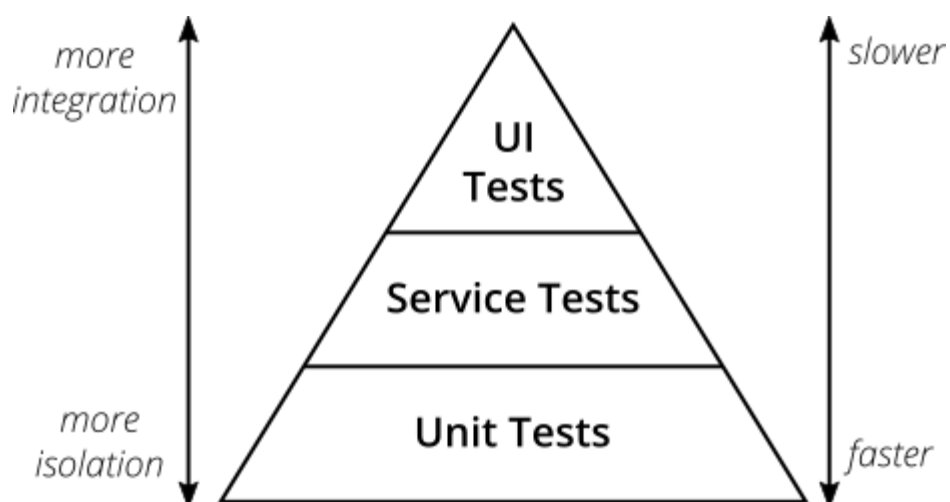


## Пирамида тестирования



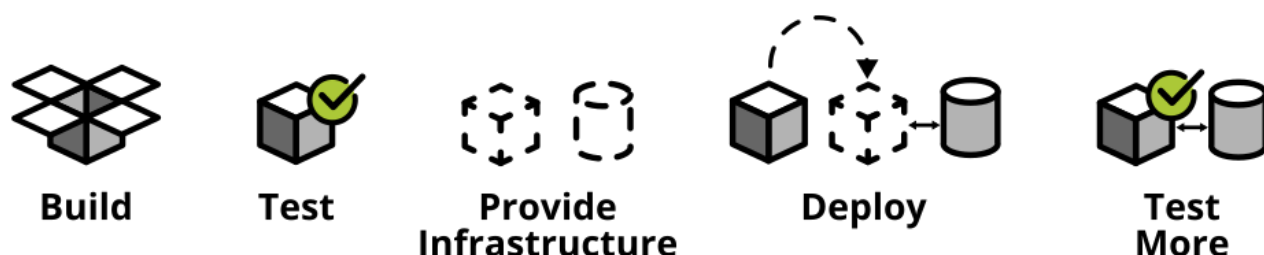
«Пирамида тестов» — метафора, она означает что тесты программного обеспечения можно сгруппировать по разным уровням детализации. Также она даёт представление, сколько тестов должно быть в каждой из этих групп или проще говоря процент тестов на каждый уровень.

Сегодня мы рассмотрим первоначальную концепцию тестовой пирамиды и также взгляды с нескольких источников

Перед тем как выпустить программное обеспечение его нужно тестировать. Поэтому было создано множество вариантов как улучшить процес тестирования, сэкономить время и деньги и удовлетворить потребности и требования к ПО.

Теперь ище представим что сегодня каждая компания стремится стать первоклассной цифровой компанией. Мы каждый день выступаем пользователями всё большего количества ПО. Скорость инноваций возрастает.

Если хотите идти в ногу со временем, нужно искать более быстрые способы доставки ПО, не жертвуя его качеством. В этом может помочь непрерывная доставка — это практика, которая автоматически гарантирует, что ПО может быть выпущено в продакшн в любое время. Поэтому очень часто когда возможно используется автоматизация тестирования. В этом случае часто используется пирамида тестирования.



Представим ситуацию: ребенок хочет робота лего или куклу

Конечному пользователю ( то есть ребенку) чаще всего важен конечный продукт а не процесс.

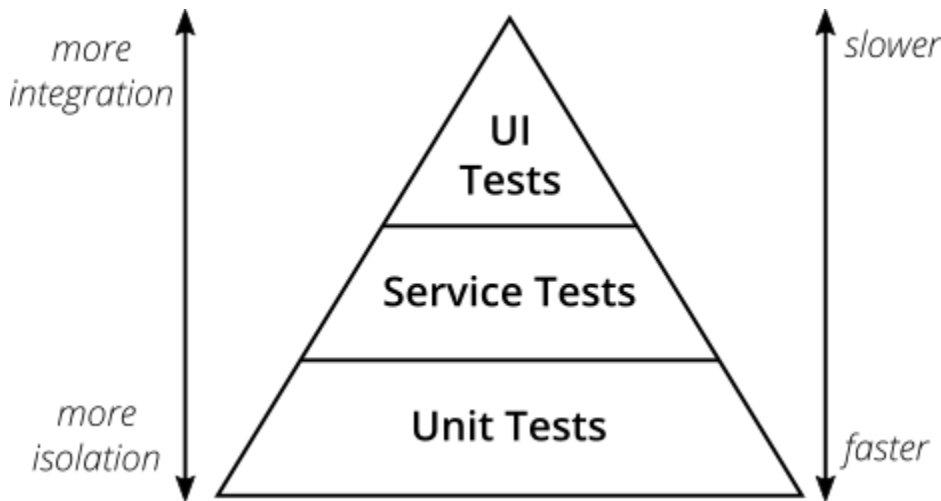
Но до того как робот лего попадет в руки он должен пройти тестирование. Сначала каждая часть отдельно, каждый кусочек лего должен пройти проверку на прочность. Ведь легче вначале найти непригодный кусочек нежели потом исправлять всю игрушку. Это впервую очередь сэкономит деньги и время. Дальше нужно проверить как эти кусочки можно сложить вместе – руки, ноги – будут ли держаться части вместе

И уже только вконец будет проверяться вся игрушка вместе – будет ли двигаться робот, засветятся ли глаза и т.д.

Так же и с тестированием ПО

Что ж это за зверь такой эта пирамида тестирования ПО?

Один из вариантов Пирамиды тестов представил Майк Кон в своей книге «Scrum: гибкая разработка ПО» (Succeeding With Agile. Software Development Using Scrum). Это отличная визуальная метафора, натапливающая на мысль о разных уровнях тестов. Она также показывает объём тестов на каждом уровне.

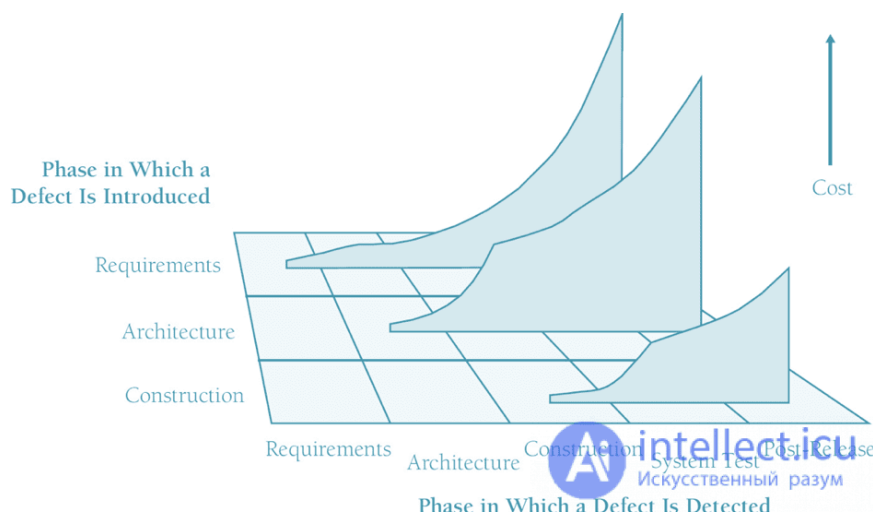


Оригинальная пирамида тестов Майка Кона состоит из трёх уровней (снизу вверх):

1. Юнит-тесты.
2. Сервисные тесты.
3. Тесты пользовательского интерфейса.

Тест, каким бы он ни был, не имеет другой цели, кроме как дать вам обратную связь: «Моя программа делает то, что должна?»

чем раньше вы узнаете, что тест не пройден, тем быстрее вы сможете определить проблему и дешевле будет починить



Поэтому хорошая стратегия тестирования направлена на максимальное увеличение количества тестов, соответствующих трем критериям: точность, скорость и надежность.

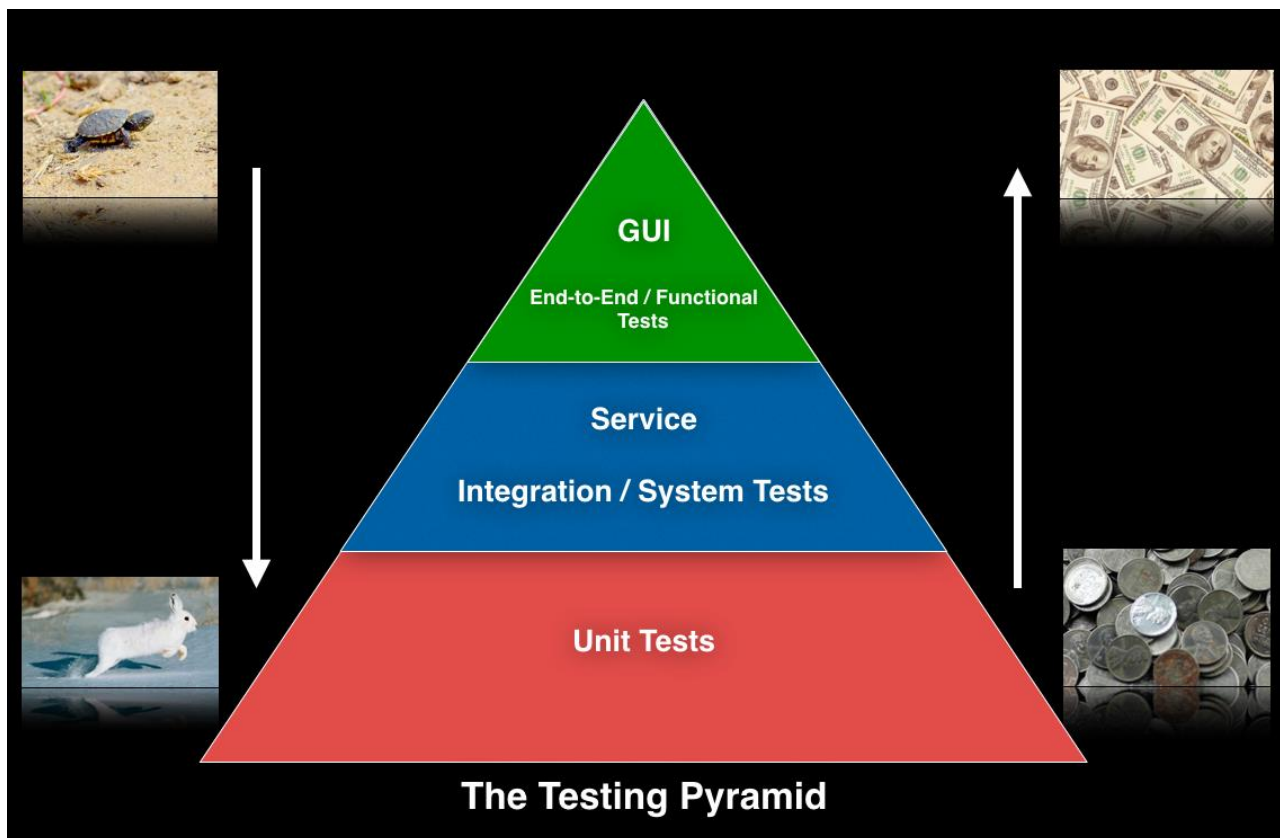
Тем не менее, из-за своей простоты суть тестовой пирамиды представляет хорошее эмпирическое правило, когда дело доходит до создания собственного набора тестов. Из этой пирамиды главное запомнить два принципа:

1. Писать тесты разной детализации.
2. Чем выше уровень, тем меньше тестов.

Напишите много маленьких и быстрых юнит-тестов. Напишите несколько более общих тестов и совсем мало высокоуровневых сквозных тестов, которые проверяют приложение от начала до конца. Следите, что у вас в итоге не получился [тестовый рожок мороженого](#), который станет кошмаром в поддержке и будет слишком долго выполняться.

К сожалению, при более тщательном осмотре концепция кажется недостаточной. Некоторые утверждают, что либо именования, либо некоторые концептуальные аспекты пирамиды тестов Майка Кона не идеальны. Поэтому вы можете встретить и другие варианты пирамиды тестирования.

И не привязывайтесь слишком сильно к названиям отдельных уровней пирамиды тестов. Учитывая недостатки оригинальных названий в пирамиде, вполне нормально придумать другие имена для своих уровней тестов. Главное, чтобы они соответствовали вашему коду и терминологии, принятой в вашей команде.



Перед вами все еще пирамида Кона.

**Давайте рассмотрим ище один критерий для использования пирамиды:**

### **Простота создания и обслуживания затрат**

Зная, что у вас вряд ли будет неограниченный бюджет, ваша стратегия тестирования будет обязательно зависеть от него. Вам необходимо сопоставить стоимость различных типов тестов со стоимостью, которую они предоставляют

- Модульные тесты, как правило, очень просты в реализации, при условии, что вы не ждете, пока код не будет поврежден техническими проблемами.

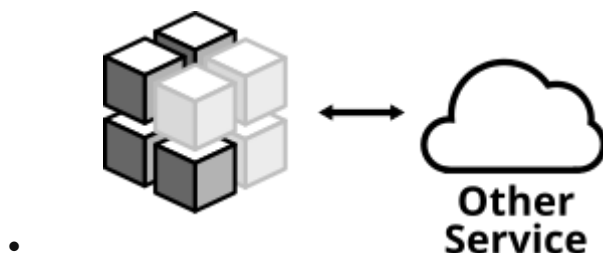
**Модульные тесты** должны составлять основную часть автоматизированного тестирования.

- Тесты являются детальными и могут помочь точно определить дефект;
- Время выполнения невероятно быстрое, потому что им не нужно полагаться на какой-либо пользовательский интерфейс или внешние системы, такие как база данных или API;
- Они недорогие, просто пишутся, легко поддерживать.

- Интеграционные тесты также довольно просты (здесь речь идет об интеграционных тестах внутри приложения, а не между приложениями).

Они охватывают более широкий спектр кода и, как следствие, более вероятно, будут затронуты изменения в коде приложения.

- Сложнее в разработке и дольше в исполнении более высокоуровневые тесты – интеграционные тесты, которые проверяют корректность работы одновременно работающих модулей, над которыми трудилась вся команда, выпуская свой продукт (систему). То есть проверяется интеграция кода, тестируется система без учета взаимодействия со внешними системами. Такие тесты уже подразумевают проверку высокоуровневую, скорее всего через обращение к системе через системное API или даже GUI (фронт). **Интеграционные тесты** должны занимать середину пирамиды. Используйте этот уровень для проверки бизнес-логики без использования пользовательского интерфейса (UI); Тестируя за пределами пользовательского интерфейса, вы можете тестировать входы и выходы API или сервисов без всех сложностей, которые вводит пользовательский интерфейс; Эти тесты медленнее и сложнее, чем модульные тесты, потому что им может потребоваться доступ к базе данных или другим компонентам.



*Этот вид интеграционного теста проверяет, что приложение способно правильно взаимодействовать с отдельными службами*




- Сквозное тестирование или тестирование GUI является более сложным для реализации, поскольку требует развертывания полной среды. Наличие зависимостей и наборов данных образуют загадку, которая усложняется нестабильностью из-за перезагрузок среды, изменений другими группами и т. Д. Эти сложности делают этот тип теста очень хрупким и приводят к затратам на обслуживание в виде анализа ошибок сборки и обслуживания среда и данные.

- **Тесты пользовательского интерфейса** должны размещаться на вершине пирамиды. Большая часть вашего кода и бизнес-логики должна быть уже протестирована до этого уровня; Тесты интерфейса пишутся, чтобы убедиться, что сам интерфейс работает правильно; Тесты пользовательского интерфейса медленнее и тяжелее в написании и поддержке, поэтому необходимо сводить их к минимуму.

•

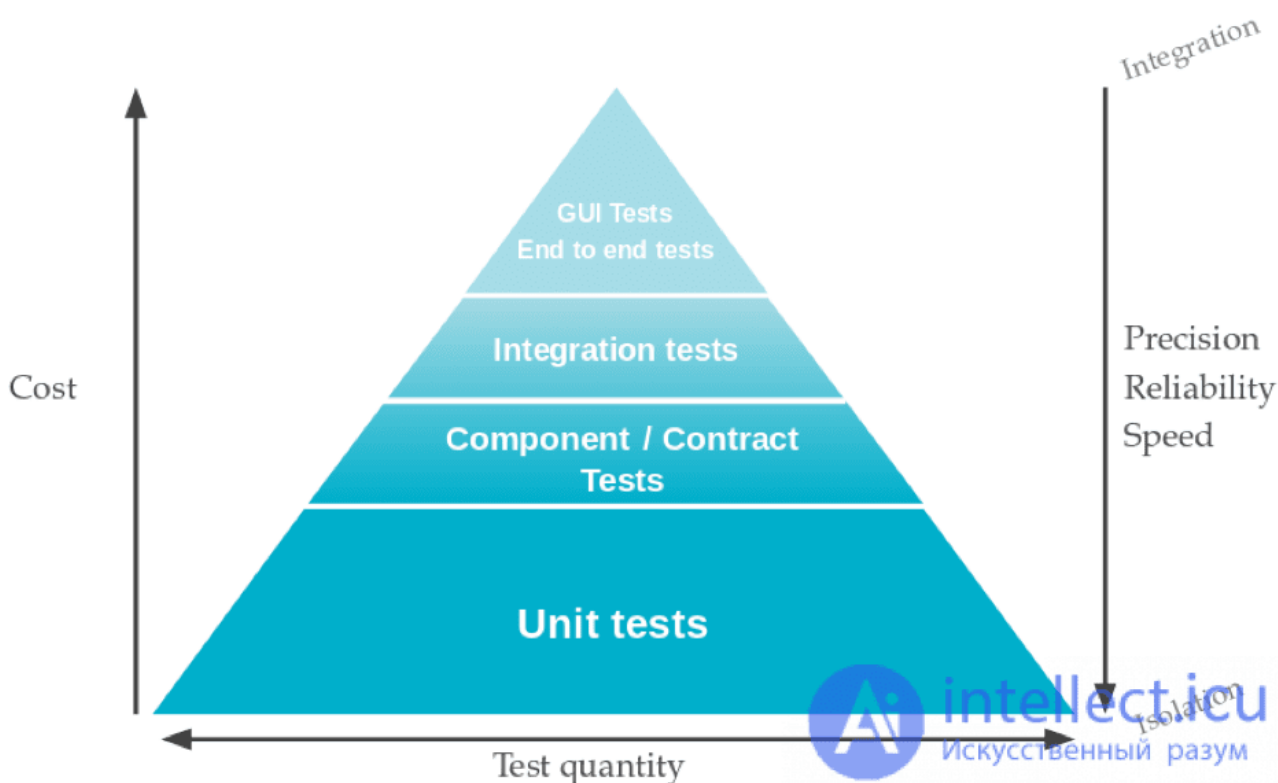
посмотрим как наша система взаимодействует со внешними системами – поставщиками и потребителями, с нашим окружением, то есть проведем [системное тестирование](#), то легко увидим, что сложность тестирования тоже возрастет. Нам нужно будет добиваться одновременной работоспособности всех взаимодействующих систем, хотя и без привлечения специалистов по ним.

В следующей таблице приведены основные критерии выбора типа теста для использования:

Test category	Feedback			Cost (creation + maintenance)
	Precision	Reliability	Speed	
<b>Unit test</b>	Very precise (Function level)	Very reliable (infinitively repeatable)	Very fast (a few seconds)	
<b>Integration Test Functional test</b>	Average (partial software part integration)	Reliable (possibility failures due to dependencies issues)	Relatively fast (a few minutes)	
<b>End to end Test GUI Test</b>	Low (Full software integration)	Low (relatively instable)	Slow (around tens of minutes)	

С этой

точки зрения мы можем сказать: «Отлично! Мне нужно только выполнять модульные тесты », но существуют и другие типы тестов по уважительной причине: модульные тесты не могут проверить все.

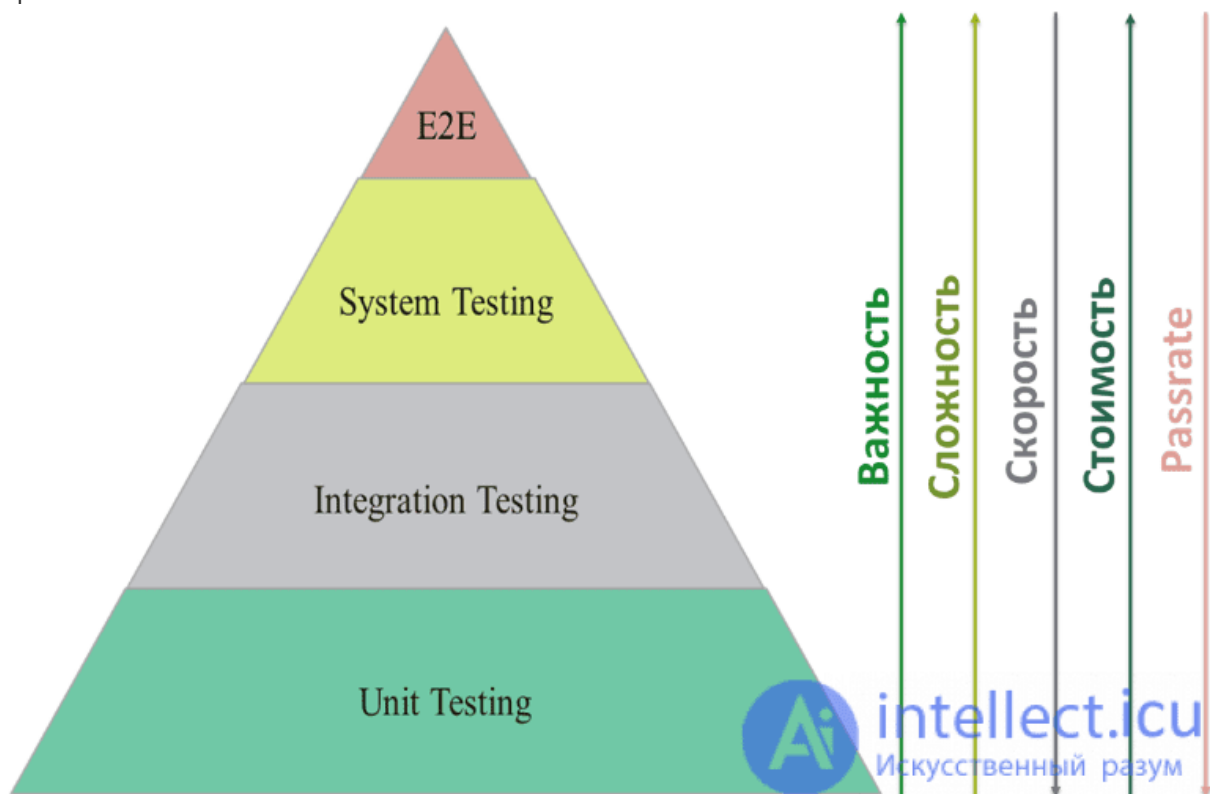


- Мы вкладываем значительные средства в модульные тесты, которые обеспечивают прочную основу для пирамиды, предоставляя нам быструю и точную обратную связь. В сочетании с непрерывной интеграцией модульные тесты обеспечивают защиту от регрессий, что очень важно, если мы хотим контролировать наш продукт в среднесрочной и долгосрочной перспективе.
- На вершине пирамиды мы сводим наше сквозное тестирование к минимуму: например, проверяем интеграцию компонента в общую систему, любые

самодельные графические компоненты и, возможно, некоторые дымовые тесты в качестве приемочных тестов.

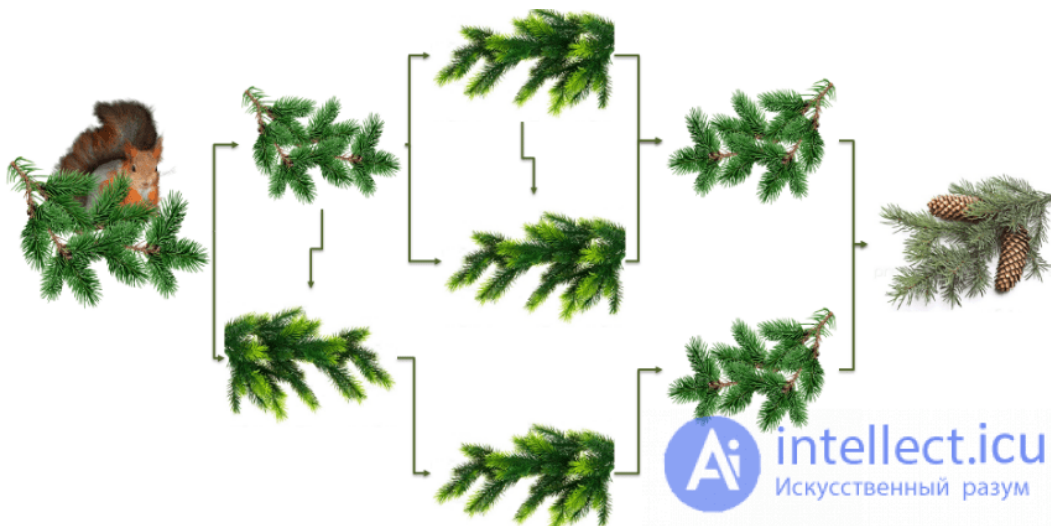
- И в середине пирамиды интеграционные тесты позволяют нам проверять компонент (внутренняя интеграция) и его границы (внешняя интеграция). Принцип снова заключается в том, чтобы отдавать предпочтение внутренним тестам компонентов, которые изолированы от остальной системы, по сравнению с внешними тестами, которые более сложны для реализации:

Согласно intellect.icu есть 4 уровня пирамиды: и она включает так же e2e тестирование, то есть рассмотр всевозможных входов и выходов из вашей программы



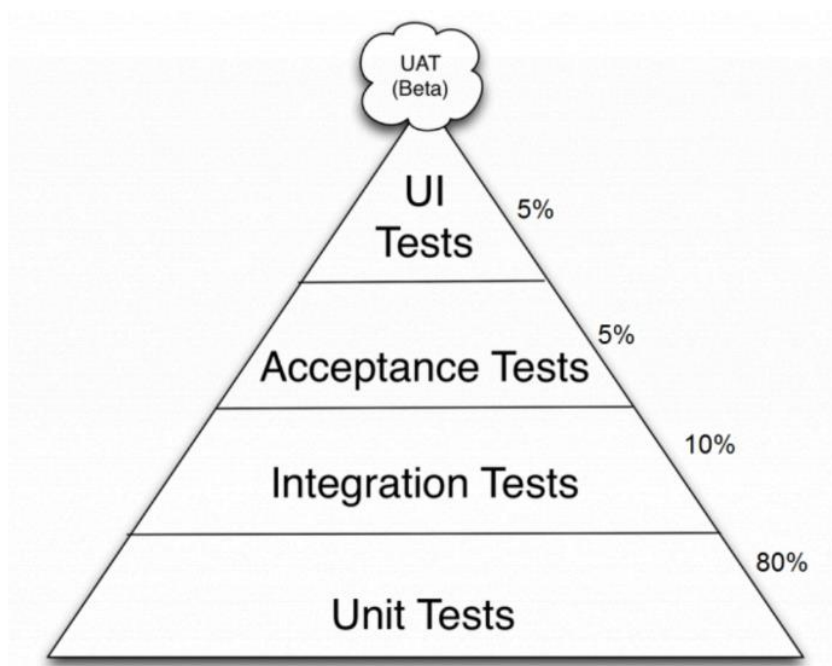
Конечно, проверка интерфейсов классов и функций вряд ли та вещь, которую можно показать заказчику, но без этого прочного, монолитного, безотказного фундамента вряд ли что-то получится выстроить выше. Как правило несколько десятков функций, методов, классов реализуют какую-либо функциональность для заказчика, и по сути десяток модульных тестов можно свести к каким-то верхнеуровневым тестам. Заказчику нужна уже красивая квартира с отделкой, но при этом вряд ли он останется довольным, когда перекошенные окна в его квартире перестанут открываться, а пол и потолок пойдет трещинами от первого подувшего ветерка. Об этом говорит сайт <https://intellect.icu>. Однако самому заказчику зайти в квартиру и проверить ее качество может быть не самой лучшей идеей. Согласитесь, сложно пользователю проверить качество бетона в фундаменте, так и воспроизвести все погодные условия. Так и в тестировании, конечно, верхнеуровневое тестирование нужно, то только тогда, когда у нас отработали модульные тесты, так и тесты уже более высокого уровня.





Сколькими путями может дойти грызунчик до шишки?

[Sberbank](#) например создала такую пирамиду:



- Модульное или блочное тестирование работает с определенными функциями, классами или методами отдельно от остальных частей системы. Благодаря этому при возникновении бага можно по очереди изолировать каждый блок и определить, где ошибка. В этой группе выделяют поведенческое тестирование. По сути оно похоже на модульное, но предполагает жесткий регламент написания тестового кода, который регламентирован набором ключевых слов.
- Интеграционное тестирование сложнее модульного, так как в нем проверяется взаимодействие кода приложения с внешними системами. Например, UI-тесты на EarlGrey отслеживают, как приложение взаимодействует с API-ответами сервера.

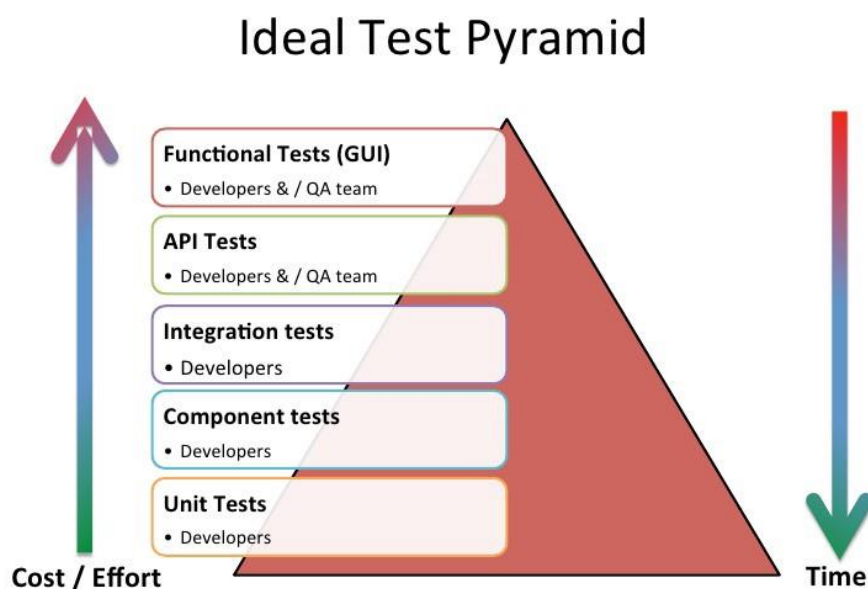
- *Приемочное тестирование* проверяет, что функции, описанные в техническом задании, работают правильно. Оно состоит из наборов больших кейсов интеграционного тестирования, которые расширены или сужены с точки зрения тестового окружения и конкретных условий запуска.
- *UI-тестирование* проверяет, как интерфейс приложения соответствует спецификациям от дизайнеров или запросам пользователей. В нем выделяют *snapshot-тестирование*, когда скриншот приложения попиксельно сравнивают с эталонным. При таком способе невозможно закрыть все приложение, так как сравнение двух картинок требует много ресурсов и загружает тестовые сервера.
- *Ручное или регрессионное тестирование* остается после автоматизации всех остальных сценариев. Эти кейсы выполняются QA-специалистами, когда нет времени на написание UI-тестов или надо поймать плавающую ошибку.

Кроме известных принципов SOLID, которые определяют качество и красоту кода, при программировании тестов нужно учитывать еще два требования: к скорости и надежности.

Если совсем упрощать, то тесты должны быть быстрыми и надежными. Для этого важна синхронность, отсутствие ожидания внешних реакций и внутренних таймаутов.

Вот ище одна пирамида которую нашла на

**qastartby**

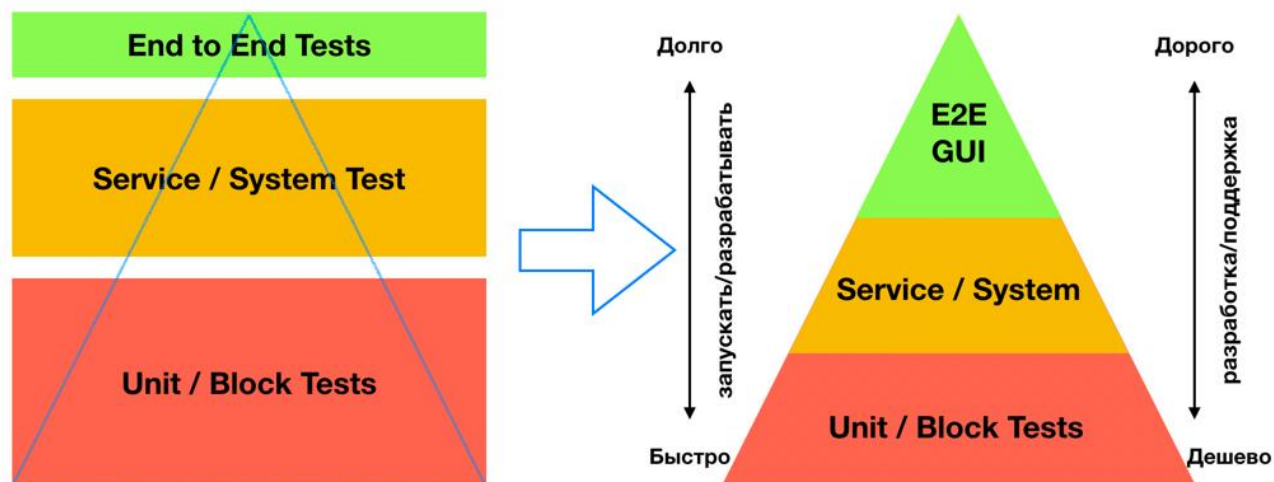


**Мартин Роберт С.**



У основания пирамиды располагаются модульные тесты. Они пишутся программистами для программистов на языке программирования системы. Целью этих тестов является определение спецификации системы на самом нижнем уровне.

## Сайт Teamlead Roadmap



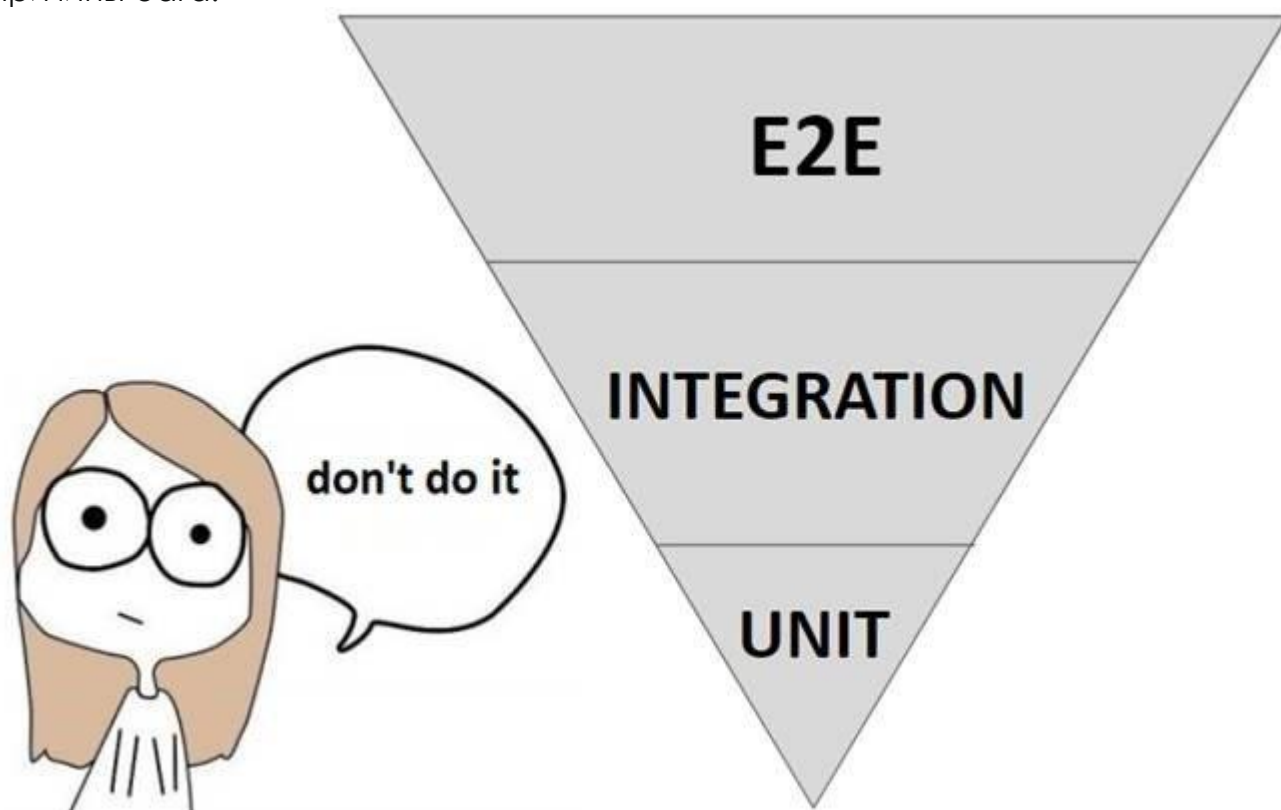
Пирамида тестирования, в том числе, помогает наглядно объяснить причины, почему количество Unit тестов должно быть больше чем интеграционных. Части треугольника закрашенные разными цветами подразумевают количество необходимых тестов данной категории, чем больше площадь, тем больше тестов. Чем ниже находятся на пирамиде тесты, тем:

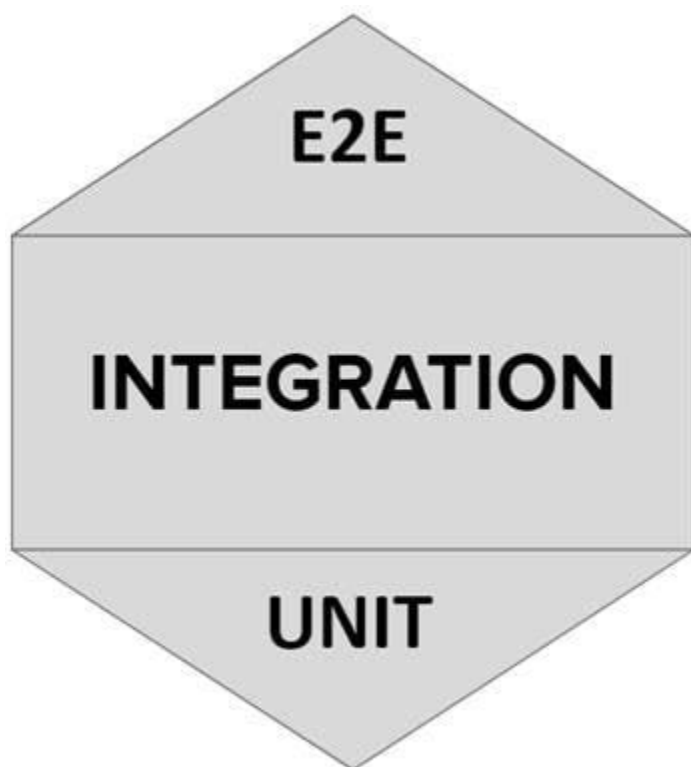
- проще и быстрее они разрабатываются
- ниже затраты на поддержку тестов
- быстрее скорость запуска атомарного теста
- выше уровень изоляции компонент между собой
- меньше нужно денег на содержание инфраструктуры для запуска этих тестов
- ниже уровень необходимой квалификации того, кто эти тесты может разрабатывать

## Константин Яковлев в статье на Troger рассказал и о

**Перевернутой пирамиде**, или рожок тестирования. В этой стратегии основной упор — на E2E-тесты, т. к. они дают наибольшую уверенность, что работа всей системы полностью соответствует ожиданиям конечного пользователя.

С одной стороны, мы уверены в качестве продукта, а с другой — тратим огромное время на получение фидбека, что система работает после внесения каких-либо изменений. Если каждый Unit-тест выполняется за миллисекунды, Integration — за секунды, E2E может занимать несколько десятков секунд или минуты. И даже если тест выявил дефект, мы точно не знаем, в каком сервисе и в каком месте кода произошел сбой. Мы должны будем потратить достаточно много времени на поиск причины бага.





Следующая стратегия — **сота тестирования** (testing honeycomb). Здесь основной упор делается на integration-тесты. Она идеально подходит для микросервисов, в частности, ее используют в Spotify.

Когда сервис небольшой (как говорят, размер микросервиса должен быть таким, чтобы agile-команда смогла его полностью переписать за 1 спринт) и в нем мало бизнес-логики, этот подход дает большие плюсы:

- Уверенность, что код делает то, что должен. При этом виде тестов вы проверяете только реалистичный input сервиса и ожидаемый output. Вам нет никакого дела, как все реализовано внутри.
- Мы можем рефакторить код внутри без изменения тестов. Например, мы можем полностью изменить БД и/или переписать бизнес-логику, это никак не повлияет на тесты.

Но есть и минусы:

- Некоторые потери в скорости выполнения тестов. Как мы уже знаем, каждый Integration-тест выполняется несколько секунд, что приводит к потерям времени. Но в случае микросервисов потери незначительны, потому что объем тестов небольшой.
- Мы можем потерять некоторый фидбэк, если тесты упали. Но хоть мы точно и не знаем место, где произошел сбой, можем быстро его найти и устранить.

Время идет, и наш сервис развивается, в нем появляется больше кода, и становится сложнее бизнес-логика. При использовании стратегии призмы тестирования у нас могут возникнуть большие проблемы. Какие? Давайте разберем.

Все мы сталкивались с ситуацией, когда все тесты зеленые, но выявляется баг. Что мы можем сделать после этого? Найти причину, закрыть ее Integration-тестом, который воспроизводит проблему, пофиксить и выпустить патч. И в этом кроется основной недостаток Integration и любого другого вида сквозного тестирования. Мы упускаем из виду проблемы в архитектуре приложения. Как бы удивительно ни звучало, Unit-тесты нужны не только для проверки бизнес-логики и поиска багов, но и для выявления

проблем в дизайне. Всем известно, что если вы не можете покрыть какую-то часть кода Unit-тестами, у вас проблемы в архитектуре. Unit-тесты позволяют заметить их на ранних стадиях. Например, если для написания одного Unit-теста вам необходимо замочать кучу зависимостей, есть проблема, необходимо провести рефакторинг.

Таким образом, Unit-тесты помогают не только находить логические ошибки, но и выявлять проблемы в дизайне. А что происходит, если вы их не пишете? Архитектура ухудшается и становится более запутанной. Повышается вероятность сделать ошибку при каких-либо изменениях. Это как гидра: пофиксили что-то здесь — сломалось что-то там. И круг замкнулся, у вас опять все тесты зеленые, но появляется ошибка, вы покрываете ее Integration-тестом, фиксиете следствие, а не причину. И так опять, и опять, и опять. В итоге ваш проект состоит только из костылей, а вы — несчастны.

Другая проблема — время. Чем больше у вас Integration тестов, тем больше времени занимает их полный прогон. Для микросервисов со сложной бизнес-логикой и взаимодействием прогон всех тестов может занять ни один десяток минут. Это снова негативно сказывается на производительности и качестве кода.

## Источники:

<https://intellect.icu/piramida-testirovaniya-i-skvozhnoe-testirovanie-end-to-end-naznachenie-primer-9070>

<https://intellect.icu/avtomatizirovannoe-funktsionalnoe-testirovanie-automation-testing-i-functional-automation-testing-6122#term-piramida-testirovaniya>

<https://habr.com/ru/company/sberbank/blog/358224/>

<https://qastart.by/main/terms/64-piramida-testov-testirovaniya>

<https://it.wikireading.ru/4767>

<https://medium.com/@arturbasak/the-testing-pyramid-2fec1c1fef91>

<https://troadmap.io/roles/technical-lead/product-quality/testing/test-pyramid.html>

<https://dou.ua/lenta/digests/qa-digest-34/>

<https://habr.com/ru/post/358950/>