Изпитна тема № 14: Софтуерно инженерство

1. Етапи в разработката на софтуер

1. Описва и обяснява етапите в разработката на софтуер. Общо: 6т.

Етапите в разработката на софтуер се разделят на няколко вида:

- Планиране
- Анализ
- Дизайн
- Разработка & имплементация
- Тестване
- Инсталация и поддръжка

Планирането включва събиране на всяко едно изискване, организиране на срещи между заинтересованите страни и предвиждане на възможните проблеми, които биха могли да възникнат. Разработчиците трябва да се запознаят изключително добре с изискванията на клиента. В противен случай шансовете за допускане на грешка се увеличават.

Анализирането е процес, при който е необходимо екипът да дефинира детайлно целия проект, проверявайки неговата приложимост. Работният процес се разделя на по-малки задачи, за да може разработчиците, тестърите, дизайнерите и мениджърите на проекта да дадат обратна връзка (оценка) за своята работа. Те имат за цел да преценят дали конкретната задача е приложима спрямо цена, време, функционалност, надеждност и др.

Дизайнът на софтуера се дефинира на база вече събраните изисквания от фазата „Планиране“. В този етап се определят всички системни и хардуерни изисквания за проекта. Също така се създава план за тестване, а именно кое да се тества, как да се тества и т. н. Дизайнът съдържа цялостната софтуерна архитектура на продукта, както и на базата от данни.

Разработката е най-дългият етап в софтуерната разработка. След приключване на етапа „Дизайн“ всеки разработчик получава собствени задачи, които е необходимо да изпълни в определен срок. Този срок се определя от това каква методология за разработване на софтуер се използва в екипа, колко време е предоставено на разработчиците и др.

Тестването е един от последните и изключително важни етапи. Когато разработчиците са готови със своите задачи и проектът е разработен, той се подлага на тестване, преглеждане за евентуални бъгове (грешки), проблеми и тяхното разрешаване. Софтуерът не преминава в следващия етап на разработка, ако тестърите не го одобрят.

Инсталацията и поддръжката са последния етап от разработката на софтуер. При него софтуерът се предоставя на клиентите, за да се инсталира на техните устройства. От там той преминава в етап на поддръжка, поправят се възникнали грешки (ако има такива) и се обновява периодично.

2. Методологии за разработка на софтуер

2. Посочва и различава методологии за разработка на софтуер. Прави заключения и изводи за значението на методологии за разработка на софтуер. Общо: 14т.

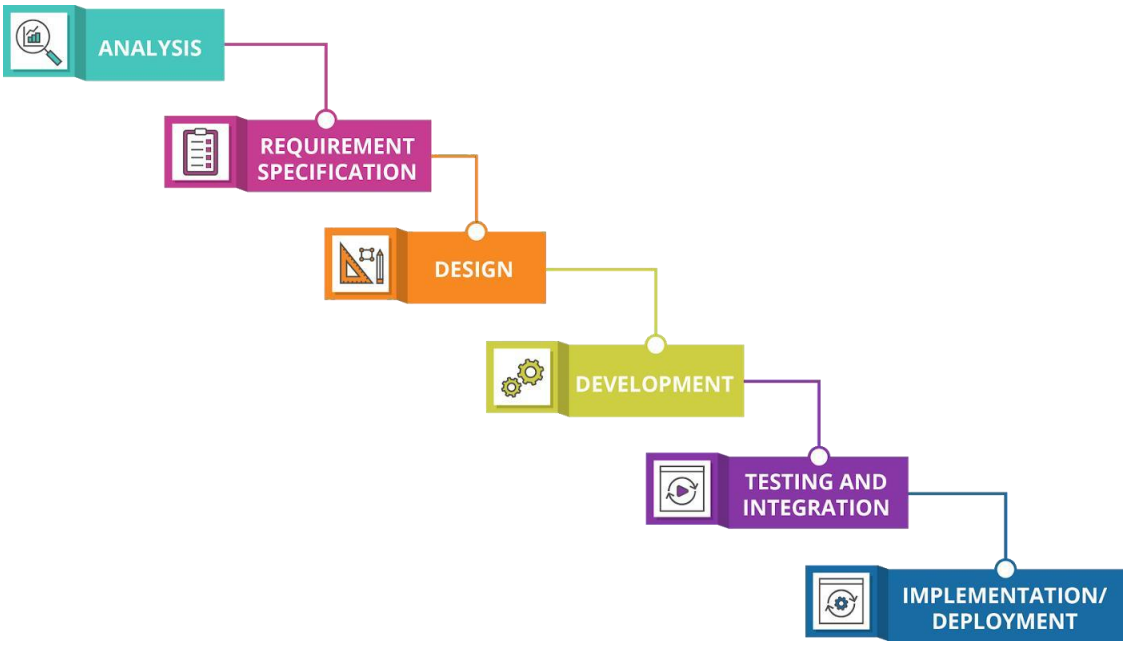
2.1 Посочва методологии за разработка на софтуер. 2т.

2.2 Различава методологии за разработка на софтуер. 4т.

2.3 Прави заключения и изводи за значението на методологии за РС. 8т.

- **Waterfall**

Тази методология представлява линеен подход за управление на проекти. Наречен е така, защото всяка фаза на проекта преминава каскадно в следващата, следвайки линейно като водопад. Това е задълбочена, структурирана методология, която съществува от дълго време. Като пример жизненият цикъл на разработка на софтуер с водопад или с SDLC (Software Development Life Cycle) с водопад се използва широко за управление на проекти за софтуерно инженерство. При тази методология всеки етап трябва да бъде приключен преди започване на следващия. Не съществува припокриване между етапите, тоест резултатът на даден етап играе ролята на начало за следващия. Състои се от вече споменатите етапи за разработка на софтуер.



Предимствата на тази методология са, че тя е лесна за разбиране и използване, както и за менажиране, поради устойчивостта на модела. Етапите се обработват един по един, като те са ясно дефинирани. Задачите са лесни за разпределяне, а процесът и резултатите са добре документирани.

Минусите от друга страна са, че няма работещ софтуер до самия край на последния етап. Освен това съществува голяма доза риск и несигурност. Този тип методология не е подходящ за комплексни, обектно-ориентирани и продължителни проекти. Не е добър модел за проекти, чиито условия биха се изменили в бъдеще. При него е трудно да се измери прогрес в контекста на етапите.

- **Agile**

Тази методология (наричана още „гъвкава методология“) е начин за управление на проект чрез разделянето му на няколко фази. Това включва постоянно сътрудничество със заинтересованите страни

и непрекъснато подобрение на всеки етап. След като работата започне екипите преминават през процес на планиране, изпълнение и оценка. Непрекъснатото сътрудничество е жизненоважно както с членовете, така и със заинтересованите страни по проекта. Agile е комбинация от итеративни и инкрементални модели за разработка, с фокус върху задоволяване на желанията на клиента чрез бърза доставка на работещи сегменти от продукта. Продуктът се разбива на малки инкрементални части, които се снабдяват в итерации. Всяка итерация трае средно между 1 и 3 седмици, като включва вече споменатите етапи за разработка на софтуер. След всяка итерация работещият продукт се показва на клиента и заинтересованите страни. Лесно се реагира при промени и се характеризира с адаптивност.

Предимствата при тази методология са лесното й управление, промотирането на екипна работа, възпроизвеждането на бърза функционалност и съответно демонстриране след кратък срок. Съществуват минимални ресурсни изисквания. Поради това тя е подходяща както за непроменящи се, така и за често променящи се изисквания. Позволява конкурентна разработка и започването на работа с малко или с почти никакво предварително планиране.

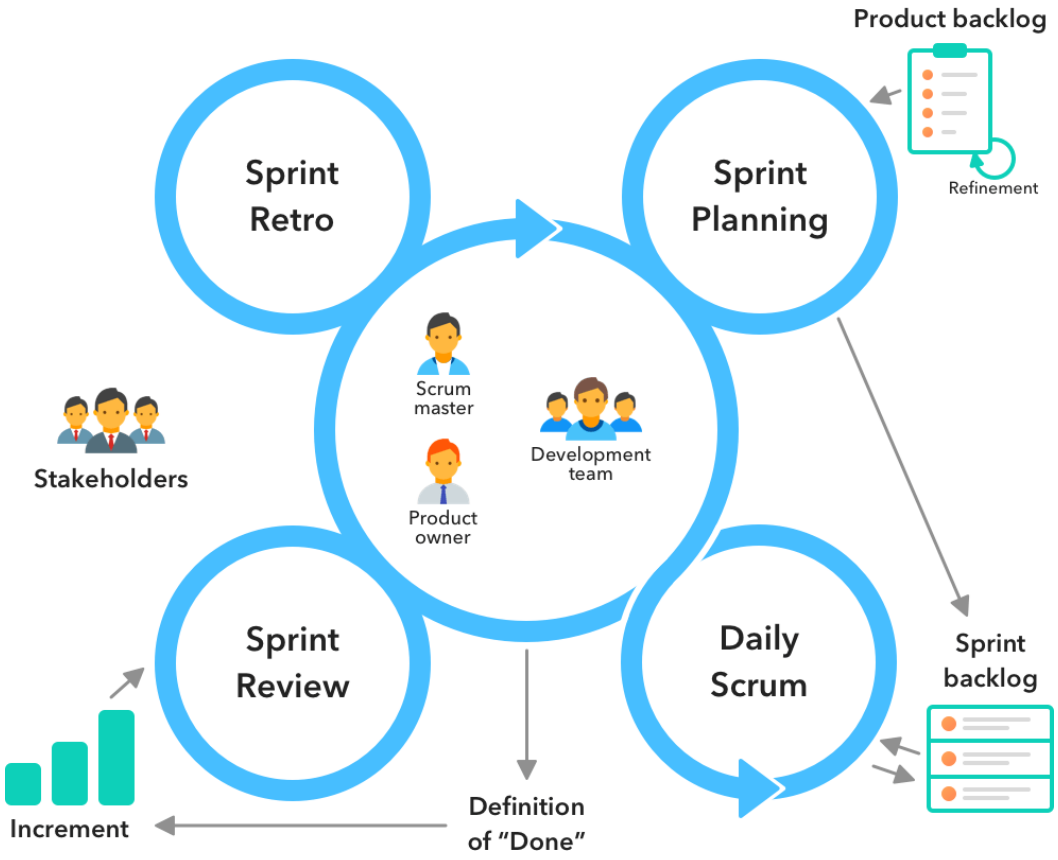
Недостатъците основно са доверяването на интеракцията с клиента. Ако клиентът не е наясно с изискванията си, екипът може да бъде подведен в грешна посока. Изключително се разчита на индивидуални личности от екипа, тъй като липсва или съществува минимална документация. За нови разработчици би било доста трудно да навлязат в екипа поради липсата на документацията.

- **SCRUM**

SCRUM е гъвкава методология за разработка, използвана при разработването на софтуер, базирана на итеративни и постепенни процеси. Тя е бърза, адаптивна и ефективна рамка, която е проектирана да доставя стойност на клиента по време на развитието на проекта. Основната й цел е да задоволи нуждите на клиента чрез колективна отговорност и непрекъснат напредък. Разработката започва от обща идея за това какво трябва да се изгради, създавайки се списък от характеристики, подредени по приоритет, които собственикът на продукта иска да получи. Фокусира се върху това как да се управляват задачи в среда за разработка, базирана на екипна работа. Тази методология използва итеративен и инкрементален подход, като съществуват кратки периоди на итерациите. При нея има сравнително лесна

имплементация. Състои се от SCRUM екипи и техните роли, събития, артефакти и правила. Всеки компонент във фреймуърка има специфично значение.

SCRUM спринтовете са събития с фиксирана продължителност от един месец или по-малко, за да се създаде последователност. Нов спринт започва веднага след приключването на предишния. През този период се разработва продукт за потенциално пускане в работа (release). Той се състои от спринт планиране, дневни SCRUM срещи, работата по разработването на софтуер, ревю на спринта и неговата



ретроспекция. По време на спринт планирането се планира работата, която трябва да се свърши, като това се случва в сътрудничество със SCRUM екипа. Срещите са ежедневно 15-минутно събитие, в което SCRUM екипа синхронизира работата си и изгражда план за деня. Ревюто се провежда на края на спринта, за да се инспектира новосъздадената функционалност при и нужда да се променят изискванията по продукта. Спринт ретроспекцията се състои веднага след спринт ревюто и преди започването на следващия спринт. SCRUM екипът анализира работата си и изгражда план за подобрение на ефективността си.

3. SCRUM – артефакти, събития и роли

3. Свързва SCRUM артефакти, събития и роли. Обобщава и прави изводи за SCRUM артефакти, събития и роли. Общо: 18т.

3.1 Свързва SCRUM артефакти, събития и роли. 4т.

3.2 Обобщава SCRUM артефакти, събития и роли. 6т.

3.3 Прави изводи за SCRUM артефакти, събития и роли. 8т.

- **SCRUM артефакти**

Артефактите на SCRUM представят работа или стойност (value). Те са проектирани за да направят всяка ключова информация максимално прозрачна. Така всички, които ги инспектират, имат едно и също осн ование за адаптация. Те предоставят ключова информация, която екипът и заинтересованите страни трябва да знаят, за да разберат продукта в процес на разработка, планираните дейности и вече извършените такива в проекта.
Всеки артефакт съдържа ангажимент, за да осигури предоставянето на информация, която увеличава пр озрачността и устрема, спрямо които напредъкът може да бъде измерван за:

- Product Vision (дефинира дългосрочната цел на продукта)
- Sprint Goal (целта за провеждане и фокусиране на спринта, като се внедри прогнозиране)
- Product Backlog (списък с всички неща, които се изискват в продукта).
- Sprint Backlog (набор от артикули на Product Backlog-а, избрани за спринта, както и план за доставяне на продуктово увеличение и реализиране на целта на спринта).
- Definition of Done (споделено разбиране на екипа за значението на това работата да бъде завършена)
- Increment (сумата от всички елементи на Product Backlog, завършени по време на спринт и всички предишни спринтове).
- Burn-Down Chart (графики, които дават общ преглед на напредъка във времето при завършване на проект; когато задачите са изпълнени, графикът „изгаря“ до нула)

и много други.

Тези ангажименти съществуват, за да укрепват емпиризма и SCRUM ценностите за SCRUM екипа и негов ите заинтересовани лица.

- **SCRUM събития**

Това са времево-регламентирани (time-boxed) събития, в които екипът изпълнява определен набор от задачи, които са част от цялостния проект, наричани още спринтове, които споменахме по-горе.

- **SCRUM роли**

Product owner, SCRUM master и екипът за разработка. Това са трите роли, предписани от SCRUM рамката. Заедно тези роли съставят SCRUM екипа, който споделя отговорността за управлението и изпълнението на работата в рамките на спринтовете.

- **ScrumMaster** – отговорен за:
  - Процесът да върви гладко
  - Да премахва пречките, които намаляват продуктивността
  - Да организира срещите от ключова важност

- **Собственик на продукта** – отговорен за:
  - Изискванията да бъдат разбираеми за всички
  - Да подреди в подходящ ред задачите и изискванията, за да се постигне най-добър резултат

- Да подсигури видимостта и яснотата на изискванията, като показва на екипа по какво ще работи в бъдеще
- Подсигурява, че екипът разбира условията на достатъчно добро ниво

➢ **Екип**

- Екипът се самоорганизира
- Хора от различни сфери на дадена организация работят заедно в екипа
- Достатъчно малък, за да остане гъвкав и подвижен
- Достатъчно голям, за да свърши достатъчно работа по време на спринт (6 - 9 души)

4. Софтуерна документация

4. Описва и обобщава съдържанието на софтуерната документация. Общо: 20т.

4.1 Описва съдържанието на софтуерната документация. 4т.

4.2 Обобщава съдържанието на софтуерната документация. 8т.

Софтуерната документация е част от всеки софтуер. Добрите практики за документиране са важни за успеха на софтуера. Документацията трябва да включва интерактивно потребителско изживяване, информационна архитектура и добро разбиране на вашата аудитория. Препоръчително е да се вгради доставената документация в процеса на разработка, докато се опитваме да използваме Agile методологиите за разработка на софтуер. Тя трябва да служи за разрешаване на проблеми, ако с тях се сблъскат разработчикът, крайният потребител и др. Подходящите подробности и описание трябва да бъдат в документацията, за да се постигнат следните цели:

- Разрешаване на проблема, възникнал от разработчика по време на процеса на разработка
- Помагане на клиентите и екипа за поддръжка да намерят информация
- Помагане на крайния потребител да разбере продукта

Документацията може да бъде свързана с API документация (която може да се използва или за включване в кода, или за разширяване на функционалността на съществуващото приложение, бележки за изданието, които обслужват кои грешки са били коригирани в текущата версия и какъв код е бил пречупен) или насочено към клиентите помощно съдържание за лесно намиране на необходимата информация незабавно. Софтуерната документация ви помага да разберете продукта, интерфейса, възможностите, възможността за изпълнение на задача и бързо търсене и намиране на конкретен раздел в документа или намиране на решение, когато се натъкнете на продукта. Изискват се и се доставят много типове документи по време на жизнения цикъл на разработка на продукта и жизнения цикъл на разработка на софтуер, като софтуерна документация, документация за разработчици, документ за софтуерни изисквания и проектна документация и анализ на аудиторията.

User Documentation

Този документ се доставя най-вече за краен потребител, който действително иска сам да използва продукта, за да разбере и изпълни определена задача.

• Ръководства с инструкции – Насочва потребителя да изпълни задача или предварително определена цел.

• Уроци – Научава концепция, като следва поредица от стъпки

• Референтни документи – Описва техническите детайли на продукта (спецификация на софтуерните изисквания, документи за проектиране на софтуер и т.н.)

• Ръководство за администриране: Позволява на администратора да се обърне към това след инсталиране на приложение

• Ръководство за конфигуриране: Позволява на администратора да препраща към този документ за конфигурационни параметри.

Developer Documentation

Тази документация се отнася до документация, свързана със системата.

• API документация – Указва как да се извикват API повиквания и класове или как да се включи API в кода, който се разработва.

• Бележки по изданието: Описва най-новия софтуер, изданията на функциите и кои грешки са коригирани. Обикновено този документ е текстов файл с файлово разширение (.txt).

• README: Форма на документация, обикновено е прост файл с обикновен текст, наречен Read Me, READ.ME, README.TXT с общ преглед на софтуера на високо ниво, обикновено заедно с изходния код.

• Системна документация – Описва системните изисквания, включва проектни документи и UML диаграми.

Just-in-time Documentation

Може да възникне ситуация, при която документ точно навреме бързо обслужва поддръжката за документация, насочена към клиента. Не е необходимо потребителят да се препраща към документи или често задавани въпроси за информация.

4.3 Прави изводи и заключения за значението на софтуерната документация в работатана програмистите. 8т.

Препоръчително е инструментите за документиране да бъдат общи за целия екип за разработка, така че да могат да бъдат лесно достъпни в средата и трябва да бъде инициирано документацията да стане задължителна част от процеса на жизнения цикъл на разработката на софтуер. Например GitHub е „cloud-based do my papers“ апликация, коя служи за разработчиците и авторите на кодове.

5. Случаи на употреба(use cases) и потребителски истории (user stories)

5. Различава случаи на употреба и потребителски истории. Общо: 4т.

- **Use Case**

Това е писмено описание на това как потребителите ще изпълняват задачи на вашия уебсайт. Той очертава, от гледна точка на потребителя, поведението на системата, докато тя отговаря на заявка. Всеки случай на употреба е представен като поредица от прости стъпки, започващи с целта на потребителя и завършващи, когато тази цел бъде изпълнена. В зависимост от това колко задълбочени и сложни са случаите, те описват комбинация от следните елементи:

Actor – всеки или всяко нещо, което извършва поведение (който използва системата).

Stakeholder – някой или нещо с лични интереси в поведението на обсъжданата система (SUD – system under discussion).

Primary actor – заинтересована страна, която инициира взаимодействие със системата за постигане на цел.

Predictions – какво трябва да е вярно или да се случи преди или след изпълнението.

Triggers – това е събитие, което предизвиква стартирането на use cases.

Main success scenarios – случай, при който нищо не се обърква.

Alternative paths – вариация на основната тема. Тези изключения се случват, когато нещо се обърква на системно ниво.

Предимства на use case:

Помагат да се обясни как трябва да се държи системата и в процеса, помагат да се предвиди какво може да се обърка. Те предоставят списък с цели и този списък може да се използва за установяване на цената и сложността на системата. След това екипите на проекта могат да преговарят кои функции стават изисквания и се изграждат. Ясни и подробни са, а предварителното проучване може да послужи в дългосрочен план.

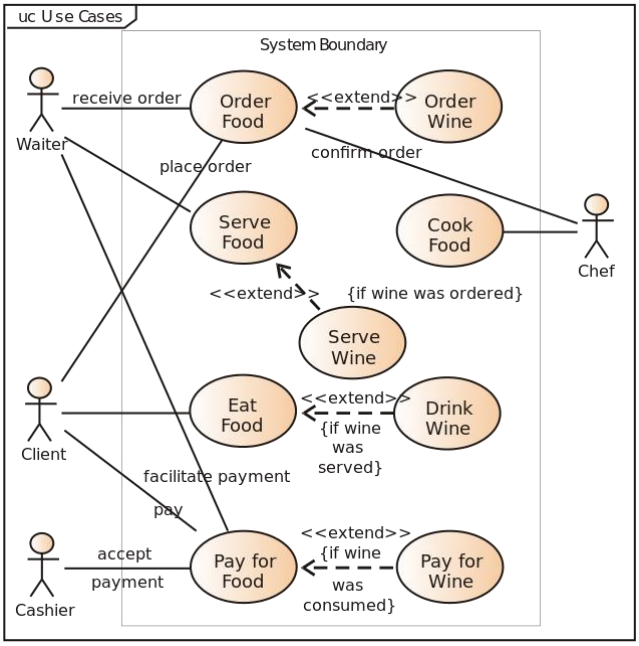
Недостатъци на use case:

При тях е необходима много поддръжка, не винаги са подходящи за agile разработка на софтуер.

Примери за писане на use case:

Simple Laundry Use Case

Use Case 1	Do laundry
Actor	Housekeeper
Basic Flow	On Wednesdays, the housekeeper reports to the laundry room. She sorts the laundry that is there. Then she washes each load. She dries each load. She folds the items that need folding. She irons and hangs the items that are wrinkled. She throws away any laundry item that is irrevocably shrunken, soiled or scorched.



- **User story**

Това е неформално, общо обяснение на софтуерна функция, написано от гледна точка на крайния потребител. Целта му е да формулира как дадена софтуерна функция ще предостави стойност на клиента. Потребителските истории са един от основните компоненти на една гъвкава програма. Те помагат да се осигури фокусирана върху потребителя рамка за ежедневна работа - която стимулира сътрудничеството, креативността и по-добрия продукт като цяло. Потребителските истории са няколко изречения на прост език, които очертават желания резултат. Не навлизат в подробности. Изискванията се добавят по-късно, след като бъдат съгласувани от екипа.

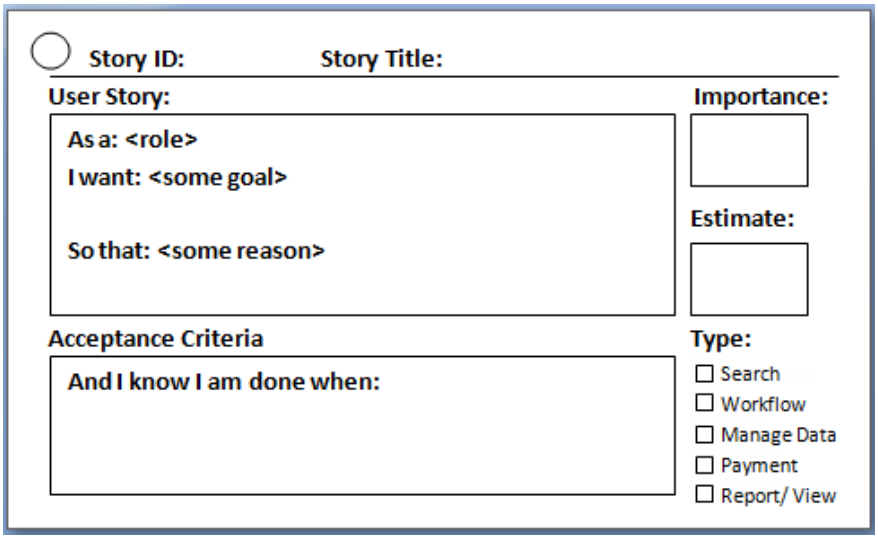
Предимства на user story:

Поддържат фокуса върху потребителя. Списъкът със задачи държи екипа фокусиран върху задачи, които трябва да бъдат отметнати, но колекцията от истории държи екипа фокусиран върху решаването на проблеми за реални потребители. Позволяват сътрудничество. С определената крайна цел екипът може да работи заедно, за да реши как най-добре да обслужва потребителя и да постигне тази цел. Водят до творчески решения. Историите насърчават екипа да мисли критично и креативно за това как най-добре да реши крайната цел. Те създават импулс. С всяка следваща история екипът за разработка се радва на малко предизвикателство и малка победа, движеща инерция. С две думи те са кратки, разбираеми за потребителите и разработчиците, не е нужна поддръжка и съответно не налагат особени усилия.

Недостатъци на user story:

При тях съществува вероятност данните да са непълни, отворени са за интерпретация и се използват в комбинация с критерии за приемане (условия за удовлетворяване).

Примери за писане на user story:



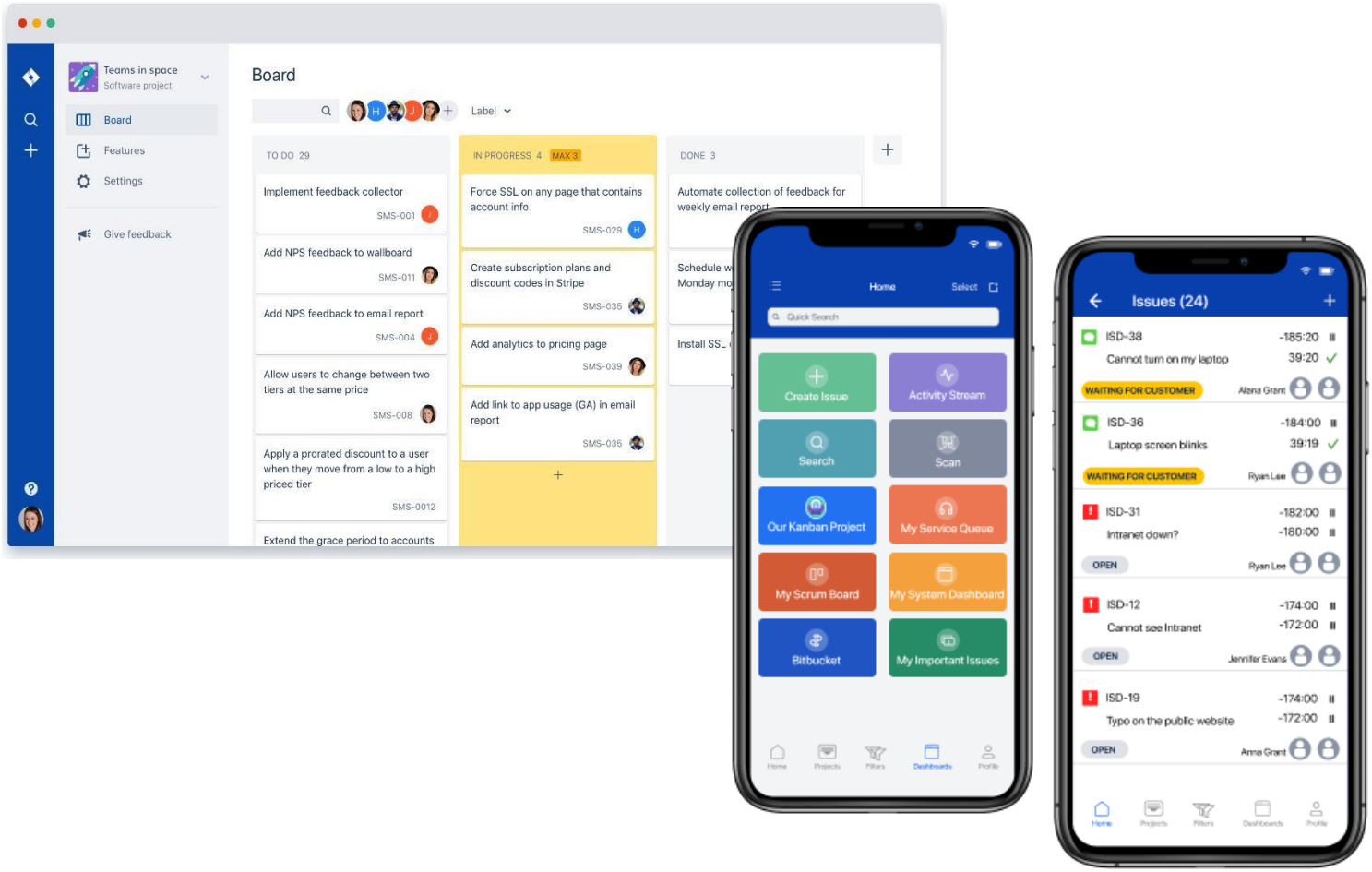
6. Инструменти за управление на екип

6. Познава и демонстрира инструменти за управление на работата на екип Общо: 8т.

Софтуерът за управление на екип подобрява сътрудничеството и координацията на работата, но също така повишава производителността и качеството на работа на дадения екип. Понякога използването на прости инструменти, като например приложения за проследяване на времето или табло със задачи, може да помогне много. Изборът на най-подходящия софтуер за управление на проекти е важна стъпка за ефективна работа във всеки проект, не само в инженерния. Всеки има свои любими методи и инструменти за управление на работните процеси. Докато един предпочита календари и електронни таблици, друг събира набор от различни приложения. Третият е за опция за химикалка и тефтер от старата школа, а четвъртият комбинира всички тях. Пример за такъв тип софтуер може да бъде Jira. Той предлага любимия на всички Kanban външен вид и е създаден като един от инструментите за Agile управление на проекти. При първото влизане системата задава няколко въпроса относно експертизата с различни методологии и предпочитания и препоръчва най-подходящия шаблон. Интуитивният интерфейс улеснява максимално настройването на работното пространство за всеки разработчик и екипа. Разработчиците широко използват този инструмент. Помага им да имат ясна визия за напредъка, независимо от различните използвани методологии.

Jira, поддържа приложението за уеб и мобилни устройства, което улеснява мениджъра на екипа да знае състоянието и да действа при различни нива на сложност.

Друга най-предизвикателна работа в управлението на проекти е проследяването на крайните срокове. Сега всеки член на екипа е запознат с крайния срок и приоритетите на задачите. Това прави целия екип уведомен и може да завърши задачите навреме.



7. Работа със системи за сорс-контрол

7. Демонстрира команди за работа със система за сорс-контрол. Общо: 12т.

Version control, още познат като source control, е практика за проследяване и управление на промените в софтуерния код. Системите за контрол на версиите са софтуерни инструменти, които помагат на софтуерните екипи да управляват промените в изходния код с течение на времето. Тъй като средите за разработка се ускориха, системите за контрол на версиите помагат на софтуерните екипи да работят по-бързо и по-интелигентно. Те са особено полезни за екипите на DevOps, тъй като им помагат да намалят времето за разработка и да увеличат успешните внедрявания. Софтуерът за контрол на версиите следи всяка модификация на кода в специален вид база данни. Ако бъде направена грешка, разработчиците могат да върнат часовника назад и да сравнят по-ранни версии на кода, за да помогнат за отстраняването на грешката, като същевременно минимизират смущенията за всички членове на екипа. Git е специфична система за контрол на версиите с отворен код, което означава, че цялата кодова база и история са достъпни на компютъра на всеки разработчик, което позволява лесно разклоняване и сливане. GitHub е уеб базирана услуга за разполагане на софтуерни проекти и техни съвместни разработки върху отдалечен интернет сървър, която предлага базирана на облак услуга за хостинг на Git хранилище.

GitHub - най-често използвани команди:

- git status - извежда информация относно branch-а, в който се намираме, дали branch-ът съдържа последната версия на файловете, маркира ако има нещо, което трябва да се commit-не
- git add - обновява съществуващи файлове или добавя проследяване на нови
- git add - работи върху всички файлове
- git add <file_name> - Работи върху файл с посоченото име
- git commit - прави commit съдържащ в себе си променените файлове в даден момент от времето
- git commit –a - пропуска Staging фазата (изпуска git add командата) и директно прави commit
- git commit -a -m "Commit message" - флага -m задава съобщение за бъдещия commit
- git push / git push origin master - изпраща направените промени до източника (сървъра)
- git log - връща историята на commit-ите
- git log -1 - връща последния commit
- git log --oneline - връща всички commit-и форматирани на един ред
- git log –patch - връща всеки commit детайлно + git diff за всеки от тях

Стъпки с branch-ове:

- git checkout -b <branch_name> - създаване на нов клон
- git checkout <branch_name> - преминаване между клоновете
- git branch <branch_name> - създаване на нов клон без да се преминава на него
- git stash - позволява да се "избутат" настрана текущите промени, за да се върнем към чиста работната директория, полезно в случаите, в които искаме да сменим клона и да работим по друг аспект, но работата на текущия клон се е объркала (не искаме да я commit-ваме)
- git stash list - изкарва списък на stash-натите промени
- git stash pop - връща промените обратно в работната директория
- git merge <branch_name> - обединява няколко клона

8. Работа с чужд код. Преглед на чужд код (code review)

8. Демонстрира процеса на преглед на чужд код (code review). Общо: 4т.

Code Review, известен също като Peer Code Review, е актът на съзнателно и систематично събиране на колеги програмисти, за да проверяват кода на другия за грешки и многократно е доказано, че ускорява и рационализира процеса на разработка на софтуер, както малко други практики могат. Има инструменти и софтуер за партньорска проверка на кода, но самата концепция е важна за разбиране. Софтуерът е написан от човешки същества. Следователно софтуерът често е пълен с грешки. Но това, което не е толкова очевидно, е защо разработчиците на софтуер често разчитат на ръчно или автоматизирано тестване, за да проверят своя код за пренебрегване на този друг голям дар на човешката природа: способността сами да виждаме и коригираме грешките.

1. Тестване на кода

Един от начините да разберете кода е да създадете тестове за характеризиране и unit тестове. Можете също така да използвате инструмент за качество на кода — като анализатор на статичен код — върху вашия код, за да идентифицирате потенциални проблеми. Това ще ви помогне да разберете какво всъщност прави кодът. И ще разкрие всички потенциално проблемни области. След като разберете кода, можете да правите промени с по-голяма увереност.

2. Преглед на документацията

Прегледът на документацията за първоначалните изисквания ще ви помогне да разберете откъде идва кодът. Наличието на тази документация под ръка ще ви помогне да подобрите кода — без да компрометирате системата. Без тази информация можете случайно да направите промени, които въвеждат нежелано поведение.

3. Рефакториране

По-добре е да опитате да преработите наследството, вместо да го пренаписвате. И най-добре е да го правите постепенно. Рефакторингът е процес на промяна на структурата на кода — без промяна на неговата функционалност. Това почиства кода и го прави по-лесен за разбиране. Освен това елиминира потенциални грешки.

4. Сътрудничете с други разработчици

Може да не познавате кодовата база много добре. Но някои от вашите колеги разработчици вероятно го правят. Много по-бързо е да задавате въпроси на тези, които най-добре познават кодовата база. Така че, ако е възможно, сътрудничете с някой, който го знае по-добре от вас. Втори поглед върху кода може да ви помогне да го разберете по-добре.

5. Поддържайте новия код чист

Има начин да не правите кода по-проблематичен. И това е като се уверите, че новият код е чист. Трябва да бъде написано, за да се придържа към най-добрите практики. Не можете да контролирате качеството на наследения код. Но можете да се уверите, че кодът, който добавяте, е чист.

6. Направете допълнителни изследвания

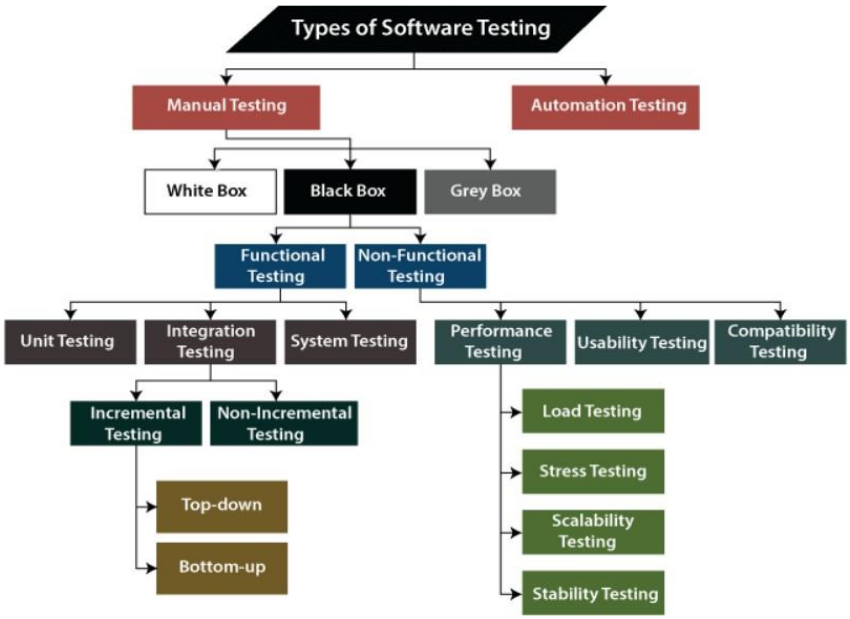
Работата с наследена кодова база става по-лесна с времето. Младши разработчик може да не разбере защо кодовата база не е преработена (и може да иска да я преработи). Но един старши разработчик ще знае кога да го остави на мира. Научаването на повече за кодовата база ще ви помогне да я подобрите.

9. Видове софтуерно тестване

9.Описва и различава видове софтуерно тестване. Обобщава и диференцира употребата на различните видове софтуерно тестване. Общо: 14т.

Софтуерно тестване

Софтуерното тестване е процес на изследване и проучване на софтуер с цел получаване на информация за качеството на продукта и услугата, която се изпитва. Процесите на софтуерното тестване са неразделна част от софтуерното инженерство и осигуряване на качеството на софтуера. Категоризирането на софтуерното тестване е част от различни тестови дейности, като тестова стратегия, тестови резултати, дефинирана тестова цел и т.н. Целта на наличието на тип тестване е да се потвърди AUT (application under test). За да започнем тестването, трябва да разполагаме с изискване, готово за приложение, необходими ресурси. За да поддържаме отчетност, трябва да възложим съответен



модул на различни тестови инженери. Тестването на софтуера е разделено главно на две части, които са както следва:

- Manual Testing (ръчно тестване)
- Automation Testing (автоматизирано тестване)

9.1 Описва видовете софтуерно тестване. 2т.

9.2 Различава видове софтуерно тестване. 4т.

Какво е ръчно тестване?

Тестването на всеки софтуер или приложение според нуждите на клиента без използване на инструмент за автоматизация е известно като ръчно тестване. С други думи, можем да кажем, че това е процедура на проверка и валидиране. Ръчното тестване се използва за проверка на поведението на приложение или софтуер в противоречие със спецификацията на изискванията. Ръчното тестване може допълнително да се класифицира в три различни типа тестване, които са както следва:

- White Box Testing - разработчикът ще инспектира всеки ред код, преди да го предаде на екипа за тестване или на съответните инженери за тестване. Впоследствие кодът е забележим за разработчиците по време на тестването. От там идва и наименованието WBT.
- Black Box Testing - процес на проверка на функционалността на приложение според изискванията на клиента. Изходният код не се вижда при това тестване. Затова се нарича BBT.
- Grey Box Testing – комбинация от WBT и BBT.

Какво е автоматизирано тестване?

Най-важната част от тестването на софтуера е автоматизираното тестване. То използва специфични инструменти за автоматизиране на тестови случаи за ръчен дизайн без човешка намеса. Автоматизираното тестване е най-добрият начин за подобряване на ефективността, производителността и покритието на софтуерното тестване. Използва се за повторно изпълнение на тестовите сценарии, които са изпълнени ръчно, бързо и многократно.

9.3 Обобщава и диференцира употребата на различните видове софтуерно тестване. 8т.

Unit Testing

Това е първото ниво на функционално тестване, за да се тества всеки софтуер. При това тестовият инженер ще тества модула на дадено приложение независимо или ще тества цялата функционалност на модула, което се нарича тестване на единици. Основната цел на изпълнението на тестването на модула е да се потвърдят компонентите на модула с тяхната производителност. Тук unit-а се дефинира като единична тествана функция на софтуер или приложение и се проверява през цялата определена фаза на разработка на приложение.

Integration Testing

След като успешно внедрим unit тестването, ще преминем към интеграционно тестване. Това е второто ниво на функционално тестване, при което тестваме потока от данни между зависимите модули или интерфейса между две функции, се нарича интеграционно тестване. Целта на изпълнението на интеграционното тестване е да се тества точността на израза между всеки модул. Интеграционното тестване също е допълнително разделено на Incremental и Non-Incremental testing. Винаги, когато има ясна връзка между модулите, ние преминаваме към поетапно тестване на интеграцията. Да

предположим, че вземаме два модула и анализираме потока от данни между тях дали работят добре или не. Ако тези модули работят добре, тогава можем да добавим още един модул и да тестваме отново и можем да продължим със същия процес, за да постигнем по-добри резултати. С други думи, можем да кажем, че постепенното добавяне на модулите и тестването на потока от данни между модулите е известно като Incremental Integration Testing. Има два вида Incremental Integration Testing:

- Top-down - При този подход ние ще добавяме модулите стъпка по стъпка или постепенно и ще тестваме потока от данни между тях. Трябва да сме сигурни, че модулите, които добавяме, са дъщерни на по-ранните.
- Bottom-up – При него ще добавяме модулите постепенно и ще проверяваме потока от данни между модулите. И също така се уверете, че модулът, който добавяме, е родител на по-ранните.

BigBang Integration Testing е добър подход за малки системи. Това тестване интегрира всички модули в един, като обаче отнема повече време, за да го направи.

Компонентно тестване

Тестването на компоненти, известно също като тестване на програми или модули, се извършва след unit тестване. При този тип тестване тези обекти могат да бъдат тествани независимо като компонент, без да се интегрират с други компоненти, напр. модули, класове, обекти и програми. Това тестване се извършва от екипа за разработка.

Регресивно тестване

Това е тип софтуерно тестване, където се уверяваме, че при въвеждането на промени не са настъпили дефекти в други компоненти на софтуерния продукт. Промените в приложението могат да предизвикат непреднамерени странични ефекти. Не е различно от пълното или частичното селектиране на вече проведени тестове, за да се уверим, че съществуващата функционалност продължава да работи.

Mocking тестване

Mocking тестването е метод, използван за изолиране на поведението на даден обект. Зависимостите се заменят с обекти, които имитират поведението на реалните такива. Основно се използва при unit тестване, като намалява сложността на тестовете и подобрява времето им за изпълнение.