой функции в исходном коде. Этот процесс позволяет разработчикам и специалистам по обеспечению качества определить части системы, которые при нормальной работе используются очень редко или никогда не используются (такие как код обработки ошибок и т. п.). Это позволяет сориентировать тестировщиков на тестирование наиболее важных режимов.

Тестировщики могут использовать результаты теста покрытия кода для разработки тестов или тестовых данных, которые расширят покрытие кода на важные функции.

Как правило, инструменты и библиотеки, используемые для получения покрытия кода, требуют значительных затрат производительности и/или памяти, недопустимых при нормальном функционировании ПО. Поэтому они могут использоваться только в лабораторных условиях.

# Функциональное тестирование

Функциональное тестирование — это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности ПО в определённых условиях решать задачи, нужные пользователям. Функциональные требования определяют, что именно делает ПО, какие задачи оно решает. Функциональные требования включают в себя:

* Функциональная пригодность (англ. suitability).
* Точность (англ. accuracy).
* Способность к взаимодействию (англ. interoperability).
* Соответствие стандартам и правилам (англ. compliance).
* Защищённость (англ. security).

# Тестирование производительности

**Тестирование производительности в инженерии программного обеспечения** — тестирование, которое проводится с целью определения, как быстро работает вычислительная система или её часть под определённой нагрузкой. Также может служить для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надёжность и потребление ресурсов.

**Тестирование производительности** — это одна из сфер деятельности развивающейся в области информатики инженерии производительности, которая стремится учитывать производительность на стадии моделирования и проектирования системы, перед началом основной стадии кодирования.

В тестировании производительности различают следующие направления:

* нагрузочное (load)
* стресс (stress)
* тестирование стабильности (endurance or soak or stability)
* конфигурационное (configuration)

Возможны два подхода к тестированию производительности программного обеспечения:

* в терминах рабочей нагрузки: программное обеспечение подвергается тестированию в ситуациях, соответствующих различным сценариям использования;
* в рамках бета-тестирования, когда система испытывается реальными конечными пользователями.

# Нагрузочное тестирование

Нагрузочное тестирование — это простейшая форма тестирования производительности. Нагрузочное тестирование обычно проводится для того, чтобы оценить поведение приложения под заданной ожидаемой нагрузкой. Этой нагрузкой может быть, например, ожидаемое количество одновременно работающих пользователей приложения, совершающих заданное число транзакций за интервал времени. Такой тип тестирования обычно позволяет получить время отклика всех самых важных бизнес-транзакций. В случае наблюдения за базой данных, сервером приложений, сетью и т. д., этот тип тестирования может также идентифицировать некоторые узкие места приложения.

# Стресс-тестирование

**Стресс-тестирование (англ. Stress Testing)** — один из видов тестирования программного обеспечения, которое оценивает надёжность и устойчивость системы в условиях превышения пределов нормального функционирования.

Стресс-тестирование особенно необходимо для «критически важного» ПО, однако также используется и для остального ПО. Обычно стресс-тестирование лучше обнаруживает устойчивость, доступность и обработку исключений системой под большой нагрузкой, чем то, что считается корректным поведением в нормальных условиях.

Термин «стресс-тестирование» часто используется как синоним «нагрузочного тестирования», а также «тестирования производительности», что ошибочно, так как эти виды тестирования отвечают на разные бизнес-вопросы и используют различную методологию.

В общем случае методология стресс-тестирования основана на снятии и анализе показателей производительности приложения при нагрузках, значительно превышающих ожидаемые на стадии сопровождения, и несёт в себе цель определить выносливость или устойчивость приложения на случай всплеска активности по его использованию.

Необходимость стресс-тестирования диктуется следующими факторами:

* Большая часть всех систем разрабатываются с допущением о функционировании в нормальном режиме и даже в случае, когда допускается возможность увеличения нагрузки, реальные объёмы её увеличения не принимаются во внимание.
* В случае SLA-контракта (соглашения об уровне услуг) стоимость отказа системы в экстремальных условиях может быть очень велика.
* Обнаружение некоторых ошибок или дефектов в функционировании системы не всегда возможно с использованием других типов тестирования.
* Тестирования, проведенного разработчиком, может быть недостаточно для эмуляции условий, при которых происходит отказ системы.
* Предпочтительнее быть готовым к обработке экстремальных условий системы, чем ожидать её отказа.

Основные направления применения стресс-тестирования:

1. Общее исследование поведения системы при пиковых нагрузках.

2. Исследование обработки ошибок и исключительных ситуаций системой при пиковых нагрузках.

3. Исследование узких мест системы или отдельных компонент при диспропорциональных нагрузках.

4. Тестирование ёмкости системы.

Стресс-тестирование, как и нагрузочное тестирование, также может быть использовано для регулярной оценки изменений производительности с целью получения для дальнейшего анализа динамики изменения поведения системы за длительный период.

# Пропорциональная нагрузка

Стресс-тестирование может применяться как для обособленных приложений, так и для распределенных систем с клиент-серверной архитектурой. Простейшим примером стресс-тестирования обособленного приложения может являться открытие файла размером в 50 Мб программой Notepad, входящей в комплект ОС Windows. Условия стресс-тестирования приложения обычно формируются исходя из критических бизнес-процессов его функциональности, определенными на стадии разработки требований и анализа рисков группой, ответственной за производительность. В общем случае в качестве условий для стресс-тестирования может использоваться линейно увеличенная ожидаемая нагрузка.

# Диспропорциональная нагрузка

В случае тестирования многозвенных распределённых систем необходимо учитывать уже не только фактический объём нагрузки, состоящей из множества элементов, но и их пропорции в общем объёме.

**Например:**  
Веб-сервис предназначен для обработки и отображения созданных пользователем страниц, каждая из которых может состоять из обычного текста и динамических элементов управления. Известно, что один пользователь создает 1 страницу в день, которая содержит в среднем 1000 символов текста и одну форму. Известно также, что к системе идет 1 запрос на отображение исходной страницы в минуту. При этом скорость отображения страницы является критическим бизнес-процессом. Используя описанную выше модель поведения использующих систему пользователей, несложно смоделировать изменение динамики нагрузки при их увеличении. Стресс-тест, в котором взята за основу такая модель нагрузки не адресует рисков, связанных с тем, что система перестанет удовлетворять требованиям производительности при изменении сценария её использования. Например, скорость отображения страницы может существенно снизиться, если пользователи будут добавлять на неё десятки форм вместо одной.

Использование диспропорциональной нагрузки в стресс-тестах может также применяться для выявления узких мест отдельных компонент системы.

# Тестирование емкости

Тестирование емкости (англ. Capacity Testing) является одним из самых важных с точки зрения развития бизнеса направлений стресс-тестирования и самых сложных с точки зрения проведения и анализа. Тестирование Ёмкости проводится с целью определить запас прочности системы при полном соответствии требованиям к производительности. При моделировании нагрузки для тестирования ёмкости системы учитывается как текущая нагрузка в виде количества и пропорций одновременно поступающих в систему запросов, так и ожидаемая в перспективе.

**Например:**

Веб-сервис предназначен для отображения данных для зарегистрированных пользователей в формате .docx. Разрабатывающая компания собирается опубликовать возможность отображения данных в формате .pdf, при этом ожидается, что текущие пользователи будут продолжать работать со старым форматом, а новые будут использовать формат .pdf. Моделируя нагрузку на системы во время тестирования ёмкости, учитывается не только текущий сценарий ее использования (отображение данных в .docx), но и предполагаемый сценарий использования в будущем (часть одновременно работающих с системой пользователей используют .docx, а часть - .pdf).

Результатом тестирования ёмкости приложения или системы является набор максимально допустимых характеристик нагрузки системы, при которых приложение или система отвечает требованиям к производительности, разработанным и документированным на этапе проектирования архитектуры.

# Тестирование стабильности

Тестирование стабильности проводится с целью убедиться в том, что приложение выдерживает ожидаемую нагрузку в течение длительного времени. При проведении этого вида тестирования осуществляется наблюдение за потреблением приложением памяти, чтобы выявить потенциальные утечки. Кроме того, такое тестирование выявляет деградацию производительности, выражающуюся в снижении скорости обработки информации и/или увеличении времени ответа приложения после продолжительной работы по сравнению с началом теста.

# Конфигурационное тестирование

**Конфигурационное тестирование** — ещё один из видов традиционного тестирования производительности. В этом случае вместо того, чтобы тестировать производительность системы с точки зрения подаваемой нагрузки, тестируется эффект влияния на производительность изменений в конфигурации. Хорошим примером такого тестирования могут быть эксперименты с различными методами балансировки нагрузки. Конфигурационное тестирование также может быть совмещено с нагрузочным, стресс или тестированием стабильности.

# Цели тестирования производительности

В общих случаях тестирование производительности может служить разным целям.

* С целью демонстрации того, что система удовлетворяет критериям производительности.
* С целью определения производительность какой из двух или нескольких систем лучше.
* С целью определения, какой элемент нагрузки или часть системы приводит к снижению производительности.

Многие тесты на производительность делаются без попытки осмыслить их реальные цели. Перед началом тестирования всегда должен быть задан бизнес-вопрос: «Какую цель мы преследуем, тестируя производительность?». Ответы на этот вопрос являются частью технико-экономического обоснования (или business case) тестирования. Цели могут различаться в зависимости от технологий, используемых приложением, или его назначения, однако, они всегда включают что-то из нижеследующего:

## Параллелизм / Пропускная способность

Если конечными пользователями приложения считаются пользователи, выполняющие логин в систему в любой форме, то в этом случае крайне желательно достижение параллелизма. По определению это максимальное число параллельных работающих пользователей приложения, поддержка которого ожидается от приложения в любой момент времени. Модель поведения пользователя может значительно влиять на способность приложения к параллельной обработке запросов, особенно если он включает в себя периодически вход и выход из системы.Если концепция приложения не заключается в работе с конкретными конечными пользователями, то преследуемая цель для производительности будет основана на максимальной пропускной способности или числе транзакций в единицу времени.

## Время ответа сервера

Эта концепция строится вокруг времени ответа одного узла приложения на запрос, посланный другим. Простым примером является HTTP 'GET' запрос из браузера рабочей станции на веб-сервер. Практически все приложения, разработанные для нагрузочного тестирования, работают именно по этой схеме измерений. Иногда целесообразно ставить задачи по достижению производительности времени ответа сервера среди всех узлов приложения.

## Время отображения

Время отображения — одно из самых сложных для приложения для нагрузочного тестирования понятий, так как в общем случае они не используют концепцию работы с тем, что происходит на отдельных узлах системы, ограничиваясь только распознаванием периода времени, в течение которого нет сетевой активности. Для того, чтобы замерить время отображения, в общем случае требуется включать функциональные тестовые сценарии в тесты производительности, но большинство приложений для тестирования производительности не включают в себя такую возможность.

## Требования к производительности

Очень важно детализировать требования к производительности и документировать их в каком-либо плане тестирования производительности. В идеальном случае это делается на стадии разработки требований при разработке системы, до проработки деталей её дизайна. Однако тестирование производительности часто не проводится согласно спецификации, так как нет зафиксированного понимания о максимальном времени ответа для заданного числа пользователей. Тестирование производительности часто используется как часть процесса профайлинга производительности. Его идея заключается в том, чтобы найти «слабое звено» — такую часть системы, соптимизировав время реакции которой, можно улучшить общую производительность системы.

Тестирование производительности может проводиться с использованием глобальной сети и даже в географически удаленных местах, если учитывать тот факт, что скорость работы сети Интернет зависит от местоположения. Оно также может проводиться и локально, но в этом случае необходимо настроить сетевые маршрутизаторы таким образом, чтобы появилась задержка, присутствующая во всех публичных сетях. Нагрузка, прилагаемая к системе, должна совпадать с реальным положением дел. Так, например, если 50 % пользователей системы для доступа к системе используют сетевой канал шириной 56К, а другая половина использует оптический канал, то компьютеры, создающие тестовую нагрузку на систему, должны использовать те же соединения (идеальный вариант) или эмулировать задержки вышеуказанных сетевых соединений, следуя заданным профайлам пользователей.

## Типичные вопросы тестирования производительности

Требования к производительности должны адресовать следующие, как минимум, вопросы:

* Что охватывается тестом производительности? Какие подсистемы, компоненты, интерфейсы и т. д. должны быть протестированы?
* Если в тест включаются пользовательские интерфейсы, то, сколько одновременно работающих в системе пользователей ожидается для каждого интерфейса (необходимо определить пиковые и нормальные значения).
* Как выглядит аппаратная составляющая тестируемой системы?
* Каков сценарий использования каждого компонента системы? (например, 20 % запросов составляет вход в систему, 40 % — поиск, 30 % — выбор элемента, 10 % — выход из системы)
* Каков сценарий использования системы? [в одном тесте на производительность могут быть задействованы разные сценарии использования каждого компонента]
* Каковы требования ко времени выполнения серии операций серверной части приложения?

# Основные показатели (метрики) производительности

Одним из результатов, получаемых при нагрузочном тестировании и используемых в дальнейшем для анализа, являются показатели производительности приложения.

## 1 Потребление ресурсов центрального процессора (CPU, %)

Метрика, показывающая, сколько времени из заданного определённого интервала было потрачено процессором на вычисления для выбранного процесса. В современных системах важным фактором является способность процесса работать в нескольких потоках для того, чтобы процессор мог производить вычисления параллельно. Анализ истории потребления ресурсов процессора может объяснять влияние на общую производительность системы потоков обрабатываемых данных, конфигурации приложения и операционной системы, мультипоточности вычислений, и других факторов.

## 2 Потребление оперативной памяти (Memory usage, Mb)

Метрика, показывающая количество памяти, использованной приложением. Использованная память может делиться на три категории:

* Virtual — объём виртуального адресного пространства, которое использует процессор. Этот объём не обязательно подразумевает использование соответствующего дискового пространства или оперативной памяти. Виртуальное пространство конечно и процесс может быть ограничен в возможности загружать необходимые библиотеки.
* Private — объём адресного пространства, занятого процессом и не разделяемого с другими процессами.
* Working Set — набор страниц памяти, недавно использованных процессом.

В случае, когда свободной памяти достаточно, страницы остаются в наборе, даже если они не используются. В случае, когда, свободной памяти остается мало, использованные страницы удаляются. При работе приложения память заполняется ссылками на объекты, которые, в случае неиспользования, могут быть очищены специальным автоматическим процессом, называемым «сборщиком мусора» (англ. Garbage Collector). Время, затрачиваемое процессором на очистку памяти таким способом, может быть значительным, в случае, когда процесс занял всю доступную память (в Java — так называемый «постоянный Full GC») или когда процессу выделены большие объёмы памяти, нуждающиеся в очистке. На время, требующееся для очистки памяти, доступ процесса к страницам выделенной памяти может быть заблокирован, что может повлиять на конечное время обработки этим процессом данных.

## 3 Потребление сетевых ресурсов

Эта метрика не связана непосредственно с производительностью приложения, однако её показатели могут указывать на пределы производительности системы в целом.

Например:

Серверное приложение, обрабатывая запрос пользователя, возвращает ему видеопоток, используя сетевой канал в 2 мегабит. Требование гласит, что сервер должен обрабатывать 5 запросов пользователей одновременно.

Нагрузочное тестирование показало, что эффективно сервер может предоставлять данные только 4 пользователям одновременно, так как мультимедиа-поток имеет битрейт в 500 килобит. Очевидно, что предоставление этого потока 5 пользователям одновременно невозможно в силу превышения пропускной способности сетевого канала, а значит, система не удовлетворяет заданным требованиям производительности, хотя при этом потребление ей ресурсов процессора и памяти может быть невысоким.

## 4 Работа с дисковой подсистемой (I/O Wait)

Работа с дисковой подсистемой может значительно влиять на производительность системы, поэтому сбор статистики по работе с диском может помогать выявлять узкие места в этой области. Большое количество чтений или записей может приводить к простаиванию процессора в ожидании обработки данных с диска и в итоге увеличению потребления CPU и увеличению времени отклика.

## 5. Время выполнения запроса (request response time, ms)

Время выполнения запроса приложением остаётся одним из самых главных показателей производительности системы или приложения. Это время может быть измерено на серверной стороне, как показатель времени, которое требуется серверной части для обработки запроса; так и на клиентской, как показатель полного времени, которое требуется на сериализацию / десериализацию, пересылку и обработку запроса. Надо заметить, что не каждое приложение для тестирования производительности может измерить оба этих времени.

## Мифы тестирования производительности

Некоторые из самых распространенных мифов приведены ниже.

1. Тестирование производительности проводится с целью сломать систему. Стресс тестирование делается с целью найти критическую точку прочности системы. В других же случаях, обычное нагрузочное тестирование делается с целью исследовать поведение системы при ожидаемой нагрузке. В зависимости от других требований, может потребоваться тестирование стабильности, конфигурационное или стресс-тестирование.

2. Тестирование производительности должно осуществляться только после Интеграционного тестирования производительности. Хотя это практически норма в индустрии создании ПО, тестирование производительности может также производиться на первичной стадии разработки приложения. Такой подход называется Раннее тестирование производительности. Он помогает целостному подходу к разработке, учитывая параметры производительности и, таким образом, уменьшает не только вероятность нахождения проблемы производительности непосредственно перед релизом, но и стоимость исправления подобных проблем.

3. Тестирование производительности состоит только из написания скриптов и любое изменение в приложении приводит к небольшому рефакторингу этих скриптов. Тестирование производительности само по себе — это развивающаяся отрасль индустрии программного обеспечения. Написание скриптов, хоть и важная, но всего лишь часть тестирования производительности. Наиболее сложная задача для специалиста по тестированию — это определение необходимых к проведению тестов и анализ различных метрик производительности для выявления узких мест системы.

Другая часть мифа, касательно небольших изменений в скриптах тоже неправда, так как любые изменения в UI, особенно в сетевом протоколе, приведет к полному переписыванию скриптов с самого начала. Проблема становится более ощутимой в случае использования таких протоколов, как Web Services, Siebel, Citrix, SAP.

4. Стресс-тестирование, нагрузочное тестирование и тестирование стабильности это одно и то же. Один из самых распространенных мифов, связанный с недопониманием терминологии. Стресс-тестирование и нагрузочное тестирование — два различных вида деятельности, которая называется общим термином тестирования производительности, и решающих различные задачи. Задача стресс-тестирования — найти критическую точку прочности системы при нагрузках значительно превышающих ожидаемых или же диспропорциональных; задача нагрузочного тестирования — проверить соответствие системы требованиям при ожидаемой нагрузке.

# Нагрузочное тестирование программного обеспечения

**Нагрузочное тестирование (англ. load testing)** — подвид тестирования производительности, сбор показателей и определение производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

Для исследования времени отклика системы на высоких или пиковых нагрузках производится стресс-тестирование, при котором создаваемая на систему нагрузка превышает нормальные сценарии её использования. Не существует чёткой границы между нагрузочным и стресс-тестированием, однако эти понятия не стоит смешивать, так как эти виды тестирования отвечают на разные бизнес-вопросы и используют различную методологию.

Термин нагрузочное тестирование может быть использован в различных значениях в профессиональной среде тестирования ПО. В общем случае он означает практику моделирования ожидаемого использования приложения с помощью эмуляции работы нескольких пользователей одновременно. Таким образом, подобное тестирование больше всего подходит для многопользовательских систем, чаще — использующих клиент-серверную архитектуру (например, веб-серверов). Однако и другие типы систем ПО могут быть протестированы подобным способом. Например, текстовый или графический редактор можно заставить прочесть очень большой документ; а финансовый пакет — сгенерировать отчёт на основе данных за несколько лет. Наиболее адекватно спроектированный нагрузочный тест даёт более точные результаты.

Основная цель нагрузочного тестирования заключается в том, чтобы, создав определённую ожидаемую в системе нагрузку (например, посредством виртуальных пользователей) и, обычно, использовав идентичное программное и аппаратное обеспечение, наблюдать за показателями производительности системы.

В идеальном случае в качестве критериев успешности нагрузочного тестирования выступают требования к производительности системы, которые формулируются и документируются на стадии разработки функциональных требований к системе до начала программирования основных архитектурных решений. Однако часто бывает так, что такие требования не были четко сформулированы или не были сформулированы вовсе. В этом случае первое нагрузочное тестирование будет являться пробным (англ. exploratory load testing) и основываться на разумных предположениях об ожидаемой нагрузке и потреблении аппаратной части ресурсов.

Одним из оптимальных подходов в использовании нагрузочного тестирования для измерений производительности системы является тестирование на стадии ранней разработки. Нагрузочное тестирование на первых стадиях готовности архитектурного решения с целью определить его состоятельность называется 'proof-of-concept' тестированием.

# Основные принципы нагрузочного тестирования

Ниже рассмотрены некоторые экспериментальные факты, обобщённые в принципы, используемые при тестировании производительности в целом и применимые к любому типу тестирования производительности (в частности, и к нагрузочному тестированию).

## 1 Уникальность запросов

Даже сформировав реалистичный сценарий работы с системой на основе статистики ее использования, необходимо понимать, что всегда найдутся исключения из этого сценария.

## 2 Время отклика системы

В общем случае время отклика системы подчиняется функции нормального распределения. В частности, это означает, что, имея достаточное количество измерений, можно определить вероятность, с которой отклик системы на запрос попадёт в тот или иной интервал времени.

## 3 Зависимость времени отклика системы от степени распределённости этой системы

Дисперсия нормального распределения времени отклика системы на запрос пропорциональна отношению количества узлов системы, параллельно обрабатывающих такие запросы и количеству запросов, приходящихся на каждый узел. То есть, на разброс значений времени отклика системы влияет одновременно количество запросов, приходящихся на каждый узел системы и само количество узлов, каждый из которых добавляет некоторую случайную величину задержки при обработке запросов.

## 4 Разброс времени отклика системы

Из утверждений 1, 2 и 3 можно также заключить, что при достаточно большом количестве измерений величины времени обработки запроса в любой системе всегда найдутся запросы, время обработки которых превышает определённые в требованиях максимумы; причем, чем больше суммарное время проведения эксперимента, тем выше окажутся новые максимумы. Этот факт необходимо учитывать при формировании требований к производительности системы, а также при проведении регулярного нагрузочного тестирования.

## 5 Точность воспроизведения профилей нагрузки

Необходимая точность воспроизведения профилей нагрузки тем дороже, чем больше компонент содержит система. Часто невозможно учесть все аспекты профиля нагрузки для сложных систем, так как чем сложнее система, тем больше времени будет затрачено на проектирование, программирование и поддержку адекватного профиля нагрузки для неё, что не всегда является необходимостью. Оптимальный подход в данном случае заключается в балансировании между стоимостью разработки теста и покрытием функциональности системы, в результате которого появляются допущения о влиянии на общую производительность той или иной части тестируемой системы.

# Тестирование стабильности или надежности

Тестирование стабильности или надежности (Stability / Reliability Testing) — один из видов автоматизированного тестирования, ПО, целью которого является проверка работоспособности приложения при длительном тестировании с ожидаемым уровнем нагрузки.

Перед тем как подвергать ПО экстремальным нагрузкам стоит провести проверку стабильности в предполагаемых условиях работы, то есть погрузить продукт в полную рабочую атмосферу. При тестировании, длительность его проведения не имеет первостепенного значения, основная задача - наблюдая за потреблением ресурсов, выявить утечки памяти и проследить чтобы скорость обработки данных и/или время отклика приложения в начале теста и с течением времени не уменьшалась. В противном случае вероятны сбои в работе продукта и перезагрузки системы.  
Часто в "домашних" условиях тестирование стабильности совмещают со стресс-тестированием, то есть проверяют не только стабильность, но и способность приложения переносить жесткие условия и сильные нагрузки длительное время.

# Юзабилити-тестирование

Юзабилити-тестирование (проверка эргономичности) — исследование, выполняемое с целью определения, удобен ли некоторый искусственный объект (такой как веб-страница, пользовательский интерфейс или устройство) для его предполагаемого применения. Таким образом, проверка эргономичности измеряет эргономичность объекта или системы. Проверка эргономичности сосредоточена на определённом объекте или небольшом наборе объектов, в то время как исследования взаимодействия человек-компьютер в целом — формулируют универсальные принципы.

Проверка эргономичности — метод оценки удобства продукта в использовании, основанный на привлечении пользователей в качестве тестировщиков, испытателей и суммировании полученных от них выводов.При испытании многих продуктов пользователю предлагают в «лабораторных» условиях решить основные задачи, для выполнения которых этот продукт разрабатывался, и просят высказывать во время выполнения этих тестов свои замечания.

Процесс тестирования фиксируется в протоколе (логе) и/или на аудио-и видеоустройства — с целью последующего более детального анализа. Если проверка эргономичности выявляет какие-либо трудности (например, сложности в понимании инструкций, выполнении действий или интерпретации ответов системы), то разработчики должны доработать продукт и повторить тестирование.

Наблюдение за тем, как люди взаимодействуют с продуктом, нередко позволяет найти для него более оптимальные решения. Если при тестировании используется модератор, то его задача — держать респондента сфокусированным на задачах (но при этом не «помогать» ему решать эти задачи).

Основную трудность после проведения процедуры проверки эргономичности нередко представляют большие объёмы и беспорядочность полученных данных. Поэтому для последующего анализа важно зафиксировать:

1. Речь модератора и респодента;

2. Выражение лица респондента (снимается на видеокамеру);

3. Изображение экрана компьютера, с которым работает респондент;

4. Различные события, происходящие на компьютере, связанные с действиями пользователя:

* Перемещение курсора и нажатия на клавиши мыши;
* Использование клавиатуры;
* Переходы между экранами (браузера или другой программы).

Все эти потоки данных должны быть синхронизированы по тайм-кодам, чтобы при анализе их можно было бы соотносить между собой. Наряду с модератором в тестировании нередко участвуют наблюдатели. По мере обнаружения проблем они делают свои заметки о ходе тестирования так, чтобы после можно было синхронизировать их с основной записью. В итоге каждый значимый фрагмент записи теста оказывается, прокомментирован в заметках наблюдателя. В идеале ведущий (то есть модератор) представляет разработчика, наблюдатели — заказчика (например издателя, дистрибьютора), а испытатели — конечного пользователя (например покупателя).

Кроме вышеизложенного существует ещё один подход к проверке эргономичности: для решения задачи, предложенной пользователю, разрабатывается «идеальный» сценарий решения этой задачи. Как правило, это сценарий, на который ориентировался разработчик. При выполнении задачи пользователями регистрируются их отклонения от задуманного сценария для последующего анализа. После нескольких итераций доработки программы и последующего тестирования можно получить интерфейс, удовлетворительный с точки зрения пользователя.

# Тестирование безопасности

Тестирование безопасности — оценка уязвимости программного обеспечения к различным атакам. Компьютерные системы очень часто являются мишенью незаконного проникновения. Под проникновением понимается широкий диапазон действий: попытки хакеров проникнуть в систему из спортивного интереса, месть рассерженных служащих, взлом мошенниками для незаконной наживы. Тестирование безопасности проверяет фактическую реакцию защитных механизмов, встроенных в систему, на проникновение. В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено все:

* попытки узнать пароль с помощью внешних средств;
* атака системы с помощью специальных утилит, анализирующих защиты;
* подавление, ошеломление системы (в надежде, что она откажется обслуживать других клиентов);
* целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;
* просмотр несекретных данных в надежде найти ключ для входа в систему.

При неограниченном времени и ресурсах хорошее тестирование безопасности взломает любую систему. Задача проектировщика системы — сделать цену проникновения более высокой, чем цена получаемой в результате информации.

# Ручное тестирование

**Ручное тестирование** — часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно производится тестировщиком без использования программных средств, для проверки программы или сайта путем моделирования действий пользователя. В роли тестировщиков могут выступать и обычные пользователи, сообщая разработчикам о найденных ошибках.

# Автоматизированное тестирование программного обеспечения

**Автоматизированное тестирование программного обеспечения** — часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно использует программные средства для выполнения тестов и проверки результатов выполнения, что помогает сократить время тестирования и упростить его процесс.

# **Модульное тестирование, или юнит-тестирование**

Модульное тестирование, или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.  
Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны. Этот тип тестирования обычно выполняется программистами.

# **Поощрение изменений**

Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (регрессионное тестирование). Это поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений.

# **Упрощение интеграции**

Модульное тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируя отдельные части программы, а затем программу в целом.

# **Документирование кода**

Модульные тесты можно рассматривать как «живой документ» для тестируемого класса. Клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

## Отделение интерфейса от реализации

Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Например, класс пользуется базой данных; в ходе написания теста программист обнаруживает, что тесту приходится взаимодействовать с базой. Это ошибка, поскольку тест не должен выходить за границу класса. В результате разработчик абстрагируется от соединения с базой данных и реализует этот интерфейс, используя свой собственный mock-объект. Это приводит к менее связанному коду, минимизируя зависимости в системе.

## Сложный код

Тестирование программного обеспечения — комбинаторная задача. Например, каждое возможное значение булевской переменной потребует двух тестов: один на вариант TRUE, другой — на вариант FALSE. В результате на каждую строку исходного кода потребуется 3−5 строк тестового кода.

Как и любая технология тестирования, модульное тестирование не позволяет отловить все ошибки программы. В самом деле, это следует из практической невозможности трассировки всех возможных путей выполнения программы, за исключением простейших случаев.

# **Результат известен лишь приблизительно**

Например, в математическом моделировании. Бизнес-приложения зачастую работают с конечными и счётными множествами, научные — с континуальными. Поэтому сложно подобрать тесты для каждой из ветвей программы, сложно сказать, верен ли результат, выдерживается ли точность, и т. д. А во многих случаях качество моделирования определяется «на глаз», и последний результат записывается как «опорный». Если найдено расхождение, новый результат проверяют вручную и выясняют, какой качественнее: старый или новый.

# **Ошибки интеграции и производительности**

При выполнении юнит-тестов происходит тестирование каждого из модулей по отдельности. Это означает, что ошибки интеграции, системного уровня, функций, исполняемых в нескольких модулях, не будут определены. Кроме того, данная технология бесполезна для проведения тестов на производительность. Таким образом, модульное тестирование более эффективно при использовании в сочетании с другими методиками тестирования.

# **При общей низкой культуре программирования**

Для получения выгоды от модульного тестирования требуется строго следовать технологии тестирования на всём протяжении процесса разработки программного обеспечения. Нужно хранить не только записи обо всех проведённых тестах, но и обо всех изменениях исходного кода во всех модулях. С этой целью следует использовать систему контроля версий ПО. Таким образом, если более поздняя версия ПО не проходит тест, который был успешно пройден ранее, будет несложным сверить варианты исходного кода и устранить ошибку. Также необходимо убедиться в неизменном отслеживании и анализе неудачных тестов. Игнорирование этого требования приведёт к лавинообразному увеличению неудачных тестовых результатов.

# **Техника модульного тестирования**

Для большинства популярных языков программирования высокого уровня существуют инструменты и библиотеки модульного тестирования.   
Некоторые из них:

* **Для Java и Groovy**
  + JUnit JUnit.org
  + TestNG testNG.org
  + JavaTESK UniTESK.ru
  + Spock (написан на Groovy)
* **Для C**
  + CUnit cunit
  + CTESK UniTESK
  + cfix cfix
  + API Sanity Autotest — для динамических C/C++ библиотек в Unix-подобных ОС.
  + Unity unity — для встраиваемых приложений
  + MICRO\_UNIT MICRO\_UNIT — небольшой набор макросов с римерами использования.
* **Для Ruby**
  + Rspec
  + Test::Unit
* **Для Objective-C**
  + OCUnit
* **Для C++**
  + CxxTest
  + CPPUnit
  + Boost Test
  + Google C++ Testing Framework
  + Symbian — фреймворк для Symbian OS всех версий.
  + API Sanity Autotest — для динамических C/C++ библиотек в Unix-подобных ОС.
  + Qt Test framework - для программ, разработанных с помощью библиотеки Qt
* **Для .NET**
  + Nunit
  + XUnit.net
  + MbUnit
  + DUnit — для Delphi
  + EUnit — Erlang
* **Для Perl**
  + Test
  + Test::Simple
  + Test::More
  + Test::Unit
  + Test::Unit::Lite
* **Для PHP**
  + SimpleTest
  + PHPUnit
* **Для Python**
  + PyUnit
  + PyTest
  + Nose
  + vbUnit — Visual Basic
  + utPLSQL — PL/SQL
* **Для T-SQL**
  + TSQLUnit
  + SPUnit
* **Для ActionScript 2.0** — язык сценариев, используемый виртуальной машиной Adobe Flash Player версии 7 и 8
  + AsUnit
  + AS2Unit
* **Для ActionScript 3.0** — скриптовый язык, используемый виртуальной машиной Adobe Flash Player версии 9 и выше
  + FlexUnit
  + AsUnit
* **Для JavaScript**
  + Mocha (тестовый фреймворк)
  + Chai ("assertion library", используется совместно с тестовым framework'ом)
  + Sinon.JS (библиотека для создания mock'ов, stub'ов, spy'ев, используется совместо с тестовым framework'ом)
  + Karma runner (от создателей Angular.JS, "test runner" - организует среду выполнения тестов)
  + QUnit (от создателей jQuery)
  + JsUnit (больше не поддерживается создателями)
  + Jasmine (рекомендован создателями jsUnit)
  + D.O.H

# **Интеграционное тестирование**

Интеграционное тестирование (англ. Integration testing, иногда называется англ. Integration and Testing, аббревиатура англ. I&T) — одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию.

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования.

Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности. Тестирование этих проектируемых единиц — объединения, множества или группы модулей — выполняется через их интерфейс, с использованием тестирования «чёрного ящика».

# **Системы непрерывной интеграции**

Для автоматизации интеграционного тестирования применяются системы непрерывной интеграции (англ. Continuous Integration System, CIS).

Принцип действия таких систем состоит в следующем:

1. CIS производит мониторинг системы контроля версий;

2. При изменении исходных кодов в репозитории производится обновление локального хранилища;

3. Выполняются необходимые проверки и модульные тесты;

4. Исходные коды компилируются в готовые выполняемые модули;

5. Выполняются тесты интеграционного уровня;

6. Генерируется отчет о тестировании.

Таким образом, автоматические интеграционные тесты выполняются сразу же после внесения изменений, что позволяет обнаруживать и устранять ошибки в короткие сроки.

# **Системное тестирование программного обеспечения**

**Системное тестирование программного обеспечения** — это тестирование программного обеспечения, выполняемое на полной, интегрированной системе, с целью проверки соответствия системы исходным требованиям. Системное тестирование относится к методам тестирования чёрного ящика, и, тем самым, не требует знаний о внутреннем устройстве системы. Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований к системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи. Можно выделить два подхода к системному тестированию:

* на базе требований (requirements based) - для каждого требования пишутся тестовые случаи (test cases), проверяющие выполнение данного требования.
* на базе случаев использования (use case based) - на основе представления о способах использования продукта создаются случаи использования системы (Use Cases). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся тест кейсы (test cases), которые должны быть протестированы.

# **Тестирование на основе** **модели**

Тестирование на основе модели (англ. Model-based testing) — это тестирование программного обеспечения, в котором варианты тестирования (англ. test cases) частично или целиком получаются из модели, описывающей некоторые аспекты (чаще функциональные) тестируемой системы (англ. system under test). Модели могут отображать желаемое поведение системы или использоваться для создания тестовых стратегий или среды тестирования. Модель, описывающая тестируемый объект, как правило, абстрактная и описывает лишь часть функциональности объекта. Тесты, сгенерированные из подобных моделей, тоже абстрактны и не могут напрямую использоваться для тестирования объекта. На основе сценариев абстрактных тестов, необходимо выполнить их реализацию для тестируемого объекта. Такую реализацию можно использовать для непосредственного тестирования. В некоторых средах для моделирования, модели могут содержать достаточное   
количество информации для генерации исполняемых тестов. Поскольку модели обычно строятся на основе требований или ожидаемого поведения устройства, то такое тестирование обычно рассматривается как одна из форм тестирования по методу черного ящика.

# **Сертификация программного обеспечения**

Сертификация (лат. sertifico — удостоверяю) — подтверждение соответствия качественных характеристик товара стандартам качества. Под сертификацией подразумевается также процедура получения сертификата. Объектами сертификации являются:

* продукция;
* работы (услуги);
* системы менеджмента;
* персонал.

**Сертификация продукции** — процедура подтверждения качества, посредством которой независимая от изготовителя (продавца, исполнителя) и потребителя (покупателя) организация удостоверяет в письменной форме, что продукция соответствует установленным требованиям.

**Сертификация услуг (работ)** — это независимое подтверждение соответствия утвержденным требованиям с целью соблюдения прав потребителей.

**Сертификация систем менеджмента** — это процедура подтверждения степени соответствия и результативности определенной системы менеджмента, требованиям стандарта на данную систему менеджмента.

# **Внедрение программного обеспечения**

Внедрение программного обеспечения — процесс настройки программного обеспечения под определенные условия использования, а также обучения пользователей работе с программным продуктом. При внедрении программного обеспечения требуется действие в трех следующих плоскостях работ.

Первая из них — это выделение критических, с точки зрения общего результата, процедур в деятельности организации. Когда набор таких процедур определен, необходимо в первую очередь использовать ИТ-решение для автоматизации операций внутри именно этих процедур. Таким образом, разработанное ИТ-решение автоматически становится жизненно важным и востребованным для организации, а также будет обеспечена публичность процесса внедрения.

Вторая плоскость работ — это по своей сути расширение нормативной базы организации путем включения в нее регламентов, описывающих порядок выполнения процедур автоматизируемых процессов. В противном случае есть опасность возникновения рассогласования между автоматизированными процедурами и остальными процессами организации.

Третья — это выполнение работ по общей стандартизации существующей деятельности организации, когда выделяются лучшие практики выполнения процедур и включаются в ИТ-решение по принципу наибольшей полезности для большинства участников. Процент таких процедур относительно общего объема автоматизации может быть невелик, но это придает процессу построения решения вес в организации за счет увеличения его «полезности»