

* **Надежность**. Система должна продолжать работать корректно (осуществлять нужные функции на требуемом уровне производительности) даже при неблагоприятных обстоятельствах (в случае аппаратных или программных сбоев либо ошибок пользователя)
  1. приложение выполняет ожидаемую пользователем функцию;
  2. оно способно выдержать ошибочные действия пользователя или применение программного обеспечения неожиданным образом;
  3. его производительность достаточно высока для текущего сценария использования, при предполагаемой нагрузке и объеме данных;
  4. система предотвращает любой несанкционированный доступ и неправильную эксплуатацию.

Сбой (**fault**) и отказ (**failure**) — разные вещи. Сбой обычно определяется как отклонение одного из компонентов системы от рабочих характеристик, в то время как отказ — ситуация, когда вся система в целом прекращает предоставление требуемого сервиса пользователю

**Аппаратные сбои**

Когда речь идет о причинах отказов систем, первым делом в голову приходят аппаратные сбои. Фатальные сбои винчестеров, появление дефектов ОЗУ, отключение электропитания, отключение кем-то не того сетевого кабеля

**Первая естественная реакция на эту информацию** — повысить избыточность отдельных компонентов аппаратного обеспечения с целью снизить частоту отказов системы. Можно создать RAID-массивы из дисков, обеспечить дублирование электропитания серверов и наличие в них CPU с возможностью горячей замены, а также запастись батареями и дизельными генераторами в качестве резервных источников электропитания ЦОДов. При отказе одного компонента его место на время замены занимает резервный компонент. Такой подход не предотвращает полностью отказы, возникающие из-за проблем с оборудованием, но вполне приемлем и часто способен поддерживать бесперебойную работу машин в течение многих лет.

До недавних пор избыточность компонентов аппаратного обеспечения была достаточной для большинства приложений, делая критический отказ отдельной машины явлением вполне редким. При наличии возможности достаточно быстрого восстановления из резервной копии на новой машине время простоя в случае отказа не катастрофично для большинства приложений. Следовательно, многомашинная избыточность требовалась только небольшой части приложений, для которых была критически важна высокая доступность**. Однако по мере роста объемов данных и вычислительных запросов приложений** все больше программ начали использовать большее количество машин, что привело к пропорциональному росту частоты отказов оборудования

Поэтому **происходит сдвиг в сторону систем, способных перенести потерю целых машин**, благодаря применению методов устойчивости к сбоям вместо избыточности аппаратного обеспечения или дополнительно к ней. У подобных систем есть и эксплуатационные преимущества: система с одним сервером требует планового простоя при необходимости перезагрузки машины (например, для установки исправлений безопасности), в то время как устойчивая к аппаратным сбоям система допускает установку исправлений по узлу за раз, без вынужденного бездействия всей системы

**Программные ошибки**

Обычно считают так: аппаратные сбои носят случайный характер и независимы друг от друга: отказ диска одного компьютера не означает, что диск другого скоро тоже начнет сбоить. Конечно, возможны слабые корреляции (например, вследствие общей причины наподобие температуры в серверной стойке), но в остальных случаях одновременный отказ большого количества аппаратных компонентов маловероятен.

**Другой класс сбоев** — систематическая ошибка в системе

* 1. Программная ошибка, приводящая к фатальному сбою экземпляра сервера приложения при конкретных «плохих» входных данных. Например, возьмем секунду координации 30 июня 2012 года, вызвавшую одновременное зависание множества приложений из-за ошибки в ядре операционной системы Linux.
  2. Выходит из-под контроля процесс, полностью исчерпавший какой-либо общий ресурс: время CPU, оперативную память, пространство на диске или полосу пропускания сети.
  3. Сервис, от которого зависит работа системы, замедляется, перестает отвечать на запросы или начинает возвращать испорченные ответы.
  4. Каскадные сбои, при которых крошечный сбой в одном компоненте вызывает сбой в другом компоненте, а тот, в свою очередь, вызывает дальнейшие сбои

Быстрого решения проблемы систематических ошибок в программном обеспечении не существует. Может оказаться полезным множество мелочей, таких как:

* 1. тщательное обдумывание допущений и взаимодействий внутри системы;
  2. всестороннее тестирование;
  3. изоляция процессов;
  4. предоставление процессам возможности перезапуска после фатального сбоя;
  5. оценка, мониторинг и анализ поведения системы при промышленной эксплуатации. Если система должна обеспечивать выполнение какого-либо условия (например, в очереди сообщений количество входящих сообщений должно быть равно количеству исходящих), то можно организовать постоянную самопроверку во время работы и выдачу предупреждения в случае обнаружения расхождения

**Человеческий фактор**

Проектируют и создают программные системы люди; ими же являются и операторы, обеспечивающие их функционирование. Даже при самых благих намерениях люди ненадежны. Например, одно исследование крупных интернет-сервисов показало, что основной причиной перебоев в работе были допущенные операторами ошибки в конфигурации, в то время как сбои аппаратного обеспечения (серверов или сети) играли какую-либо роль лишь в 10–25 % случаев

**На сколько важна надежность**

Надежность нужна не только в управляющем программном обеспечении атомных электростанций и воздушного сообщения — от обычных приложений тоже ожидается надежная работа. Ошибки в коммерческих приложениях приводят к потерям производительности (и юридическим рискам, если цифры в отчетах оказались неточны), а простои сайтов интернет-магазинов могут приводить к колоссальным убыткам в виде недополученных доходов и ущерба для репутации

* **Масштабируемость**. Должны быть предусмотрены разумные способы решения возникающих при росте (в смысле объемов данных, трафика или сложности) системы проблем. **Приложение масштабируемо, если** **оно способно выделять больше ресурсов и обслуживать больше запросов, без надобности изменять что то в коде. Так же выделение ресурсов должно быть своевременным (если выделились ресурсы спустя 2 часа после пиковой нагрузки, то мы потеряем клиентов)**

**Описание нагрузки**

Во-первых, нужно сжато описать текущую нагрузку на систему: только тогда мы сможем обсуждать вопросы ее роста («Что произойдет, если удвоить нагрузку?»). Нагрузку можно описать с помощью нескольких чисел, которые мы будем называть параметрами нагрузки. Оптимальный выбор таких параметров зависит от архитектуры системы. Это может быть количество запросов к веб-серверу в секунду, отношение количества операций чтения к количеству операций записи в базе данных, количество активных одновременно пользователей в комнате чата, частота успешных обращений в кэш или что-то еще. Возможно, для вас будет важно среднее значение, а может, узкое место в вашей ситуации будет определяться небольшим количеством предельных случаев.

Чтобы прояснить эту идею, рассмотрим в качестве примера социальную сеть Twitter, задействуя данные, опубликованные в ноябре 2012 года. Две основные операции сети Twitter таковы:

* 1. публикация твита — пользователь может опубликовать новое сообщение для своих подписчиков (в среднем 4600 з/с, пиковое значение — более 12 000 з/с);
  2. домашняя лента — пользователь может просматривать твиты, опубликованные теми, на кого он подписан (300 000 з/с).

Просто обработать 12 тысяч записей в секунду (пиковая частота публикации твитов) должно быть несложно. Однако проблема с масштабированием сети Twitter состоит не в количестве твитов, а в коэффициенте разветвления по выходу — каждый пользователь подписан на множество людей и, в свою очередь, имеет много подписчиков. Существует в общих чертах два способа реализации этих двух операций

1. Публикация твита просто приводит к вставке нового твита в общий набор записей. Когда пользователь отправляет запрос к своей домашней ленте, выполняется поиск всех людей, на которых он подписан, поиск всех твитов для каждого из них и их слияние (с сортировкой по времени). В реляционной базе данных, это можно выполнить путем следующего запроса:

**SELECT** tweets.\*, users.\* **FROM** tweets

**JOIN** users **ON** tweets.sender\_id = users.id

**JOIN** follows **ON** follows.followee\_id = users.id

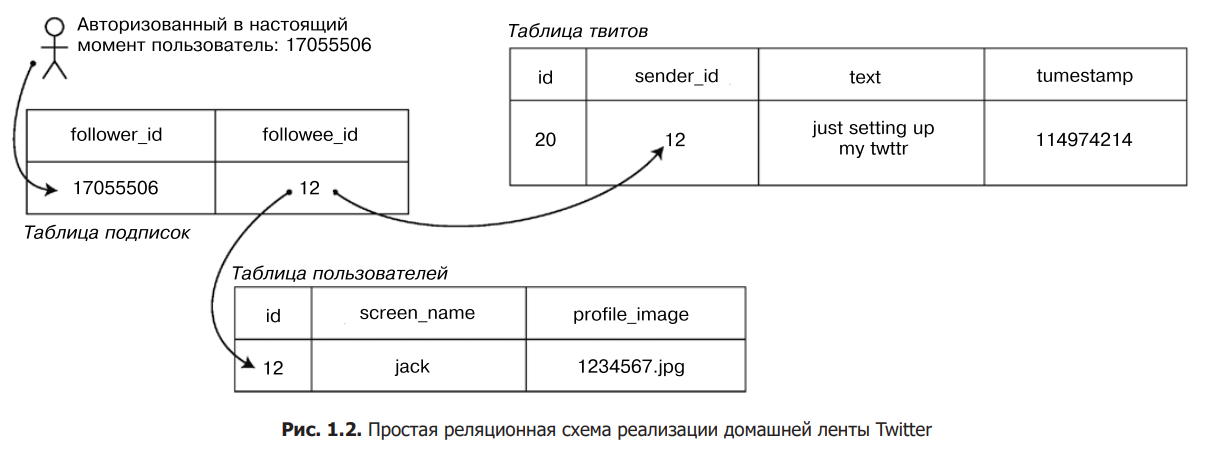
**WHERE** follows.follower\_id = current\_user

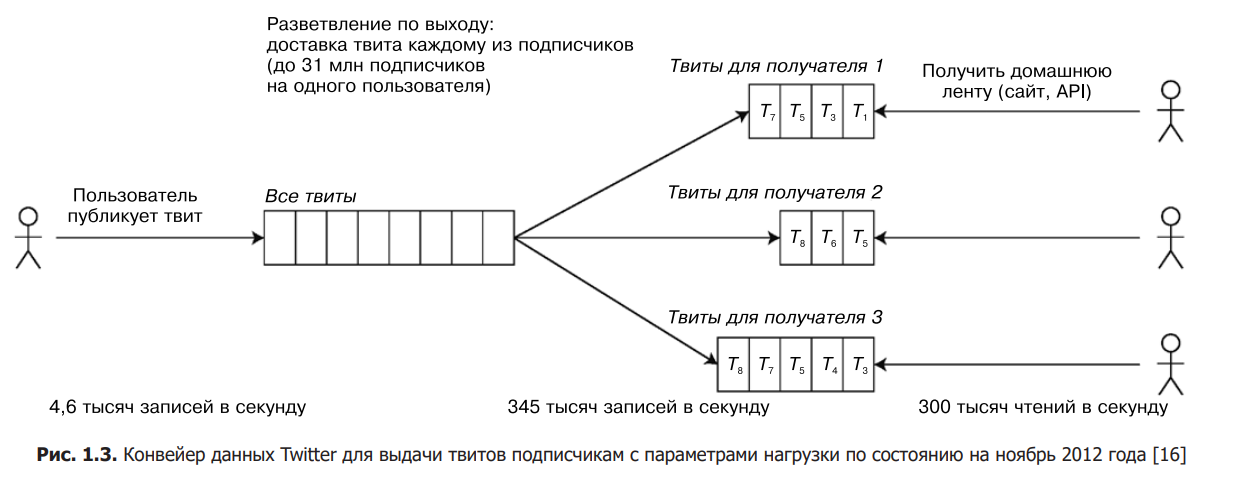
1. Поддержка кэша для домашней ленты каждого пользователя — аналог почтового ящика твитов для каждого получателя. Когда пользователь публикует твит, выполняется поиск всех его подписчиков и вставка нового твита во все кэши их домашних лент. Запрос на чтение домашней ленты при этом становится малозатратным, поскольку его результат уже был заранее вычислен.

Первая версия социальной сети Twitter использовала подход 1, но система еле справлялась с нагрузкой от запросов домашних лент, вследствие чего **компания переключилась на подход 2**. Этот вариант работал лучше, поскольку средняя частота публикуемых твитов почти на два порядка ниже, чем частота чтения домашних лент, так что в данном случае предпочтительнее выполнять больше операций во время записи, а не во время чтения

Однако недостатком подхода 2 является **необходимость в значительном количестве дополнительных действий для публикации твита**. В среднем твит выдается 75 подписчикам, поэтому 4,6 тысяч твитов в секунду означает **345 тысяч записей** в секунду в кэши домашних лент. Но приведенное здесь среднее значение маскирует тот факт, что количество подписчиков на пользователя сильно варьируется, и у некоторых пользователей насчитывается более 30 миллионов подписчиков. То есть **один твит может привести к более чем 30 миллионов записей в домашние ленты**!

1. И последний поворот истории Twitter: реализовав ошибкоустойчивый подход 2, сеть понемногу начинает двигаться в сторону объединения обоих подходов. Большинство записей пользователей по-прежнему распространяются по домашним лентам в момент их публикации, но **некое количество пользователей с очень большим количеством подписчиков** (то есть знаменитости) исключены из этого процесса. Твиты от знаменитостей, на которых подписан пользователь, выбираются отдельно и сливаются с его домашней лентой при ее чтении, подобно подходу 1. Такая гибридность обеспечивает одинаково хорошую производительность во всех случаях





**Описание производительности**

После описания нагрузки на систему, можно выяснить, что произойдет при ее возрастании. Следует обратить внимание на два аспекта.

* Как изменится производительность системы, если увеличить параметр нагрузки при неизменных ресурсах системы (CPU, оперативная память, пропускная способность сети и т. д.)?
* Насколько нужно увеличить ресурсы при увеличении параметра нагрузки, чтобы производительность системы не изменилась?

Термины «время ожидания» (latency) и «время отклика» (response time) часто используются как синонимы, хотя это не одно и то же.

* Время отклика — то, что видит клиент: помимо фактического времени обработки запроса (время обслуживания, service time), оно включает задержки при передаче информации по сети и задержки сообщений в очереди.
* Время ожидания — длительность ожидания запросом обработки, то есть время, на протяжении которого он ожидает обслуживания

Довольно часто смотрят именно на среднее время отклика. (Строго говоря, термин «среднее» не подразумевает значения, вычисленного по какой-либо конкретной формуле, но на практике под ним обычно понимается арифметическое среднее: при n значениях сложить их все и разделить на n.) Однако среднее значение далеко не лучшая метрика для случаев, когда нужно знать «типичное» время отклика, поскольку оно ничего не говорит о том, у какого количества пользователей фактически была такая задержка. Обычно удобнее применять **процентили**.

Если отсортировать список времен отклика по возрастанию, то **медиана** — средняя точка: например, медианное время отклика, равное 200 мс, означает, что ответы на половину запросов возвращаются менее чем через 200 мс, а половина запросов занимает более длительное время.

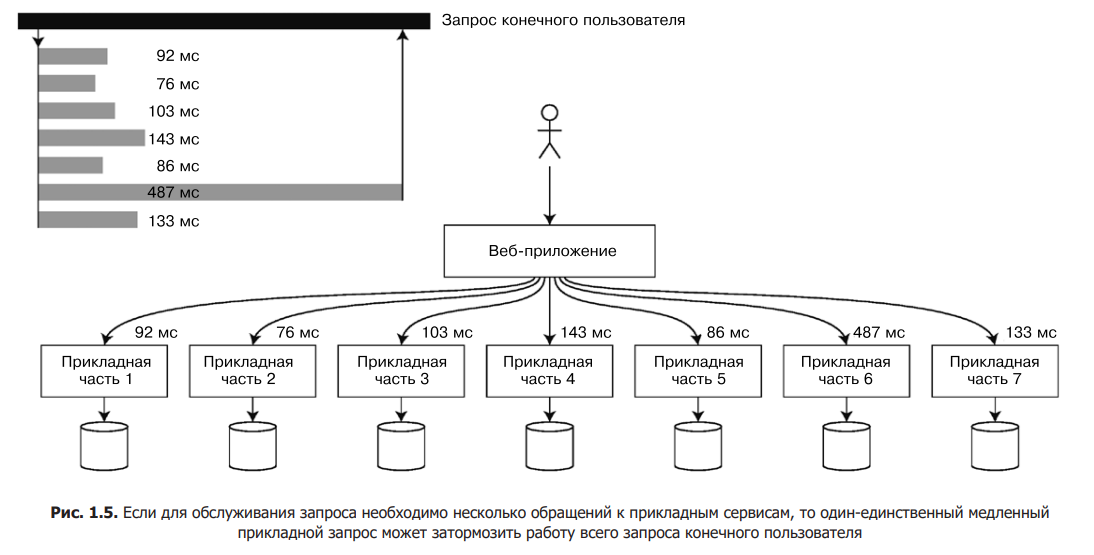
Это делает медиану отличной метрикой, когда нужно узнать, сколько пользователям обычно приходится ждать: половина запросов пользователя обслуживается за время отклика меньше медианного, а оставшиеся обслуживаются более длительное время. **Медиана также называется 50-м процентилем**

Чтобы выяснить, насколько плохи аномальные значения, можно обратить внимание на более высокие процентили: часто применяются 95-й, 99-й и 99.9-й. Это пороговые значения времени отклика, для которых 95 %, 99 % или 99,9 % запросов выполняются быстрее соответствующего порогового значения времени. Например, то, что время отклика для 95-го процентиля равно 1,5 с, означает следующее: 95 из 100 запросов занимают менее 1,5 с, а 5 из 100 занимают 1,5 с либо дольше

**Верхние процентили времени отклика**, известные также под названием «**хвостовых**» **времен** **ожидания**, важны потому, что непосредственно оказывают влияние на опыт взаимодействия пользователя с сервисом. Например, Amazon описывает требования к времени отклика для внутренних сервисов в терминах 99.9х процентилей, хотя это касается лишь 1 из 1000 запросов. Дело в том, что клиенты с самыми медленными запросами зачастую именно те, у кого больше всего данных в учетных записях, поскольку они сделали много покупок, — то есть они являются самыми ценными клиентами

С другой стороны, оптимизация по 99.99-му процентилю (самому медленному из 10 000 запросов) считается слишком дорогостоящей и не приносящей достаточно выгоды с точки зрения целей Amazon. Снижать время отклика на очень высоких процентилях — непростая задача, поскольку на них могут оказывать влияние по независящим от вас причинам различные случайные события, а выгоды от этого минимальны.

**За значительную часть времени отклика на верхних процентилях часто несут ответственность задержки сообщений в очереди**. Так как сервер может обрабатывать параллельно лишь небольшое количество заданий (ограниченное, например, количеством ядер процессора), даже небольшого количества медленных запросов достаточно для задержки последующих запросов — явление, иногда называемое блокировкой головы очереди. Даже если последующие запросы обрабатываются сервером быстро, клиентское приложение все равно будет наблюдать низкое общее время отклика из-за времени ожидания завершения предыдущего запроса. Принимая во внимание это явление, важно измерять время отклика на стороне клиента



**Как справиться с нагрузкой**

Зачастую проводят дифференциацию между **вертикальным масштабированием** — переходом на более мощную машину — и **горизонтальным масштабированием** — распределением нагрузки по нескольким меньшим машинам

На практике хорошая архитектура обычно представляет собой прагматичную смесь этих подходов: например, может оказаться проще и дешевле использовать несколько весьма мощных компьютеров, чем много маленьких виртуальных машин

Хотя распределение сервисов без сохранения состояния по нескольким машинам особых сложностей не представляет, преобразование информационных систем с сохранением состояния из одноузловых в распределенные может повлечь значительные сложности. Поэтому **до недавнего времени считалось разумным держать базу данных на одном узле (вертикальное масштабирование)** до тех пор, пока стоимость масштабирования или требования по высокой доступности не заставят сделать ее распределенной.

**Архитектура крупномасштабных систем обычно очень сильно зависит от приложения** — не существует такой вещи, как одна масштабируемая архитектура на все случаи жизни (на сленге именуемая волшебным масштабирующим соусом). Проблема может заключаться в количестве чтений, количестве записей, объеме хранимых данных, их сложности, требованиях к времени отклика, паттернах доступа или (зачастую) какой-либо смеси всего перечисленного, а также во многом другом. **Например, система, рассчитанная на обработку 100 000 з/с по 1 Кбайт каждый, выглядит совсем не так, как система, рассчитанная на обработку 3 з/мин. по 2 Гбайт каждый — хотя пропускная способность обеих систем в смысле объема данных одинакова**. Хорошая масштабируемая для конкретного приложения архитектура базируется на допущениях о том, какие операции будут выполняться часто, а какие — редко, то есть на параметрах нагрузки. Если эти допущения окажутся неверными, то работа архитекторов по масштабированию окажется в лучшем случае напрасной, а в худшем — приведет к обратным результатам. На ранних стадиях обычно важнее быстрая работа существующих возможностей в опытном образце системы или непроверенном программном продукте, чем его масштабируемость под гипотетическую будущую нагрузку. Несмотря на зависимость от конкретного приложения, масштабируемые архитектуры обычно создаются на основе универсальных блоков, организованных по хорошо известным паттернам.

* **Удобство** **сопровождения**. Необходимо обеспечить возможность эффективной работы с системой множеству различных людей (разработчикам и обслуживающему персоналу, занимающимся как поддержкой текущего функционирования, так и адаптацией системы к новым сценариям применения)
* **Удобство эксплуатации**. Облегчает обслуживающему персоналу поддержание беспрепятственной работы системы.
* **Простота**. Облегчает понимание системы новыми инженерами путем максимально возможного ее упрощения. (Обратите внимание: это не то же самое, что простота пользовательского интерфейса.)
* **Возможность развития**. Упрощает разработчикам внесение в будущем изменений в систему, адаптацию ее для непредвиденных сценариев использования при смене требований. Известна под названиями «расширяемость» (extensibility), «модифицируемость» (modifiability) и «пластичность» (plasticity).

**Удобство эксплуатации**

Хорошая команда операторов обычно отвечает за пункты, перечисленные ниже, и не только

* мониторинг состояния системы и восстановление сервиса в случае его ухудшения;
* выяснение причин проблем, например, отказов системы или снижения производительности;
* поддержание актуальности программного обеспечения и платформ, включая исправления безопасности;
* отслеживание влияния различных систем друг на друга во избежание проблемных изменений до того, как они нанесут ущерб;
* предупреждение и решение возможных проблем до их возникновения (например, планирование производительности);
* введение в эксплуатацию рекомендуемых практик и инструментов для развертывания, управления конфигурацией и т. п.;
* выполнение сложных работ по сопровождению, например переноса приложения с одной платформы на другую;
* поддержание безопасности системы при изменениях в конфигурации;
* формирование процессов, которые бы обеспечили прогнозируемость операций и стабильность операционной среды;
* сохранение знаний организации о системе, несмотря на уход старых сотрудников и приход новых

Удобство эксплуатации означает облегчение выполнения стандартных задач, благодаря чему обслуживающий персонал может сосредоточить усилия на чем-то более важном. Информационные системы способны делать многое для облегчения выполнения стандартных задач, в том числе:

* обеспечивают хороший мониторинг и предоставляют информацию о поведении системы и происходящем внутри нее во время работы;
* обеспечивают хорошую поддержку автоматизации и интеграции со стандартными утилитами;
* позволяют не зависеть от отдельных машин (благодаря чему можно отключать некоторые из них для технического обслуживания при сохранении бесперебойной работы системы в целом);
* предоставляют качественную документацию и понятную операционную модель («если я выполню действие X, то в результате произойдет действие Y»);
* обеспечивают разумное поведение по умолчанию, но вместе с тем и возможности для администраторов при необходимости переопределять настройки по умолчанию;
* запускают самовосстановление по мере возможности, но вместе с тем и позволяют администраторам вручную управлять состоянием системы при необходимости;
* демонстрируют предсказуемое поведение, минимизируя неожиданности.

**Простота – регулируем сложность**

Код небольших программных проектов может быть восхитительно простым и выразительным, но по мере роста проекта способен стать очень сложным и трудным для понимания. Подобная сложность замедляет работу над системой, еще более увеличивая стоимость сопровождения. Увязнувший в сложности проект иногда описывают как большой ком грязи

Существуют различные возможные симптомы излишней сложности: скачкообразный рост пространства состояний, тесное сцепление модулей, запутанные зависимости, несогласованные наименования и терминология, «костыли» для решения проблем с производительностью, выделение частных случаев для обхода проблем и многое другое

**Один из лучших инструментов для исключения побочной сложности — абстракция**. Хорошая абстракция позволяет скрыть бо'льшую часть подробностей реализации за аккуратным и понятным фасадом. Хорошую абстракцию можно также задействовать для широкого диапазона различных приложений. Такое многократное применение не только эффективнее реализации заново одного и тоже же несколько раз, но и приводит к более качественному ПО, по мере усовершенствования используемого всеми приложениями абстрактного компонента.

**Например, высокоуровневые языки программирования — абстракции**, скрывающие машинный код, регистры CPU и системные вызовы. SQL — тоже абстракция, скрывающая сложные структуры данных, находящиеся на диске и в оперативной памяти, конкурентные запросы от других клиентов и возникающие после фатальных сбоев несогласованности. Конечно, при создании продукта с помощью языка программирования высокого уровня мы по-прежнему используем машинный код, просто не задействуем его напрямую, поскольку абстракция языка программирования позволяет не задумываться об этом.

Есть хорошие абстракции, позволяющие выделить части большой системы в четко очерченные компоненты, допускающие повторное использование.

**Возможность развития – облегчаем внесение изменений**

Вероятность того, что ваши системные требования навсегда останутся неизменными, стремится к нулю. Гораздо вероятнее их постоянное преобразование: будут открываться новые факты, возникать непредвиденные сценарии применения, меняться коммерческие приоритеты, пользователи будут требовать новые возможности, новые платформы — заменять старые, станут меняться правовые и нормативные требования, рост системы потребует архитектурных изменений и т. д.

В терминах организационных процессов рабочие паттерны Agile обеспечивают инфраструктуру адаптации к изменениям. Сообщество создателей Agile разработало также технические инструменты и паттерны, полезные при проектировании программного обеспечения в часто меняющейся среде, например при разработке через тестирование (test-driven development, TDD) и рефакторинге.

Большинство обсуждений этих методов Agile ограничивается довольно небольшими, локальными масштабами (пара файлов исходного кода в одном приложении). В книге мы займемся поиском способов ускорения адаптации на уровне больших информационных систем, вероятно состоящих из нескольких различных приложений или сервисов с различными характеристиками.

* **Резюме**

Чтобы приносить пользу, приложение должно соответствовать различным требованиям. Выделяют **функциональные требования** (что приложение должно делать, например обеспечивать возможность хранения, извлечения, поиска и обработки информации различными способами) и некоторые **нефункциональные требования** (такие общие характеристики, как безопасность, надежность, соответствие нормативным документам, масштабируемость, совместимость и удобство сопровождения).

* **Надежность** означает обеспечение правильной работы системы даже в случае сбоев. Они могут происходить в аппаратном обеспечении (обычно случайные и невзаимосвязанные), ПО (ошибки обычно носят системный характер и слабо контролируемы) и привноситься операторами (неминуемо допускающими ошибки время от времени). Методы обеспечения устойчивости к сбоям позволяют скрывать некоторые виды сбоев от конечного пользователя
* **Масштабируемость** означает наличие стратегий поддержания хорошей производительности даже в случае роста нагрузки. Для обсуждения масштабируемости необходимы прежде всего способы количественного описания нагрузки и производительности. Мы вкратце рассмотрели домашние ленты социальной сети Twitter в качестве примера описания нагрузки и процентили времени отклика как способ оценки производительности. В масштабируемой системе есть возможность добавлять вычислительные мощности в целях сохранения надежной работы системы под высокой нагрузкой.
* **Удобство сопровождения** многолико, но, по существу, подразумевается облегчение жизни командам разработчиков и операторов, работающим с системой. Помочь снизить сложность и упростить модификацию системы и ее адаптацию к новым сценариям использования могут хорошие абстракции. Удобство эксплуатации означает хороший мониторинг состояния системы и эффективные способы управления им.