Пирамида тестов заслуживает отдельного обсуждения, особенно по количеству тестов, необходимых для каждой категории. определим две основные категории тестов: модульные (юнит-тесты) и интеграционные.

Тесты	Цель	Требует	Скорость	Сложность	Нужна настройка
Юнит-тесты	класс/метод	исходный код	очень быстро	низкая	нет
Интеграционные тесты	компонент/сервис	часть работающей системы	медленно	средняя	да

Юнит-тесты более широко известны как по названию, так и по своему значению. Эти тесты сопровождают исходный

код и обладают прямым доступ к нему. Обычно они выполняются с помощью фреймворка xUnit или аналогичной библиотеки. Юнит-тесты работают непосредственно на исходнике и имеют полное представление обо всём. Тестируется один класс/метод/функция (или наименьшая рабочая единицей для этой конкретной функциональности), а всё остальное имитируется/заменяется.

Интеграционные тесты (также именуемые сервисными тестами или даже компонентными тестами) фокусируются на целом компоненте. Это может быть набор классов/методов/функций, модуль, подсистема или даже само приложение. Они проверяют компонент путём передачи ему входных данных и изучения выдачи. Обычно требуется какое-то предварительное развёртывание или настройка. Внешние системы можно полностью имитировать или заменить (например, используя СУБД в памяти вместо реальной), а реальные внешние зависимости используются по ситуации. В сравнении с юнит-тестами требуются более специализированные инструменты либо для подготовки тестовой среды, либо для взаимодействия с ней.

Основное практическое правило таково: если тест…

- использует базу данных,
- использует сеть для вызова другого компонента/приложения,
- использует внешнюю систему (например, очередь или почтовый сервер),
- читает/записывает файлы или выполняет другие операции ввода-вывода,
- полагается не на исходный код, а на бинарник приложения,

… то это интеграционный, а не модульный тест.

Разобравшись с терминами, можно погрузиться в список антипаттернов. Их порядок примерно соответствует их распространённости. Самые частые проблемы перечислены в начале. некоторые типы проблем могут обнаружить только интеграционные тесты. Канонический пример — всё, что связано с операциями СУБД. Транзакции, триггеры и любые хранимые процедуры БД можно проверить только с помощью интеграционных тестов, которые их затрагивают. Любые подключения к другим модулям, разработанным вами или внешними

командами, требуют интеграционных тестов (они же контрактные тесты). Любые тесты для проверки производительности, являются интеграционными тестами по определению. Вот краткий обзор того, почему нам нужны интеграционные тесты:

Тип проблемы	Определяется юнит-тестами	Определяется интеграционными тестами
Основная бизнес-логика	да	да
Проблемы интеграции компонентов	нет	да
Транзакции	нет	да
Триггеры/процедуры БД	нет	да
Неправильные контракты с другими модулями/API	нет	да
Неправильные контракты с другими системами	нет	да
Производительность/таймауты	нет	да
Взаимные/самоустраняемые блокировки	возможно	да
Перекрёстные проблемы	нет	да

безопасности		

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

большая проблема интеграционных тестов — их скорость. Как правило, интеграционный тест выполняется на порядок медленнее юнит-теста. Модульному тесту нужен только исходный код приложения и больше ничего. С другой стороны, интеграционные тесты могут производить операции ввода-вывода с внешними системами, так что их

гораздо труднее оптимизировать.

Интеграционные тесты сложнее отладить, чем модульные тесты

Последняя причина, почему не рекомендуется ограничиваться только интеграционными тестами (без модульных) — это количество времени на отладку неудачного теста. Поскольку интеграционный тест по определению тестирует несколько программных компонентов, сбой может быть вызван любым из протестированных компонентов. Выявить проблему тем сложнее, чем больше задействовано компонентов.

При сбое интеграционного теста необходимо понимать причину сбоя и как её исправить. Сложность и охват интеграционных тестов делают их чрезвычайно трудными для отладки. Опять же в качестве примера предположим, что для вашего приложения сделаны только интеграционные тесты. Скажем, это типичный интернет-магазин.

Компонентное (модульное) тестирование предполагает функциональность ищет дефекты в частях приложений , которые происходят и могут быть протестированы по-отдельности (модули программ, объектов, классов, функций и т.д.). Обычно компонентное (модульное) тестирование требует наличия кода, который необходимо проверять и поддерживать при разработке, таких как каркасы (каркасы) для модульного тестирования или инструменты для отладки среды тестирования. Все обнаруженные дефекты, как правило, исправляются в коде без формального их описания в системе управления ошибками (система отслеживания ошибок). При этом код с высокой степенью чистоты (методами) для того, чтобы т есты были изолированы от окружения (БД, сеть, файловая система, время).

Один из наиболее эффективных подходов к компонентному (модульному) программированию - это подготовка тестовых тестов до начала базового кодирования (разработки) программного обеспечения. Это называется разработка от тестирования (разработка через тестирование) или подход тестирования вначале (тестовый первый подход). При этом подходе к заражению и интегрируются небольшие случаи заражения, в связи с

чем возникают случаи возникновения тестов, записанные до начала кодирования. Разработка ведется до тех пор, пока все тесты не будут успешно пройдены.

Суть пирамиды тестирования не хитрая: в основе тестирования следует использовать самые простые и самые быстрые в написании и исполнении тесты – модульные тесты. 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. 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.

Из этой пирамиды главное запомнить два принципа:

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

Можно сказать, что разработка ПО - это движение по пирамиде снизу вверх. Важно отметить:

1. Тест должен быть на том же уровне, что и тестируемый объект. Например, модульный тест должен быть на компонентном уровне. Это неправильно, если на приемочном уровне запускается тест, который проверят минимальную единицу кода.
2. Тесты уровнем выше не проверяют логику тестов уровнем/уровнями ниже.
3. Чем выше тесты уровнем, тем они:

- сложней в реализации, и соответственно, дороже в реализации;
- важнее для бизнеса и критичней для пользователей;
- замедляют скорость прохождения тестовых наборов, например, регресса.

- компонентный уровень

Юнит-тест (unit test), или модульный тест, — это программа, которая проверяет работу небольшой части кода. Разработчики регулярно обновляют сайты и приложения, добавляют фичи, рефакторят код и вносят правки, а затем проверяют, как всё работает.

Тестировать систему целиком после каждого обновления — довольно муторно и неэффективно. Поэтому обновлённые или исправленные части кода прогоняют через юнит-тесты. Чем выше тест в пирамиде, тем больше частей программы он затрагивает. Высокоуровневые тесты «ближе к бизнесу»: они проверяют бизнес-логику и пользовательские процессы. А те, что внизу пирамиды, помогают найти проблемы в отдельных частях кода. Например, какую-нибудь функцию, которая генерирует имя файла.

Большая часть тестов работает на верхних уровнях и не позволяет точечно отлавливать баги. Те же интеграционные тесты проверяют, как взаимодействуют между собой разные системы. А E2E-тесты исследуют

процессы со стороны пользователя: например, когда он регистрируется и логинится на сайте.

В отличие от них, юнит-тесты нужны в следующих случаях:

- если код непонятен — на ревью возникли вопросы по его работе;
- если код часто меняется — чтобы не проверять его вручную;
- если обновления в одной части кода могут сломать что-то в другой части.

Некоторые программисты пишут только юнит-тесты, а на интеграционные или E2E-тесты жалеют времени. На самом деле нужно покрывать систему всеми видами тестов, чтобы знать, как взаимодействуют друг с другом разные части программы, какие промежуточные результаты они выдают. Но в то же время, если юнит-тесты показывают ошибку, её покажет и интеграционный, и E2E-тест.

Чаще всего называют юнит тестированием. Реже называют модульным тестированием. На этом уровне тестируют атомарные части кода. Это могут быть классы, функции или методы классов.

Пример: твоя компания разрабатывает приложение "Калькулятор", которое умеет складывать и вычитать. Каждая операция это одна функция. Проверка каждой функции, которая не зависит от других, является юнит тестированием. Юнит тесты находят ошибки на фундаментальных уровнях, их легче разрабатывать и поддерживать. Важное преимущество модульных тестов в том, что они быстрые и при изменении кода позволяют быстро провести регресс

Тест на компонентном уровне:

1. Всегда автоматизируют.
2. Модульных тестов всегда больше, чем тестов с других уровней.
3. Юнит тесты выполняются быстрее всех и требуют меньше ресурсов.
4. Практически всегда компонентные тесты не зависят от других модулей (на то они и юнит тесты) и UI системы.

В 99% разработкой модульных тестов занимается разработчик, при нахождении ошибки на этом уровне не создается баг-репортов. Разработчик находит баг, правит, запускает и проверяет и так по новой, пока тест не будет пройден успешно.

Интеграционный уровень

Проверят взаимосвязь компоненты, которую проверяли на модульном уровне, с другой или другими компонентами, а также интеграцию компоненты с системой (проверка работы с ОС, сервисами и службами, базами данных, железом и т.д.). Часто в английских статьях называют service test или API test.

В случае с интеграционными тестами редко когда требуется наличие UI, чтобы его проверить. Компоненты ПО или системы взаимодействуют с тестируемым модулем с помощью интерфейсов. Тут начинается участие тестирования. Это проверки API, работы сервисов (проверка логов на сервере, записи в БД) и т.п. Строго говоря на модульном уровне тестирование тоже участвует. Возможно помогает проектировать тесты или во время регресса смотрит за прогоном этих тестов, и если что-то падает, то принимает меры.

Отдельно отмечу, что в интеграционном тестировании, выполняются как функциональные (проверка по ТЗ), так и нефункциональные проверки (нагрузка на

связку компонент). На этом уровне используется либо серый, либо черный ящик.

В интеграционном тестировании есть 3 основных способа тестирования (представь, что каждый модуль может состоять еще из более мелких частей):

1. Снизу вверх (Bottom Up Integration): все мелкие части модуля собираются в один модуль и тестируются. Далее собираются следующие мелкие модули в один большой и тестируется с предыдущим и т.д. Например, функция публикации фото в соц. профиле состоит из 2 модулей: загрузчик и публикатор. Загрузчик, в свою очередь, состоит из модуля компрессии и отправки на сервер. Публикатор состоит из верификатора (проверяет подлинность) и управления доступом к фотографии. В интеграционном тестировании соберем модули загрузчика и проверим, потом соберем модули публикатора, проверим и протестируем взаимодействие загрузчика и публикатор.
2. Сверху вниз (Top Down Integration): сначала проверяем работу крупных модулей, спускаясь

ниже добавляем модули уровнем ниже. На этапе проверки уровней выше данные, необходимые от уровней ниже, симулируются.Например, проверяем работу загрузчика и публикатора. Руками (создаем функцию-заглушку) передаем от загрузчика публикатору фото, которое якобы было обработано компрессором.

3. Большой взрыв ("Big Bang" Integration): собираем все реализованные модули всех уровней, интегрируем в систему и тестируем. Если что-то не работает или недоработали, то фиксим или дорабатываем.

- Системный уровень

. Системное тестирование (System Testing)

Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований ,дефекты в системе в целом. При этом выявляется 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 в 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.

О системном уровне говорили в интеграционном. Тут отметить только то, что:

1. Системный уровень проверят взаимодействие тестируемого ПО с системой по функциональным и нефункциональным требованиям.
2. Важно тестировать на максимально приближенном окружении, которое будет у конечного пользователя.

Тест-кейсы на этом уровне подготавливаются:

1. По требованиям.
2. По возможным способам использования ПО.

На системном уровне выявляются такие дефекты, 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.

На этом уровне используют черный ящик. Интеграционный уровень позволяет верифицировать требования (проверить соответствие ПО прописанным требованиям).

Приемочное тестирование

Приемочное тестирование или приемо-сдаточное испытание (Acceptance Testing)

Приемочное тестирование или приемо-сдаточное испытание (Acceptance Testing) — формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:

- определения удовлетворения системой приемочным критериям;
- вынесения решения заказчиком или другим уполномоченным лицом принятия приложения.

Приемочное тестирование выполняется на основании набора типичных тестовых случаев и сценариев, разработанных на основании требований к данному приложению.
Решение о проведении приемочного тестирования принимается, когда:

- Продукт достиг необходимого уровня качества.
- Заказчик ознакомлен с Планом Приемочных Работ (Product Acceptance Plan) или иным документом, где описан набор действий, связанных с проведением

приемочного тестирования, дата проведения, ответственные и т.д.

Фаза приемочного тестирования длится до тех пор, пока заказчик не выносит решение об отправлении приложения на доработку или выдаче приложения.

Также часто называют E2E тестами (End-2-End) или сквозными. На этом уровне происходит валидация требований (проверка работы ПО в целом, не только по прописанным требованиям, что проверили на системном уровне).

Проверка требований производится на наборе приемочных тестов. Они разрабатываются на основе требований и возможных способах использования ПО.

Отмечу, что приемочные тесты проводят, когда (1) продукт достиг необходимо уровня качества и (2) заказчик ПО ознакомлен с планом приемки (в нем описан набор сценариев и тестов, дата проведения и т.п.).

Приемку проводит либо внутреннее тестирование (необязательно тестировщики) или внешнее тестирование (сам заказчик и необязательно тестировщик).

Важно помнить, что E2E тесты автоматизируются сложнее, дольше, стоят дороже, сложнее поддерживаются и трудно выполняются при регрессе. Значит таких тестов должно быть меньше.

- Среда тестирования — это настройка программного и аппаратного обеспечения для групп тестирования для выполнения тестовых случаев. Другими словами, он поддерживает выполнение теста с настроенным оборудованием, программным обеспечением и сетью.

Испытательный стенд или тестовая среда настраиваются в соответствии с требованиями тестируемого приложения. В некоторых случаях испытательный стенд может представлять собой комбинацию тестовой среды и тестовых данных, которые он использует.

Настройка правильной среды тестирования гарантирует успех тестирования программного обеспечения. Любые недостатки в этом процессе могут привести к дополнительным затратам и времени для клиента.

- Типы сред тестирования

Среда тестирования - это конфигурация типов сред для определенного теста. Например, среда тестирования, которая использует Firefox, Windows XP и Apache Tomcat, является конфигурацией следующих типов сред: браузер, операционная система и сервер приложений.

Каждый тип среды может иметь одно или несколько значений. Например, тип среды браузера может иметь в качестве своего значения Firefox, Internet Explorer или Safari. Платформа плана тестирования — это набор из всех типов сред и их значений для плана тестирования. Платформа определяет, что поддерживает продукт и что тестирует группа тестирования.

Типы сред тестирования показываются в следующих областях:

- В редакторе создания среды тестирования в описании ресурса лаборатории
- В редакторе создания запроса в описании ресурса лаборатории
- В редакторе плана тестирования в разделе Среды тестирования

- В редакторе тестовых наборов, раздел Записи выполнения тестовых наборов, мастер Создать записи выполнения тестовых наборов
- В редакторе комплектов тестов, раздел Записи выполнения комплектов тестов, мастер Создать записи выполнения комплекта тестов

Контрольный список тестовой среды

аппаратные средства		
1	Проверьте, доступно ли необходимое оборудование для тестирования?	Если это не так, проанализируйте время поставки!
	Проверьте, доступно ли периферийное оборудование?	Например, сканеры, специальные принтеры, карманные компьютеры и т. Д.

Программное обеспечение / соединения		
2	Указаны ли необходимые приложения?	Приложение, такое как Excel, Word, рисунки и т. Д.
	Для нового программного обеспечения существует ли тестовая среда для организации?	Имеет ли организация опыт использования и обслуживания программного обеспечения?
Экологические данные		
3	Проверьте, доступны ли стандартные наборы тестовых данных?	С набором регрессионных тестов рассмотрите администрирование дефектов для сбора тестовых данных.
	Существуют ли	Рассмотрим функциональное

	соглашения с владельцами тестовых данных о тестовых данных?	обслуживание.
Инструменты / процессы обслуживания		
4	Проверьте, существует ли одна точка контакта для обслуживания тестовой среды?	Если нет, подготовьте список всех возможных членов, участвующих в поддержании работоспособности тестовой среды. Он должен также включать их контактную информацию.
	Достигнуто ли соглашение о готовности и качестве тестовой среды?	Например, критерии приемки, требования к техническому обслуживанию и т. Д. Также проверьте, согласованы ли другие / дополнительные атрибуты качества для сред.

	Все ли участники, вовлеченные в процесс обслуживания, известны?	

Риски выявляются и оцениваются группой заинтересованных сторон в ходе сессий мозговых штурмов. В команду должны входить бизнес-аналитики или носители знаний о системе от заказчика, разработчики, менеджер или руководитель проекта, архитекторы, QA-специалисты. К выявлению и оценке рисков мы привлекаем в том числе специалистов по информационной безопасности, сотрудников, которые непосредственно работают с текущей системой, бизнес-аналитика, который погружен в процессы.

Таким образом, каждый риск логически вытекает из требований, которые есть в системе, но не ограничивается ими. Риски дополняют требования и выявляют

дополнительные кейсы, которые могут возникнуть при работе с системой.
Риски могут быть снижены или компенсированы в зависимости от целей проекта и требований к системе.
Принимается условие, чтобы риск был покрыт тест-кейсом хотя бы один раз:

1. На каждый продуктовый риск подготавливается набор тест-кейсов RP1-RP4 с условием, что каждое требование и каждый риск должен быть покрыт хотя бы одним тест-кейсом. В зависимости от целей тестирования происходит расширение набора тест-кейсов до определенного уровня детализации.

2. На каждый организационный риск подготавливается список мероприятий, которые могут снизить влияние риска на разрабатываемый продукт.
Методики оценки и управления рисками