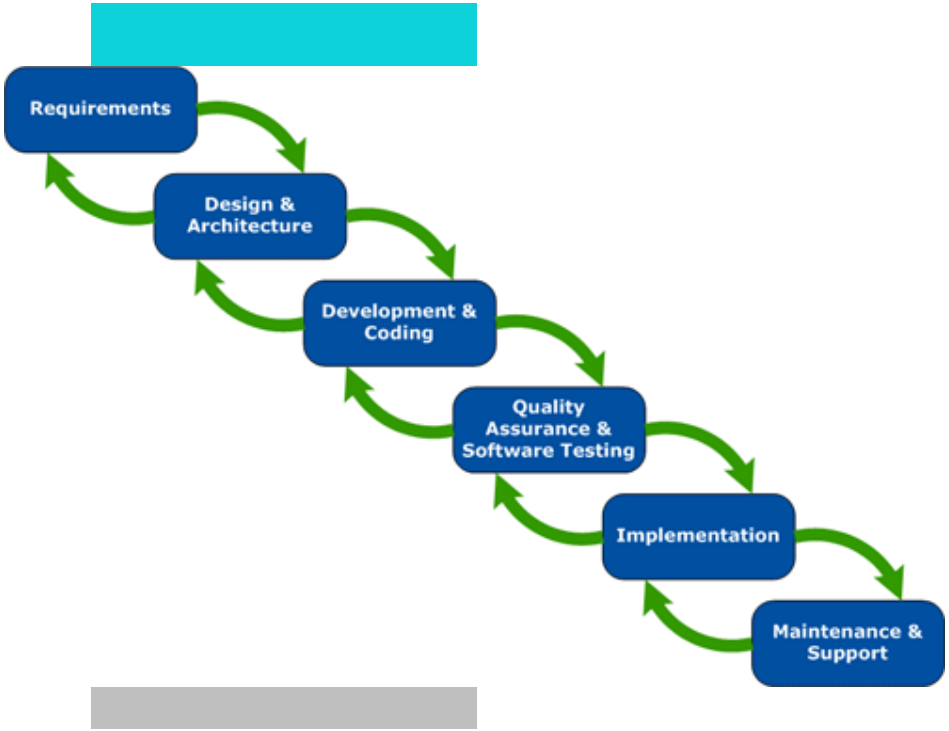
# Модели за разработка на софтуер - тестването като част от жизнения цикъл на разработката на приложението. Валидация и Верификация. Нива на тестване. Типове тестове

# Методологии за разработка на софтуер

Методологиите за разработка на софтуер се делят на две големи групи: последователни (sequential)  и итеративно-инкрементални (iterative-incremental)

# Методологии за разработка на софтуер




## Waterfall

Както името подсказва, фазите при този модел следват логично отгоре надолу, като водопад. Макар, че се използва и при други производства, най-често приложим е при разработката на софтуер. Състои се от ясно определени и разграничими фази: концепция и определяна на изискванията, дизайн, имплементация, тестване и поддръжка. Преминаването в следващата фаза може да стане само след приключване на предишната .
Моделът работи добре при малки проекти, когато ръководителят на проекта е достатъчно уверен, че няма неточни или непълни изискавания. **При този модел тестването може да започне само след приключване на фазата по разработката**

# Фази на Waterfall модела

1. **Анализ на изискванията** - Анализирането на изискванията на проекта е първата важна фаза в методологията за управление waterfall. През нея се изчучава и разбира бизнес модела на клиента. Това изисква възможност за комуникация с клиентите и за анализ на потенциалните рискове, свързани с проекта.

2. **Дизайн** - След като се установи какви са целите на проекта, трябва да се намери решение, което да удовлетворява поставените изисквания. Тази фаза може да включва произволен брой елементи, като например проектирането на софтуер, включително сигурност, дизайн на процеси или дизайн на продукта.

3. **Разработка** – Действителната фаза на разработката на продукта въз основа на дизайна, определен в предишната стъпка, се определят и разпределят и изпълняват задачите за изграждане на продукта

4. **Тестване** - След като продуктът разработен и готов, тестовете се извършват, за да се провери дали той е без грешки. Също така е важно да се провери дали продуктът или услугата са били направени в съответствие с очакванията на клиента, по същество отговаря ли на изискванията.

5. **Внедряване** - След тестването продуктът трябва да се внедри в съответствие с изискаванията, насоките и спецификите на клиента.

6. **Поддръжка** - След като окончателният продукт бъде доставен на клиента, се осигурява услуга за поддръжка, за да се увери, че продуктът или услугата продължават да се изпълняват според очакванията. Периодът на поддържане обикновено е за определен период от време.

# Методологии за разработка на софтуер - Waterfall

## Разпределение по време

Анализ и дизайн – 20-40%
Разработка 30-40%
Тестване и внедряване  20-30%

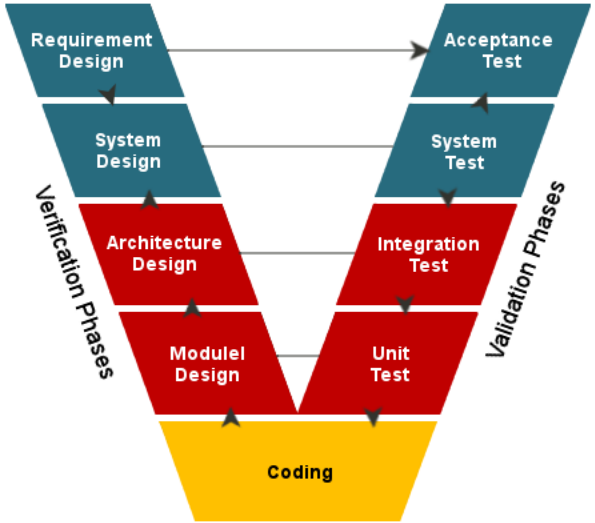
Waterfall модела е приложим при сравнително малки проекти със стабилни изисквания, при които вероятността за промяна на бюджета малка или никаква. Той е силно структуриран с конкретни срокове  и може лесно да бъде планиран и мониториран.

Основният  му недостатък е в така наречния **“time-to-market”** - времето до внедряването на продукта в производство (**production**). При големи проекти вероятността началните изисквания да се променят през този период е много голяма. В края на проекта може да е укаже, че произведеният софтуер, вече не може да изпълни предназначението си, защото прекалено много обстоятелства и изисквания са се променили.

# Методологии за разработка на софтуер – V-model

## V- model

V-модела се смята за разширение на Waterfall модела . Той показва връзката между всяка фаза от разработката на продукта със съответсващата и фаза от процеса на тестване . За всяка фаза от разработката трябва да има тест или тест фаза, които да проверят дали приложението отговаря на изискаванията, описани в спецификацията.




**Верификация:**

- Анализ на изскванията
- Дизайн на системата
- Дизайн на архитектирата
- Дизайн модулите
- Писане на код

**Валидация:**

- Component/Unit testing
- Integration testing
- System testing
- User Acceptance testing

# Валидация и Верификация

## Верификация

**Дефиниция**: Процесът на оценяване на софтуер, за да се определи дали артефактитие на дадена фаза на разработка на софтуера  отговарят на условията, наложени в началото на тази фаза.

Верификацията е **статичен процес** за проверка на документи, дизайн, код  или програма. Тя вкючва всички дейности, отнасящи се към произвеждането на софтуер с високо качество: инспекции, анализ на дизайна, анализ на спецификацията.

**Верификацията помага да се определи дали софтуерът/системата е с високо качество, но няма да докаже,че системата е годна за използване.**

## Валидация

**Дефиниция:** Процесът на оценяване на софтуера по време или в края на разработката, за да се определи дали той отговоря на изизкванията.

Валидацията е динамичен процес  за тестване на крайния  продукт от съответната фаза.

**Целта на валидацията е да докаже, че програмата или продуктът са годни за използване от крайните потребители с фокус върху техните нужди**

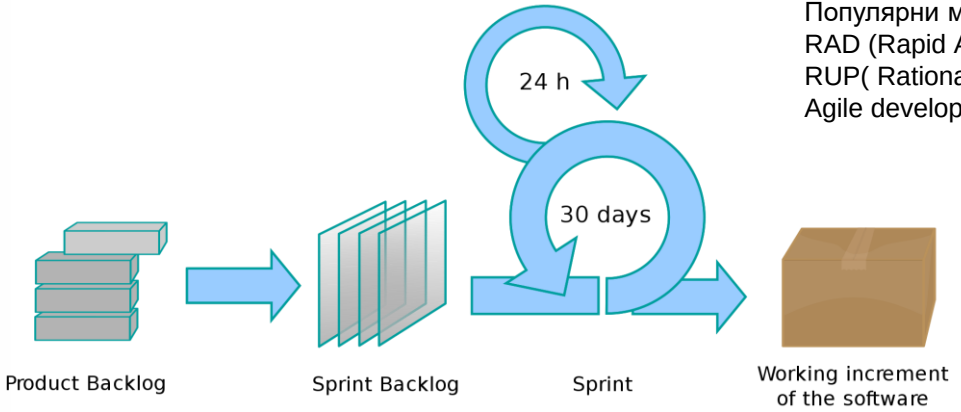
# Валидация § Верификация

| Верификация | Валидация |
| --- | --- |
| **Верификацията** е статичен процес на оценка на документи, дизайн, код, програма | **Валидацията** е динамичен процес на тестване на крайния продукт |
| Не включва изпълнение на кода | Изискава изпълнение на кода |
| **Верификацията** включва методи като inspection, review , walkthroughs | **Валидацията** използва методи като Black box testing, grey box testing , white box testing |
| **Верификацията** е процес на ниско ниво . Може да хване грешки, които валидацията не може да хване | **Валидацията** е процес на високо ниво. Може да хване грешки, които верификацията не може. |
| Предметите на **верификацията** са изискванията , спецификацията, акрхитектурата на приложението, дизайна , модела на данните, дизайна на базата данни. | Предмет на **валидацията** е самия продукт: негов компонент, модул , няколко интегрирани модула , крайния продукт |
| Извършва се преди валидацията | Извършва се след верификацията |

# Итеративно – инкрементални модели за разработка на софтуер „Agile”

## Agile

При итеративно- инкременталните модели проектът се разбива на серия от инкременти, всеко от които доставя част от функционалността на целия проект. Изискванията са приоритизирани и доставени в определен ред, така че да се постигне първоначално поставената цел на инкремента. В някои ( но не във всички) версии на този модел , всеки по-подпроект (инкремент) следва мини V-model със свой собствен дизайн,  писане на код и тест фази.
На края на инкремента може да се направи  или не доставка на клиента . Дори ако се реши, че няма да се прави доставка, продуктът трябва да е в състояние годно за доставка (potentially shippable).




Популярни методологии ,следващи този модел са:
RAD (Rapid Application Development),
RUP( Rational Unified Process),
Agile development (SCRUM) .

# Agile - ползи за тестването

- Рисковете са анализирани и елиминирани в началните фази на проекта. След това те се ревизират във всеки инкремент
- Плановете са ревизират и променят след всеки инкремент
- Могат да бъдат добавени инкременти изцяло фокусирани върху качеството, ако е необходимо.

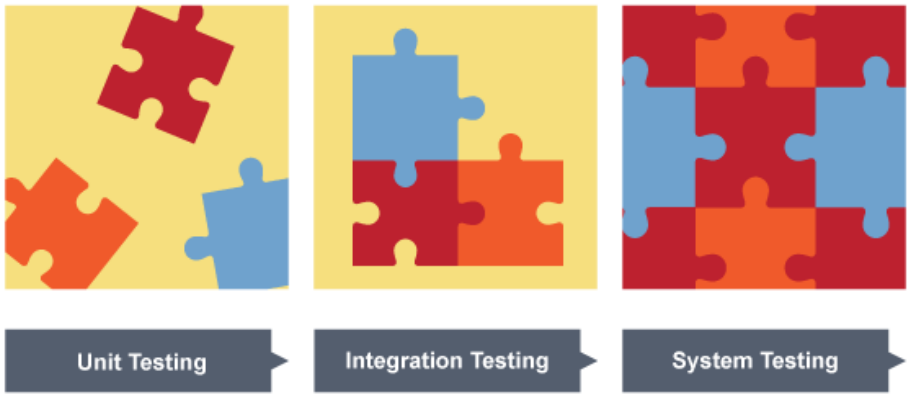
# Тестването в жизнения цикъл на приложението

## Който и модел за разработка на софтуера да се избере, доброто тестване има няколко характеристики:

- За всяка дейност по разработката, има съответсваща дейност по тестването
- Всяко ниво на тестване има цели, специфични за нивото
- Анализите и дизайна на тестовете за дадено ниво трябва да започнат със съответсващото ниво от разработката
- Екипът, който ще извърши тестването трябва да е включен и работи активно от момента, в който черновите или първите версии на докуметацията са достъпни за екипите по разработката.

# Нива на тестване (Test levels)

- Component/Unit Testing
- Integration Testing
- System Testing
- User acceptance Testing



# Component/unit testing

## Какво е компонент?
## Минимална единица код , която може да бъде тествана в изолация.

Компонент могат да бъдат само няколко реда код, или пък метод,  дори програма , която съдържа хиляди редове код.

**Какво е component testing?**
**Процесът по тестването на отделни софтуерни компоненти.**
Компонентното тестване е първата форма на тестване на кодът и обикновено се прави веднага след като той е написан. По своята природа компонентното тестване е строго техническо, за това се изисква високо техническо ниво на човекът , който ще го прави . Той трябва да познава добре кодът и неговите специфики. Най-често този вид тестване се прави от разработчиците, хората, които пишат кода. Най-добре е това да бъде самия автор на кода.
Понеже става дума за отделен компонент, човекът, който тества има нужда да постави този компонент в така наречения „ wrapper", известен още като „harness" или „driver". Това може да са програми, написани специално за нуждите по тесването на дадения продукт или могат да бъдат платени или open-source програми.
„**Test harness" понятието се среща още като „automated test framework**" и представлява набор от софтуерни програми и данни, конфигурирани така че да изпълняват програма или компнент под ралични условия, с цел да се наблюдава поведенението й както и изходните резултати.

**Популярни test frameworks в Java света са : Junit, TestNG, JWalk, JTest и др.**
Един от възможните подходи към компонентното тестване е тестовете да се напишат преди написването на сътоветния компонент, Това се нарича **test-first-approach** или **test-driven development.**
Други термини , под които е известно компонентно тестване са: **unit testing, module testing, program testing.**

# Integration testing

## Какво е integration testing?

**Тествне с цел откриване на дефекти в инерфайсите и във взаимодействеито между вече интегрираните компоненти.**

# Типове тестове в интеграция

**Component integration testing:**

Процесът по разработката на софтуера почти винаги се базира на модуларен подход. Пишат се модули , който се свързват за да формират крайното приложение от стотици дори хиляди компоменти, които си взаимодействат.  Веднага след като компонентът е тестван в изолация  той трябва да бъде свързан или интегрират с другите компоненти и взаимодействеито между тях трябва да бъде тествано.
Този процес се нарича **integration testing in the small** или **component integration testing .** Целта на този вид тестване е да определи колко добре работят отделните компоненти заедно.  Така след като е извършено Unit тестване, два или повече от компонентите се свързват и тестват, като се добавят нови компоненти и тестовете се разширяват. С добавянето на повече компоненти , се формират под-системи , които могат да бъдат тествани , докато се формира цялата система.  Тъй като се изизкват специални  инструменти : test harness, drivers, обикновето този вид тестване се прави от разработчиците.

# Типове тестове в интеграция

**System Integration testing**

Този подтип на integration testing се нарича още **integration testing in the large**. Целта при него е да се подложи на тестване ваимодействието между отделните системи в организацията или интеграцията на приложението с други системи извън компанията собственик на продукта.

Големите софтуерни приложения не работят в изолация . Те си ваимодействат в други системи и приложения , често написани на други програмни езици и със съвсем раличен жизнен цикъл. В допълнение всички те, без изключение обработват едно единствено нещо – данни . След обработката им, тези данни могат да бъдат подадени на друга система или запазени в база данни. Всичко това се прави с цел да се изпълни даден бизнес процес. Веднага след като системата е тествана в изолация , тя трябва да се подлжи на тестване в интеграция с останалите системи. Целта е да се провери, че данните идващи от други системи се обработват правилно и че данните , които се предават на други системи, са годни за употреба от другите системи .

Процесът по тестването се усложнява допълнителни при наличието на различни протоколи за комуникация между системите, както и когато системите работят на различни платформи.

Когато се извършва интегрирането на системите, това се случва по един от следните начини:

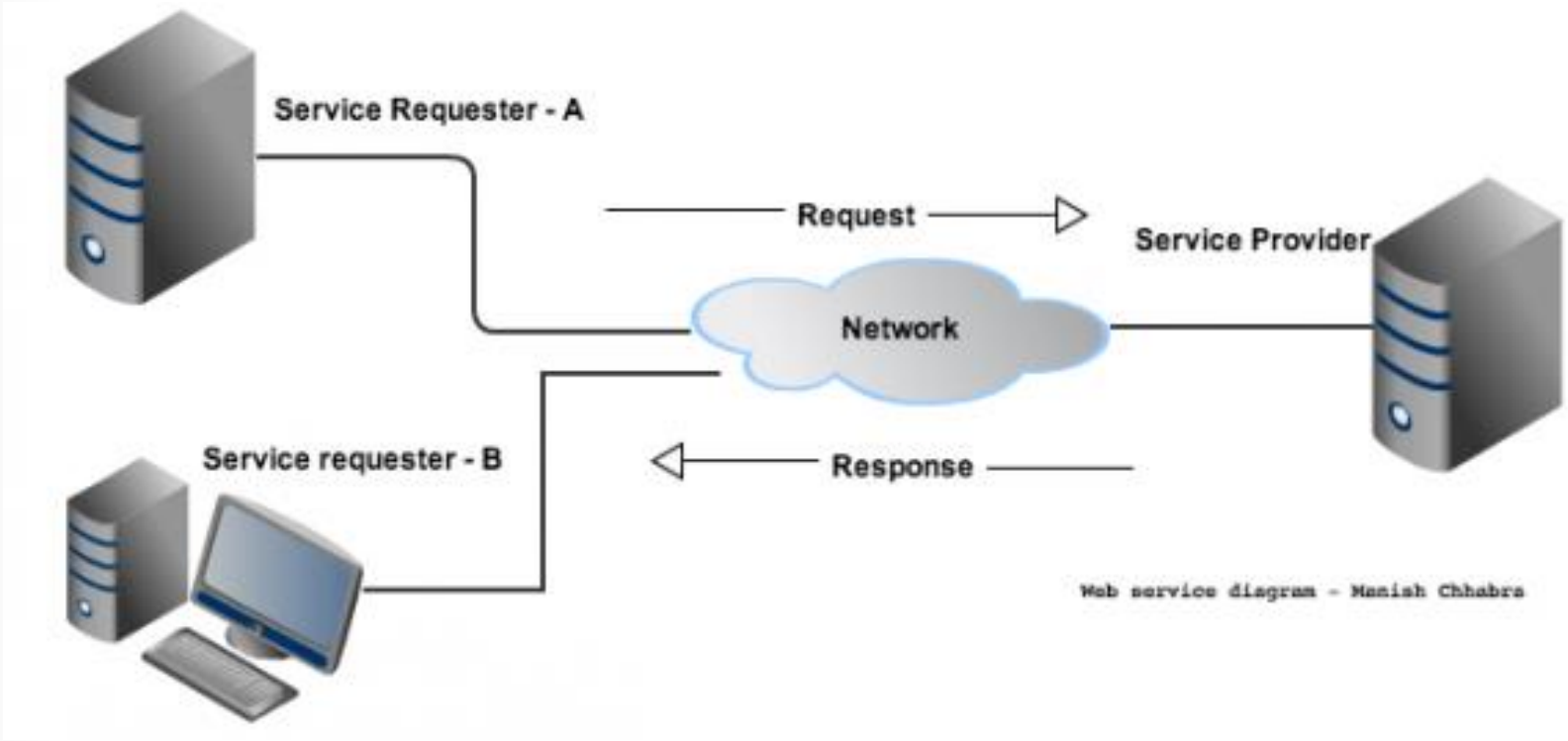
**Изходни данни от система А се четат директно в система Б**

- Ако трета страна притежава система Б , тогава integration тестовете за А , ще се съсредоточат върху генерирането на правилни изходни данни
- Ако система А е външна, а система Б е вътрешна, тогава integration тесването ще се съсредоточи да провери дали система Б може да прочете правилно данните от система А
- Ако и двете системи са вътрешни , то integration тестването ще включва горните 2 активности.

**Изходните данни от ситемата А се трансформират (конвертират)преди да че прочетат от системата Б**

При този вид тестване усилие трябва да се направи и за тестването на процедурата по конверитрането. То ще включва взимане на данни от А , конвертиране и след това подаване и обработка от система Б.

# Типове тестове в интеграция



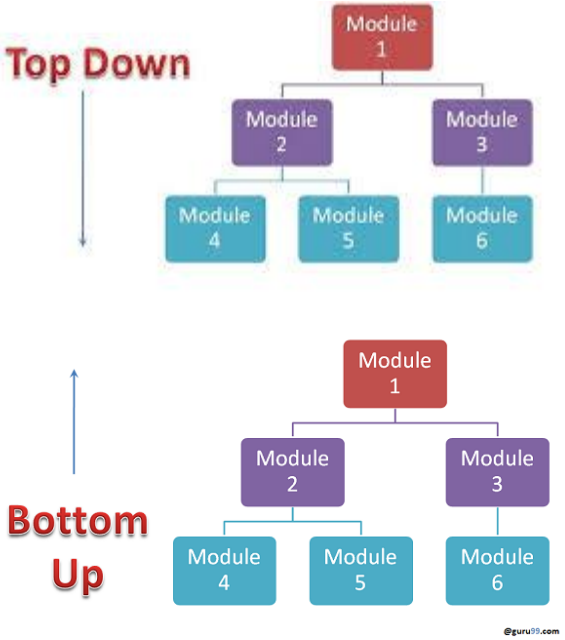
# Подходи тестове в интеграция

**Инкрементални**

**Top down**
Това е подход при integration тестване , при който първо се тества най-високия в йерархита компонент, като по-долните модули се заместват с stubs. На слеващото ниво , вече тестване копмоненти от по-горното ниво се използват като готови. Това се повтаря , докато и най-ниските в йерархията компоненти са тествани.
При този подход потецилани проблеми в дизайна се откриват по-рано.

**Bottom up**
Това е подход при integration тестване , при който първо се тества най-ниските в йерархията компоненти, след което се изпозлват, за да се подпомогне тестването на по-горните компоненти.
Грешките в дизайна могат да бъдат открити късно . Не е необходимо обаче ,голям брой компонети да бъдат готови ,за да се тества , за разлика от Big-Bang

**Functional**
Тестване базирано на анализ на спецификацията за функционалността на даден компоневнт или система. Този тип тестване е комбинация от двата предходни подхода. Това е подходът , който най-често се използва в практиката и който дава най-добри резултати.



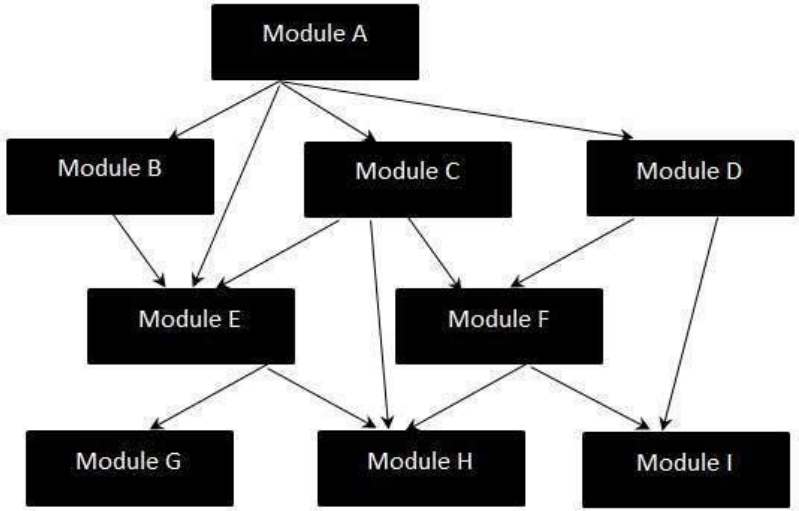
# Подходи тестове в интеграция

**Не-инкрементални**

Big bang

Това е подход при integration тестването, при който отделните компоненти (софтуерни и хардуерни) се комбинират едновремнно за изграждането на цялата система, вместо това да се извършва на етапи . Обикновено се използва при малки системи , защото има няколко значителни недостатъка:

- Локализирането на грешката е трудно
- Тестването може да започне само след като всички модули са готови
- Лесно е да бъдат пропуснати някои интерфейси или модули



# System testing

**Какво е системно тестване?**
**Процес по тестване на цялата система с цел да се провери, че тя отговаря на изискванията.**

**Обикновено системното тестване е финаланата фаза на “integration testing in the small” фазата – когато всички модули и подмодули са интегрирани заедно.**
**По време на системното тестване се извършват както функционални така и не-функционални тестове**

# System testing

**Функционално тестване(Functional testing)**

Функционалнато спецификация: документ, който описва детайлно характеристиките на продукта с фокус върху желаните й възможности.

Подходи към функционалното тестване:

Базирано на **функционалните изисквания**: целта на тестовете е да докаже или отхвърли хипотезата, че приложението отговаря на изискванията. При този подходът екипът, отговорен за извършване на тестовете изготвя своите планове и сценарии базирано на функционалната спецификация. Добре е тези сценарии да бъдат съгласувани с крайните потребители , за да се сведе до минимим риска от това сценариите да не отговарят изцяло , на това , което желае потребителя .

Базирано на **бизнес процесите**: при този вид тестване  сценариите са написани заедно с протебителите и отразяват максимално тяхната гледна точка. Такива сценарии се пишат в серия от срещи (интервюта) с потребителите . Сценарии получени по този метод се наричат „use cases” .

# System testing

**Не-функционално тестване (Non-functional testing)**

Това е процес на тестване на характеристики на системата , които не са пряко свързани с нейната функционалност , като например

- **Reliability testing** (надежност) – колко надеждно работи системата при определние условия
- **Usability testing** (използваемост) – колко лесно е да се използва системата . Интуитивен ли е интерфейсът. Как изглежда и какво е усещането да се работи със системата ( look&feel)
- **Efficiency testing** (ефикасност) – количеството код и тестове, необходими програмата да изпълни определена функция . целта е да се оцени какви ресурси са необходи, за да се достави дадена функционалност
- **Maintainability testing** (поддръжка) - целта е да се получи представа за това колко лесна е за поддръжка дадена система
- **Portability testing** (преносимост) - оценка на лекотата, с която една система може да се премести от една среда в друга. Например какво би било усилието едно мобилно приложение да работи на следващата версия на Андроид
- **Documentation testing** (документация) . Предмет на този вид тестване е всяка писмена форма на инфорация , свързана с проекта, включително процедури, резултати, спецификации. Тяхното ниско качество или липса могат да бъдат пагубни за успеха на проекта. Това е един от най- „cost-efficient" типове тестване, защото може да започне в самото начало на проекта ида открие дефекти още преди да са имплементирани.
- **Load testing** (натоварване) . Този вид тестване се провежда, за да се изучи поведението на системата при специфично очаквано натоварване – нормално и близко до пиковото.
- **Performance testing** (производителност) – целата е да се определи колко бързо работи система при определени условия
- **Compatibility testing** (съвместимост) – целта е да се определи съвместимостта на системата с различни други обекти : браузери, хардуер, операционни системи, потребители със специфични изсквания (разбиращи само един език например)
- **Security testing** (сигурност) – целта е да се провери до колко сигурна е сиситемата , колко устойчива е на хакерски атаки, сигурността на данните в системата
- **Stress testing** (стрес) целта е да се определи стабилността на системата . За това тя се тества в условия над нейните възможности . Акцентът е поставен на достъпността на системата, обработката на грешки при голямо натоварване, вместо на това как системата функционира в нормални условия
- **Recovery testing** (възстановяване) прави се , за да се провери колко бързо и колко добре системата може да се възстанови след инцидент , независимо хардуерен или друг тип.

# Acceptance testing/User acceptance testing

Това е последното ниво на тестване.  Това е процес на тестване базиран върху нуждите на потребителя , изискванията и бизнес процесите , за да се определи дали системата отговаря на условията за приемане на приложението от клиента.

Когато се прави подготовката за този вид тестване важни са следните артефакти:

- Изисквания (User and System requirements)
- Бизнес процеси
- Анализи за риска
- Use cases

В планирането на този вид тестване влиза нуждата от компютри, сървъри, бази данни и данни, както и среда, която да е макисмално близка до LIVE/PROD средата.  Едно от големите предизвикателства е да се намерят достатъчно опитни потребители, които да могат да извършват този вид тестване. Те трябва да са въвлечени  по някакъв начин в разработката на продукта. Целта е да се определи дали системата отговаря на всички изисквания, за да могат потребителите да я „приемат“ преди нейното внедряване.

# Alfa and Beta testing


| Alpha testing | Beta testing |
|---|---|
| Извършва се от тестери , които са служители на организацията , която разработва продукта | Извършва се от клиентите или от крайните потребители, които не са служители на компанията |
| Извършва се в самата организация | Извършва се извън организацията |
| Тестване за сигурност и надежност не се правят | Правят се тестове за сигурност, надежност |
| Използват и двете техники White box и black box | Използва се само Black box тестване |
| Критични дефекти могат да бъдат адресирани веднага от екипа на разработчиците | Повечето дефекти открити по време на Бета тестването се събират и се осраняват във следващата версия на продукта |

# Типове тестове
# (класификация спрямо целта на тестването)

**Спрямо целта на тестовете различваме 4 големи групи тестове**

**Тестване на функционалност (functional testing)**

Функционалността, която някоя система, под-система или компонент трябва да изпълни, може да бъде описана в артефакти като: функционална спецификация, use cases, или тя може да остане недокументирана . Функцията е „**какво**“ системата прави.
Функционалните тестове се базират на тези функции и могат да бъдат правени на всяко ниво ( Unit, Integration, System, Acceptance) . Най-често използваната техника при функционалните тестове е Black box

**Тестване на характеристики на продукта ( non-functional testing)**

Не-функционалните аспекти на системата са познати още като аспкети на качеството. Тук фокусът е не „какво“ прави системата, а „**колко“** добре го прави. Не-функционални тестове : **usability, security, load , performance** и тн .
**Техниките**, които се използват тук са малко формализирани и не толкова процедурни, колкото при функционалните тестове. Това е така защото пряко зависят от типа на системата.

**Тестване на структурата/архитектирата**

При този вид тестване се изследва вътрешната структура на приложението, за да се формулират тестовете. Изполваната техника се нарича White box или още позната като Glass box. Тези тестове изискват задълбочено познание на структурата и архитектурата на приложението или „**как“** работи приложението. Този тип тестване се концетрира върху самия код и изисква неговото задълбочено познание .Такива тестове могат да се правят на всички нива, но най-често приложими са при Unit тестовете.

# Типове тестове
# (класификация спрямо целта на тестването)

**Тестване свързано с промени (confirmation & regression testing)**

**Confirmation testing**
Целта на тестовете е да откриват грешки и дефекти в системата . Веднъж открити тези грешки, ще бъдат регистрирани  и след това отсранени от разработчиците. След като грешка или набор от грешки бъдат остранени, системата ще има нова версия , която съдържа тези поправки.  Веднага след като това се случи, нов тест трябва да се извърши, за да се провери, че тази грешка е отсранена . Това се нарича **confirmation testing или  re-testing**.

**Самият процес по тестването е итеративен по природа**. Това означава ,че едни и същи тестове се повтарят многократно, **защото целта на тестването е не само да регистрира наличието на грешки, но и да се увери ,че те са правилно отстранени.** За това сценариите трябва да бъда повтаряеми. Няма нищо по-лошо за тестерите от това да намерят дефект и след това да не могат да го възпроизведат. За да се правят добри повтаряеми тестове са необходими 2 основни неща:

- **Стабилна входна точка**: тъй като средите, в които се изпълнява теста се менят постоянно , абсолютно задължително е да се опише състоянието на системата и средата преди началото на теста

- **Стабилни входни данни**:  какви са точно данните, които се използват и в какъв ред

# Типове тестове
# (класификация спрямо целта на тестването)

**Scenario1 – Създаване на нов потребител**:

1. Въведи потребителско име : rstoilova
2. Въведи парола: 12345
3. Натисни бутона Create

**Очакван резултат:**
1. Зарежда се Welcome page
2. Потребителят е създаден в базата данни

**Scenario2 – Създаване на нов потребител:**

**Pre condition**:
1. В базата не съществува потребител rstoilova

**Steps:**
1. Въведи потребителско име : rstoilova
2. Въведи парола: 12345
3. Натисни бутона Create

**Expected results:**
1. Зарежда се Welcome page
2. Потребителят е създаден в базата данни

**Scenario3 – Създаване на нов потребител:**

**Pre condition:**
1. Налични са потребителско име и парола, генерирани на случаен принцип

**Steps:**
1. Въведи потребителско име
2. Въведи парола
3. Натисни бутона Create

**Expected results:**
1. Зарежда се Welcome page
2. Потребителят е създаден в базата данни

# Типове тестове
# (класификация спрямо целта на тестването)

**Тестване свързано с промени (confirmation & regression testing)**

**Regression testing**

След като един дефект бъде остранен, неговото остраняване може да доведе до откриването на някой скрит дефект или сам по себе си да доведе до дефекти .
Regression testing е вид re-testing в преди това тествана система след нейното модифициране с цел да се удостовери, че не са се проявили скрити дефекти или че нови дефекти не са били направени като резултат от промяната. Правят се когато **средата** или кодът се **променя**.
**Regression тестовете са едни от най-добрите кандидати за автоматизиране**. Тези тестове ще се използват много пъти не само при нова версия на продукта, но и когато се правят малки промени в системата. Всеки трябва да имат за цел и да може да бъде повтарян многократно.
**Use cases** са най-добрата основа за дизайн на regression тестове: как те използват системата най-често и кои са най-критичните бизнес процеси покрити от нея.

Regression:
"when you fix one bug, you
introduce several newer bugs."