Титулка

Завдання 1

Завдання 2

АНОТАЦІЯ

Кваліфікаційний проект на тему Розробка програмного комплексу “Smart House” для моніторингу приватної власності з використанням C++. Яковенка Ігоря Юрійовича. Тернопільський національний технічний університет імені Івана Пулюя, Факультет комп'ютерно–інформаційних систем і програмної інженерії, Кафедра програмної інженерії, група СПм–61, Тернопіль, 2021. С. – 73, рис. – 11, табл. – 3, слайдів. – __, додат. – 4, бібліогр. – 39.

Метою кваліфікаційного проекту є розробка системи так званого “Розумного будинку”, а саме системи для зручного спостереження за приватною власністю.

Пояснювальна записка складається з чотирьох розділів. В аналітичній частині описується обґрунтування актуальності теми роботи, постановка завдання, опис ключових варіантів використання та автоматизація процесу розробки. В теоретичній частині описується модель розробки програмного забезпечення, вибір сервера безперервної інтеграції та архітектура програмної системи. В практичній частині описано тестування та інтерфейс системи.

Основні питання охорони праці та техніки безпеки розглянуто в четвертому розділі.

Ключові слова: SMART HOUSE, КОМУНАЛЬНІ ПОСЛУГИ, КОНТРОЛЬ ВИТРАТ, ПРОГРАМНИЙ КОМПЛЕКС, КОНТРОЛЬ БУДИНКУ, ГЛОБАЛЬНА МЕРЕЖА, РОЗУМНИЙ ДІМ, КОМУНАЛЬНІ СЛУЖБИ, ENTITY–COMPONENT–SYSTEM, C++.

SUMMARY

Project on the development of software "Smart House" for monitoring private property using C ++. Yakovenko Ihor Yuriyovych. Ternopil Ivan Puluj National Technical University, Department of Computer Information Systems and Software Engineering, Department of Software Engineering, group SPm–51, Ternopil, 2021. C. – 73, Fig. – 11, table. – 3, slides. – __, add – 4, a ref. – 39

The aim of the diploma project is to develop a system of the so–called "Smart Home", namely a system for convenient monitoring of private property.

The explanatory note consists of four sections. The analytical part describes the rationale for the relevance of the topic of the diploma project, problem statement, description of key uses and automation of the development process. The theoretical part describes the model of software development, the choice of server continuous integration and software system architecture. The practical part describes the testing and interface of the system.

The main issues of labor protection and safety are discussed in the fourth section.

Keywords: SMART HOUSE, UTILITIES, COST CONTROL, SOFTWARE COMPLEX, HOUSE CONTROL, GLOBAL NETWORK, ENTITY–COMPONENT–SYSTEM, C++

ЗМІСТ

ЗМІСТ 6
ВСТУП 8
1. АНАЛІТИЧНА ЧАСТИНА 11
1.1. Аналіз предметної області 11
1.2. Постановка задачі 15
1.3. Опис ключових варіантів використання 16
1.4. Автоматизація процесу розробки 20
1.4.1. Методологія Agile 21
1.4.2. Життєвий цикл розробки програмного забезпечення 24
2. ТЕОРЕТИЧНА ЧАСТИНА 26
2.1. Водоспадна модель розробки програмного забезпечення 26
2.2. Вибір сервера безперервної інтеграції 29
2.3. Архітектура програмної системи 31
2.3.1. Entity–component–system 33
2.3.2. Сутності в Entity Systems 34
2.3.3. Компоненти 35
2.3.4. Системи 36
2.3.5. Складання моделі 37
2.4. Орієнтований на дані дизайн 40
2.4.1. Переваги дизайну орієнтованого на дані дизайну 42
3. ПРАКТИЧНА ЧАСТИНА 44
3.1. Архітектура тестів програмного забезпечення.. 43
3.2. Метод тестування чорної коробки 46
3.3. Рівні тестування 47
3.4. Unit Testing 48

3.5. xUnit Архітектура 49
3.6. План тестування.. 51
3.6.1. Порівняння фреймворків... 51
3.7. Розробка тестів з використанням Google Test фреймворка............................ 57
3.7.1. Google Test 60
3.8. Інтерфейс системи 60
4. ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ.............. 63
4.1. Охорона праці 63
4.2. Безпека в надзвичайних ситуаціях 66
ВИСНОВКИ.. 69
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 70
ДОДАТКИ………………………………………………………………………….. 73
ДОДАТОК А – Код програми
ДОДАТОК Б – Технічне завдання
ДОДАТОК В – Диск с програмою
ДОДАТОК Г – Наукова публікація

ВСТУП

В сьогоднішній вік, коли цифрові технології все більше інтегруються в навколишній світ людей, потрібно максимально використати потенціал пристроїв, які використовують майже всі громадяни України. Тому потрібно розробити бути система яка може зберігати, оцінювати та показувати стан будинку та допоможе користувачеві зручно моніторити за показниками та безпекою його будинку.

Моєю задачею буде реалізація, так званого "Розумного будинку", а саме Розробка програмного комплексу "Smart House" для моніторингу приватної власності з використанням С++, системи для зручного спостереження за витратами електроенергії, води та контролю безпеки в домі. Система буде вдосконалювати мікропроцесорну систему "Smart House" – прилад який складається з невеликого основного пристрою, до якого під'єднані датчики, та модуля який буде підключається до WIFI–роутера. Система буде транслювати данні житлової площі на веб сторінку. Крім того система буде контролювати датчиками воду та газ, та при аварійній ситуації буде перекривати відповідні клапани, для усунення проблеми.

Система буде корисна не тільки звичайним сім'ям, які хочуть швидко і легко контролювати свої витрати електроенергії, води та контролю безпеки а і людям, які здають квартиру в оренду, чи простим підприємцям, які не мають доступу до приміщення тому, що данні дані на сайті можна подивитися з любої точки планети, де є підключення до мережі інтернет.

Автоматизація будинку за останні роки значно змінилася. Це зв'язано з технологіями які швидко розвиваються, такі як Інтернет, мобільний зв'язок та відновлювані джерела енергії. Розробки стосуються всіх основних аспектів розумного будинку, таких як

– можливості домашньої інфраструктури та керованого пристрою

– зручність використання мобільних і стаціонарних інтерфейсів користувача;
– мотивація для інвестування в технології автоматизації та управління.

Донедавна автоматизація будинку була в основному зосереджена на встановленні керованих розеток або вимикачів світла та інфрачервоних елементів керування по всьому будинку. Та використовувала технології, розроблені на початку сімдесятих років минулого століття, які з сучасної точки зору є повільними, ненадійними і небезпечними.

Швидкий розвиток мобільного зв'язку вніс технологічний стрибок в автоматизації різних об’єктів. Бездротові мережі (3G, 4G, Wi–Fi) і розумні пристрої з інтерфейсами бездротового зв’язку (Bluetooth, ZigBee, Wi–Fi) є повсюди і дозволяють користувачеві перейти на новий рівень керування будинком. Замість того, щоб просто вмикати та вимикати розетки, можна використовувати функції побутової електроніки, побутових пристроїв або компонентів інфраструктури. В результаті, замість елементарної функціональності, сьогодні доступні можливості, які реально впливають на комфорт, безпеку та енергозбереження в житлових і промислових будівлях.

Крім того великі зміни та значні досягнення торкнулися до інтерфейсу користувача. Революція смартфонів і планшетів нарешті принесла до дому персональний універсальний пристрій дистанційного керування. Запатентовані, стаціонарні панелі та пристрої керування поступово виходять з використання, їх замінюють програми, які є простими в експлуатації, обслуговуванні та оновленні.

З покращенням зручності використання та можливостей мотивація встановлення домашньої інтелектуальної системи також стала ширшою. Бажання жити в “зеленому будинку”, який здатен значно знизити споживання енергії та води, нарешті стає реальним. Також набувають популярності нові програми для управління безпекою, та програми автоматизації побуту для людей похилого віку та інвалідів та дистанційного керування будівлею.

1. АНАЛІТИЧНА ЧАСТИНА

1.1. Аналіз предметної області

Темою даного проекту є розробка системи “Розумний дім”.

Розумний дім – це сукупність різноманітних модулів, кожен з яких має власне застосування і підходить для конкретного випадку. Грубо їх можна розділити на 5 підгруп: керуючі пристрої, керовані пристрої, датчики, шлюзи зв'язку та логічні пристрої.

Керуючі пристрої несуть на собі функції передачі команд розумному будинку. Через них проводиться контроль та стан приладів. Керовані прилади виконують команди будинку та передають її електроприладам. Датчики отримують інформацію з навколишнього світу, а шлюзи зв'язку підтримують зв'язок з керуючими пристроями, та з електроприладами, якими треба керувати не просто подавши напругу, а по якомусь протоколу(RS 485,RS 232,LAN,Wi–Fi чи інфрачервоний зв'язок).

З огляду на розподіл загального споживання енергії, частка приватного сектора є значною. У 27 країнах Європейського Союзу (ЄС–27) у 2010 році 27% загальної енергії споживалося житловим сектором, у США – 22%, що показано на рисунку 1.1. Таким чином, енергозбереження в будинках дійсно є важливим і з глобальної точки зору. А потенціал економії для всіх видів енергії, які використовуються в приватному секторі, великий. На опалення та охолодження приміщень припадає найбільша частка – від 50% до 70% загального споживання енергії будинками. На другому місці опалення води, далі йдуть електроприлади та освітлення, що показано на рисунку 1.2 [1, 2].

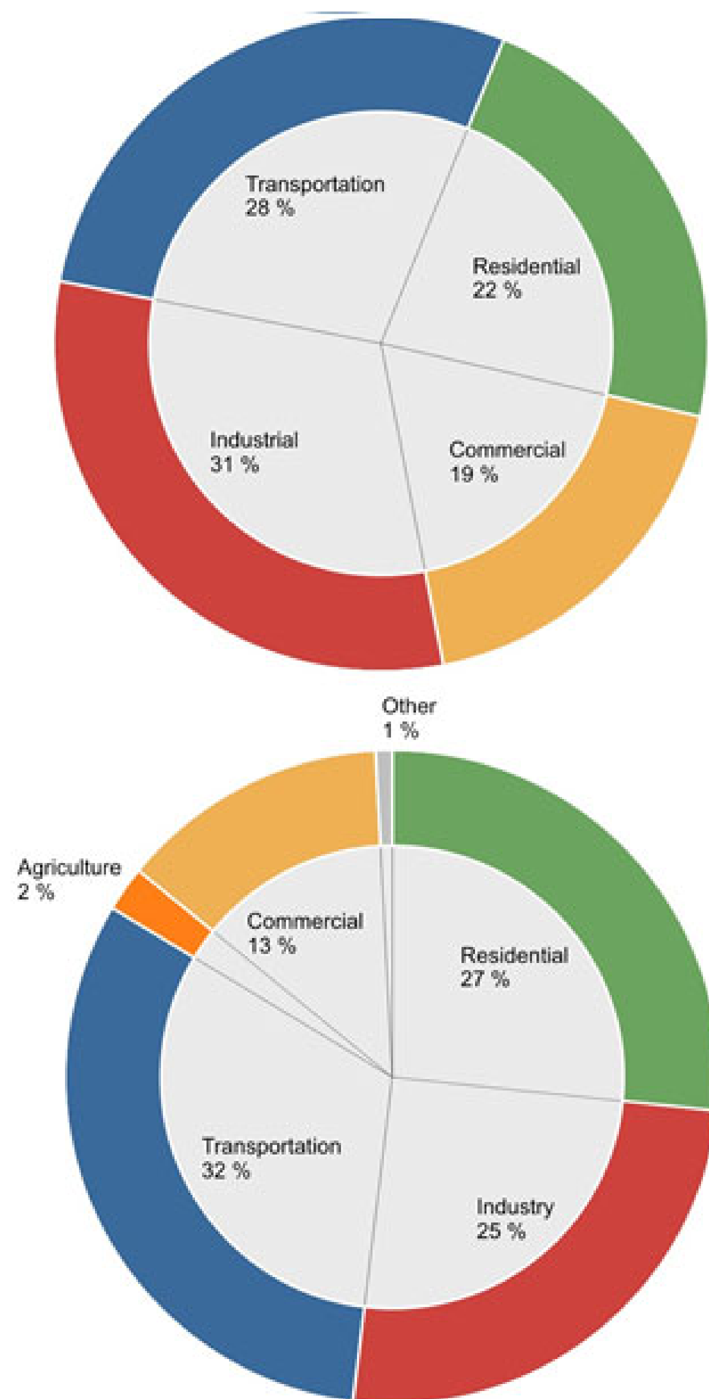

Рисунок 1.1 – Споживання енергії в США та ЄС–27 за сектором у 2010 р.

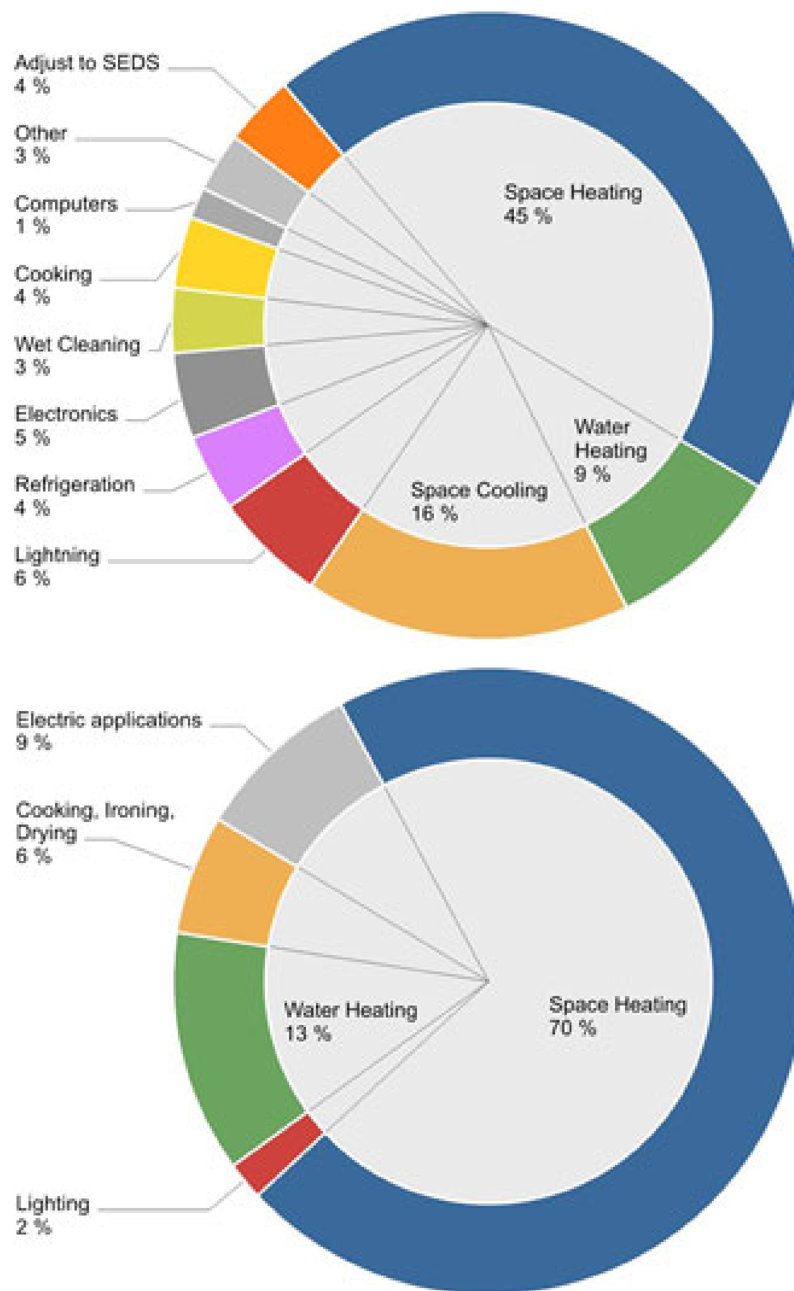

Рисунок 1.2 – Поділ кінцевого споживання енергії в житлових приміщеннях США, Німеччина, 2010 р.

Існує кілька підходів до зниження споживання енергії, на кожен з яких слід звернути увагу:

– утеплення будівлі;
– найсучасніша техніка;
– ефективні системи опалення води та опалення приміщень;
– автоматизація та контроль будівлі.

Завдяки досягненням у сфері автоматизації будинку, як описано вище, автоматизація будівель, стає все більш привабливим вибором, надаючи можливість значної економії при відносно низьких початкових інвестиціях. Розумні прилади координують свою роботу з розумними лічильниками, зменшуючи загальне споживання енергії та уникаючи пікових навантажень [3].

Нарешті, розумні будинки можуть бути інтегровані з розумними електромережами, які будуються по всьому світу, завдяки зростанню виробництва відновлюваної енергії. Розумні лічильники та розумні клапани можуть працювати лише за наявності інфраструктури керування та автоматизації будинку. Ця інфраструктура може взаємодіяти з витратами на електроенергію, обумовленою попитом і пропозицією в розумних електромережах. Виробництво відновлюваної енергії на основі вітру та сонця викликає значні коливання рівня енергії в електромережах комунальних підприємств. Таким чином, наприклад, може мати сенс охолоджувати морозильну камеру на два–три градуси нижче нормальної роботи під час сильного вітру, щоб вона могла залишатися вимкненою довше вдень із меншим енергопостачанням.

Завдяки можливості постійного моніторингу рівня енергії та цін у розумній електромережі, а також планування (відкладення або ранній запуск) високоенергетичних процесів, таких як

– котли;
– робота з посудомийною та пральною машинами;
– охолодження морозильної камери та холодильника.

Тому розумні лічильники можуть сприяти значній економії енергії, не впливаючи на рівень комфорту мешканців [4].

Ще один напрямок для найсучаснішої автоматизації будинку – це дистанційний контроль будівлі та управління безпекою з такими функціями, як

- Контроль порожнього будинку (температура, енергія, газ, вода, дим, вітер);
- Годування та спостереження за домашніми тваринами;
- Полив рослин у приміщенні та на вулиці;
- Симуляція присутності для запобігання зловмисникам;
- Допоміжні живі системи, що дозволяють людям похилого віку та людям з обмеженими можливостями залишатися вдома в безпеці за допомогою систем нагадування, видачі ліків, моніторингу артеріального тиску, пульсу та екстреного сповіщення.

Розумний будинок і автоматизація будівель пройшли довгий шлях. Технологічний прогрес, зміна клімату та демографічний перехід змістили статус розумного будинку як дорогого футуристичного житла до невід'ємної частини життя мільйонів. Стандартизація домашньої автоматизації на основі відкритих Інтернет–технологій і всюдисущих смартфонів, готових керувати світом, стала каталізатором для виробників, які почали інтегрувати функції керування в свої продукти за замовчуванням. Отже, (нарешті) є все, що потрібно для розумного дому [5].

1.2.Постановка задачі

Після вибору предметної області, перегляду всіх основних функцій і проблем житлового простору, а саме енергоефективність і безпека, потрібно оцінити завдання на даний проект. Потрібно відповісти на ряд запитань перед початком роботи:

- актуальність вибраної області;
- чи розглянуті всі питання і проблеми предметної області;
- опитування людей, які є частиною предметної області;
- чи правильно розставлені зв’язки між сутностями у розробленій ER–діаграмі;
- розставлення правильних атрибутів до кожної з сутностей з відповідними типами даних;
- які дані виводити для кожного з користувачів чи вони потрібні чи ними можна знехтувати;
- моделювання зручного інтерфейсу програми;
- збільшити швидкодію і зрозумілість програмного забезпечення;
- чи є зайві, непотрібні функції, якими можна знехтувати

В системі існує 2 основні сутностей: Користувач; Адміністратор.

Кожен з користувачів має свої функції для роботи. Загалом система виконує наступні запити і функції:

- вхід користувача в систему з введенням паролю;
- реєстрація;
- добавлення нового житла;
- перегляд інформації про комунальні послуги;
- перегляд інформації про вартість комунальних послуг;
- встановлення і налаштування системи “Розумний дім”

1.3. Опис ключових варіантів використання

Хмарні обчислення – це загальний пул обчислювальних ресурсів, готових надавати різноманітні обчислювальні послуги на різних рівнях, від базової

інфраструктури до найскладніших прикладних сервісів, які легко розподіляються та випускаються з мінімальними зусиллями або взаємодією постачальника послуг [6, 7]. На практиці сервер керує обчислювальними ресурсами, сховищами та комунікаційними ресурсами, які використовуються кількома користувачами у віртуальному та ізольованому середовищі.

Сервіс інтернет речей і розумний дім можуть скористатися широкими ресурсами та функціональними можливостями хмарних обчислень, щоб компенсувати обмеження в сховищі, обробці, зв'язку, збільшені швидкості вибірки, підтримати резервне копіювання та відновлення. Наприклад, сервер може підтримувати керування послугами інтернету речей і їх виконання, а також виконувати додаткові додатки, використовуючи дані, створені системою інтернет речей. Розумний дім може бути сконцентрований і зосереджений лише на основних і важливих функціях і таким чином мінімізувати ресурси локального пристрою та покладатися на можливості та ресурси хмарних обчислень. Розумний дім і сервіс інтернет речей будуть зосереджені на зборі даних, базовій обробці та передачі на сервер для подальшої обробки. Щоб впоратися з проблемами безпеки, сервер може бути приватним для високозахищених даних і загальнодоступним для решти даних. Розумний дім і хмарні обчислення – це не просто злиття технологій. А скоріше, баланс між локальними та центральними обчисленнями разом з оптимізацією споживання ресурсів. Обчислення може виконуватися на пристроях розумного дому, або передаватися на сервер. Де вартість обчислень, залежить від компромісів накладних витрат, доступності даних, залежності від даних, обсягу транспортування даних, залежності від комунікацій та міркувань безпеки. З одного боку, модель потрійних обчислень, що включає хмарні технології, інтернет речей і розумний дім, повинна мінімізувати всю вартість системи, як правило, з більшою увагою до зменшення споживання ресурсів вдома. З іншого боку, модель послуг інтернету речей та інтелектуального дому має покращити досвід користування інтернету речей, щоб користувачі задовольнили їхні потреби під час використання хмарних

додатків та вирішували складні проблеми, що виникають із нової моделі послуг інтернету речей, розумного дому та хмарних обчислень.

В таблиці 1.1 показано варіанти використання системи.

Таблиця 1.1 – Варіанти використання системи

№	Основний актор	Найменування	Короткий опис
1	Користувач	Вибір режиму роботи систем	Дозволяє налаштувати та вибрати режими роботи системи, які зберігають встановлені раніше налаштування систем
2	Користувач	Налаштування систем в будинку	Дозволяє налаштувати конкретну систему в комплексі
3	Користувач	Віддалене керування системами	Дозволяє отримати доступ до систем які курують будинком та пристроями в будинку
4	Користувач	Огляд параметрів та систем будинку	Відображає всю інформацію про стан будинку та пристроїв в будинку
5	Технічний спеціаліст	Меню технічного налаштування	Дозволяє виконувати технічне налаштування систем будинку

На рисунку 1.3 показана модель варіантів використання системи.

Моніторинг поточного та минулого споживання електроенергії та визначення профілів навантаження забезпечують основу для інтелектуального керування живленням із такими можливостями, як:

– Інтелектуальне управління опаленням шляхом автоматичного керування температурою в приміщенні на основі часу, зовнішньої температури та присутності;
– Розумна система освітлення, яка керує освітленням на основі виявлення присутності, часу сходу або заходу сонця та функції кімнати;

– Інтелектуальні, жалюзі, які зберігають прохолоду або тепло всередині будівлі;
– Моніторинг та управління споживанням електроенергії;
– Зменшення споживання води за допомогою сенсорних кранів та інтелектуального управління поливом рослин.

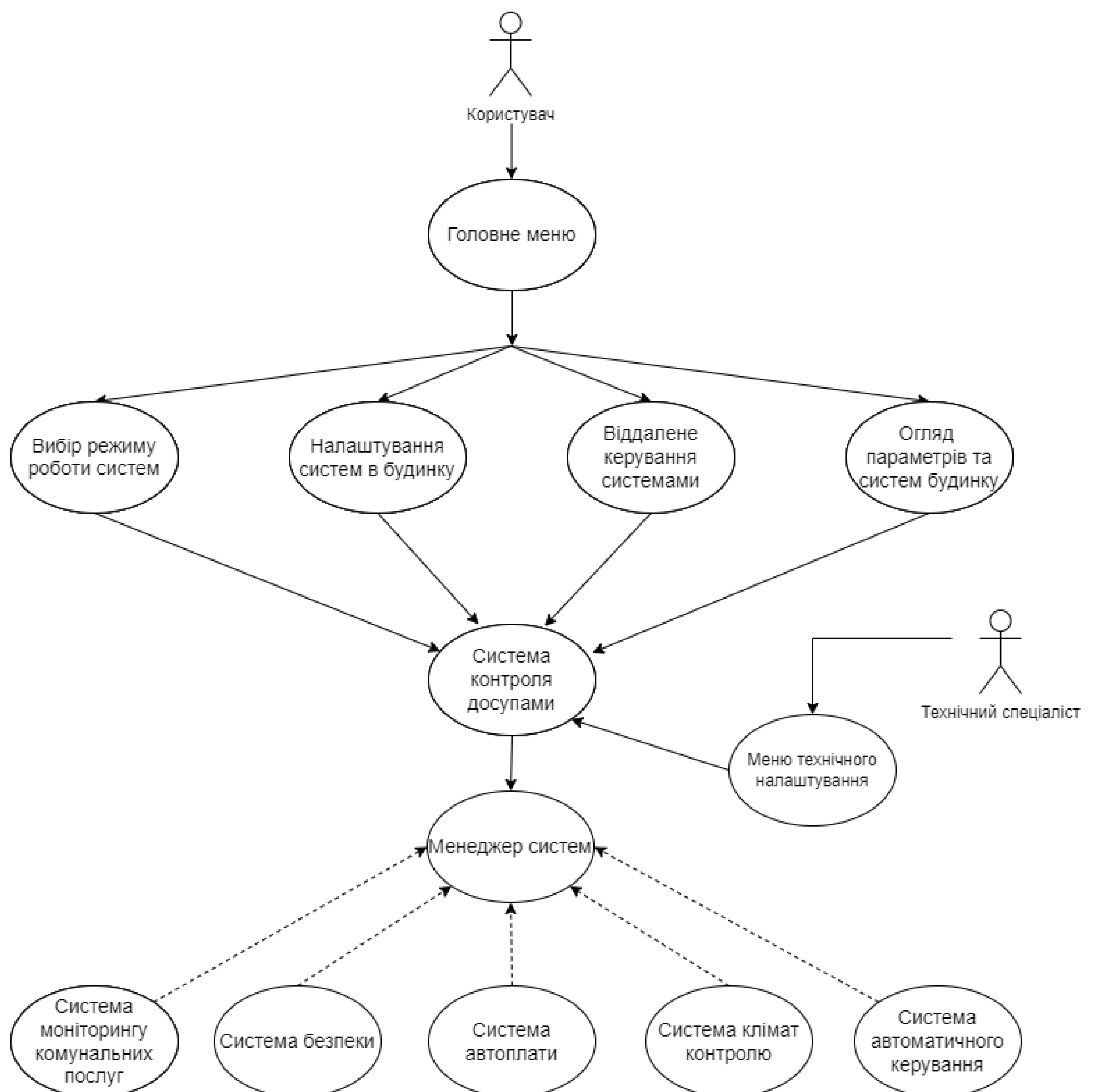

Рисунок 1.3 – Модель варіантів використання системи

1.4. Автоматизація процесу розробки

Програмні технології розвивалися так само, як життя на Землі. Спочатку веб–сайти були статичними, а мови програмування були примітивними, як і ті прості багатоклітинні організми в стародавніх океанах. У ті часи програмні рішення були призначені лише для кількох великих організацій. Тоді, на початку 90–х, популярність Інтернет призвів до швидкого збільшення кількості різних нових мов програмування та веб–технологій. І раптом стався кембрійський вибух у сфері інформаційних технологій, який призвів до розробки різноманітної кількості програмних технологій та інструментів. Зростання популярності Інтернету змінило спосіб відображення та отримання інформації.

Це триває досі. В останні роки в багатьох великих і малих організаціях існував величезний попит на програмні рішення. Кожна компанія хоче викладати свій продукт в Інтернеті через веб–сайти чи програми. Ця величезна потреба в економічних програмних рішеннях призвела до зростання різноманітного нового програмного забезпечення та методології розробки, які роблять розробку програмного забезпечення та його поширення швидкими та простими. Прикладом цього є принцип екстремального програмування (XP), який намагався спростити багато сфер розробки програмного забезпечення.

Великі програмні системи в минулому значною мірою покладалися на документовані методології, такі як модель водоспаду. Навіть сьогодні багато організацій у всьому світі продовжують це робити. Однак, оскільки розробка програмного забезпечення продовжує розвиватися, відбувається зміна в способі розробки програмних рішень, і світ стає більш гнучкішим.

1.4.1. Методологія Agile

Назва agile справедливо говорить про швидкість і легкість. Agile – це набір методологій розробки програмного забезпечення, у якому програмне забезпечення розробляється шляхом співпраці між самоорганізованими командами. Гнучка розробка програмного забезпечення сприяє адаптивному плануванню. Принципами agile є поетапна, швидка та гнучка розробка програмного забезпечення.

Agile є альтернативою традиційним процесам розробки програмного забезпечення, як обговорювалося раніше. Нижче наведено 12 принципів, на яких базується agile:

- Задоволеність клієнтів за рахунок ранньої та постійної доставки корисного програмного забезпечення;
- Можливі зміни вимог, навіть на пізніх стадіях розробки;
- Працездатне програмне забезпечення постачається часто (за тижні, а не місяці);
- Тісна щоденна співпраця між бізнесом та розробниками;
- Проекти будуються навколо мотивованих людей, яким слід довіряти;
- Розмова віч–на–віч – найкраща форма спілкування (сумісне розташування);
- Основним показником прогресу є робоче програмне забезпечення;
- Сталий розвиток, здатний підтримувати постійні темпи;
- Постійна увага до технічної досконалості та гарного дизайну;
- Простота – мистецтво максимізації обсягу невиконаної роботи – має важливе значення;
- Самоорганізаційні команди;
- Регулярна адаптація до обставин.

У процесі agile все програмне забезпечення розбивається на безліч функцій або модулів. Ці функції або модулі постачаються в ітераціях. Кожна ітерація триває 3 тижні і включає багатофункціональні команди, які працюють одночасно в різних сферах, таких як планування, аналіз вимог, проектування, кодування, модульне тестування. В результаті немає жодної людини, яка простоює в будь–який момент часу, тоді як у каскадній моделі, в той час як команда розробників зайнята розробкою програмного забезпечення, команда тестування, команда підтримки, простоюють.

На рисунку 1.4 показано, що на аналіз або розробку вимог не витрачається час. Натомість готується план дуже високого рівня, достатнього, щоб окреслити масштаби проекту.

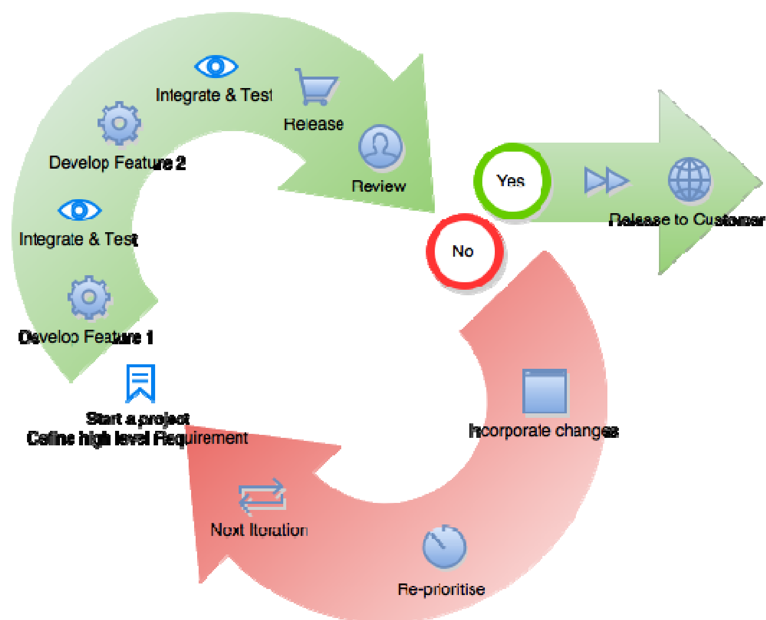

Рисунок – 1.4 План розробки проекту при використанні процеса agile

Потім команда проходить серію ітерацій. Ітерації можна класифікувати як часові рамки, кожна з яких триває місяць або навіть тиждень у деяких старих проектах. За цей час команда проекту розробляє та тестує функції. Метою є розробка, тестування та випуск функціональності за одну ітерацію. Наприкінці ітерації функціональність демонструють клієнтам. Якщо клієнти задоволені, функціональність впроваджується в проект. Якщо ні, функціональність переноситься в беклог, і переробляється в наступній ітерації. Також є можливість паралельної розробки та тестування. За одну ітерацію можливо паралельно розробляти та тестувати більше однієї функціональності.

Функціональність можна швидко розробити та продемонструвати: у agile програмний проект поділяється на фічі, і кожна фіча може бути переміщена в беклог. Ідея полягає в тому, щоб розробити одну чи набір фіч від її концептуалізації до розгортання, за тиждень чи місяць. Це ставить принаймні одну чи дві фічі на розгляд клієнта, які він може почати використовувати.

Потреба в ресурсах менша: у agile немає окремої групи розробників і тестувальників. Немає ні команди збірки чи випуску, ні команди розгортання. У agile одна проектна команда складається з восьми учасників, і кожен у команді здатний робити все. Між членами команди немає відмінностей.

Модель agile сприяє командній роботі та перехресному навчанню: оскільки існує невелика команда приблизно з восьми членів, члени команди по черзі міняються своїми ролями та дізнаються про досвід один одного.

Модель підходить для проектів, де вимоги часто змінюються: у agile програмне забезпечення поділяється на фічі, і кожна фіча розробляється та поставляється за короткий проміжок часу. Отже, зміна фічі або навіть повне її відмовлення не впливає на весь проект.

Мінімалістична документація: ця методологія в першу чергу зосереджена на швидкій розробці робочого програмного забезпечення, а не на створенні величезних документів. Документація існує, але вона обмежена загальною функціональністю.

Планування мало або воно зовсім не потрібне: оскільки фічі розробляються одна за одною за короткий проміжок часу, отже, немає потреби в широкому плануванні.

Паралельна розробка: ітерація складається з однієї або кількох функцій, які розвиваються послідовно або навіть паралельно.

1.4.2. Життєвий цикл розробки програмного забезпечення

Життєвий цикл розробки програмного забезпечення, який коротко також іноді називають SDLC, – це процес планування, розробки, тестування та розгортання програмного забезпечення. Команди виконують послідовність етапів, і кожна фаза використовує результат попереднього етапу, як показано на рисунку 1.5.

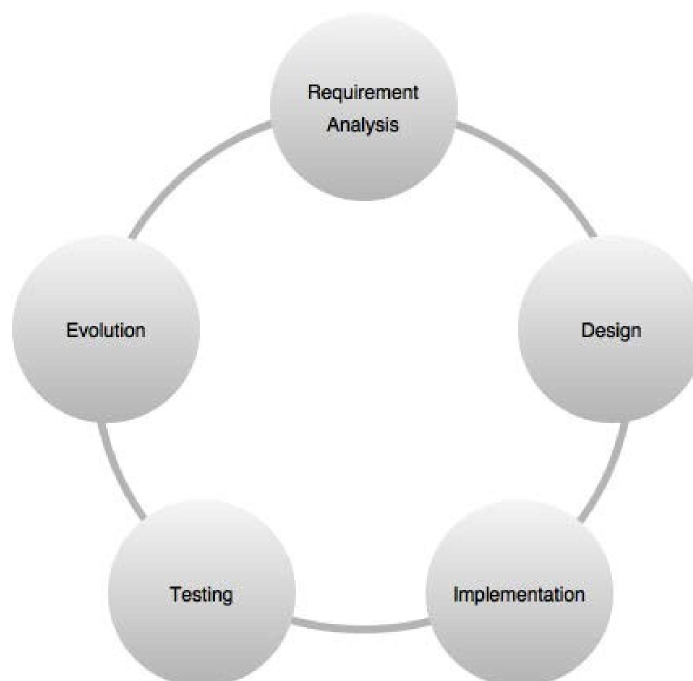

Рисунок 1.5 – Життєвий цикл розробки програмного забезпечення

По–перше, є фаза аналізу вимог: на цьому етапі бізнес–команди, які в основному складаються з бізнес–аналітиків, виконують аналіз потреб бізнес–потреб. Вимоги можуть бути внутрішніми для організації або зовнішніми з боку замовника.

Цей аналіз включає пошук природи та масштабу проблеми. На основі зібраної інформації є пропозиція або вдосконалити систему, або створити нову. Також визначається вартість проекту та викладаються переваги. Потім визначаються цілі проекту.

Другий етап – етап проектування. На цьому етапі архітектори системи та системні дизайнери формулюють бажані характеристики програмного рішення та створюють план проекту. Це може включати діаграми процесів, загальні інтерфейси, макет і величезний набір документації.

Третій етап – етап впровадження. На цьому етапі менеджер проекту створює та призначає завдання розробникам. Розробники пишуть код залежно від завдань і цілей, визначених на етапі проектування. Цей етап може тривати від кількох місяців до року, залежно від проекту.

Четвертий етап – це етап тестування. Після того, як усі визначені функції розроблені, команда тестування береться за роботу. Протягом наступних кількох місяців проводиться ретельне тестування всіх функцій. Кожен модуль програмного забезпечення збирається в одному місці і тестується. Дефекти виникають, якщо під час тестування виникають помилки. У разі збоїв команда розробників швидко реагує на це. Потім ретельно перевірений код розгортається у виробничому середовищі.

Остання фаза – це фаза еволюції або фаза підтримки. Відгуки користувачів/клієнтів аналізуються, розробляються, тестуються та публікуються у вигляді виправлень або оновлень.

2. ТЕОРЕТИЧНА ЧАСТИНА

2.1.Водоспадна модель розробки програмного забезпечення

Одним з найвідоміших і широко використовуваних процесів розробки програмного забезпечення є водоспадна модель. Водоспадна модель – це послідовний процес розробки програмного забезпечення. Модель була отримана з обробної промисловості. Модель визначає високоструктурований потік процесів, що протікають в одному напрямку. В часи коли буле розроблена модель не існувало методологій розробки програмного забезпечення, і єдине, що могли уявити розробники, це процес виробничої лінії, який було легко адаптувати для розробки програмного забезпечення. На рисунку 2.1 показано послідовність кроків у водоспадній моделі.

Тестування в цій моделі триває три місяці або більше, залежно від програмного рішення. Після успішного завершення тестування вихідний код розгортається у виробничому середовищі. Для цього знову ж таки день–два планується розгортання. Є ймовірність того, що можуть виникнути деякі проблеми з розгортанням.

Коли програмне рішення запрацює. Команди отримують зворотній зв’язок, а також зможуть передбачати проблеми. Останній етап – етап обслуговування. На цьому етапі команда розробників працює над розробкою, тестуванням та випуском оновлень програмного забезпечення та виправлень, залежно від відгуків та помилок, які виявили клієнти. Безсумнівно, що модель водоспаду чудово працювала протягом десятиліть. Проте недоліки все ж існували, але їх просто ігнорували протягом тривалого часу, оскільки тоді у програмних проектів було достатньо часу та ресурсів, щоб виконати роботу.

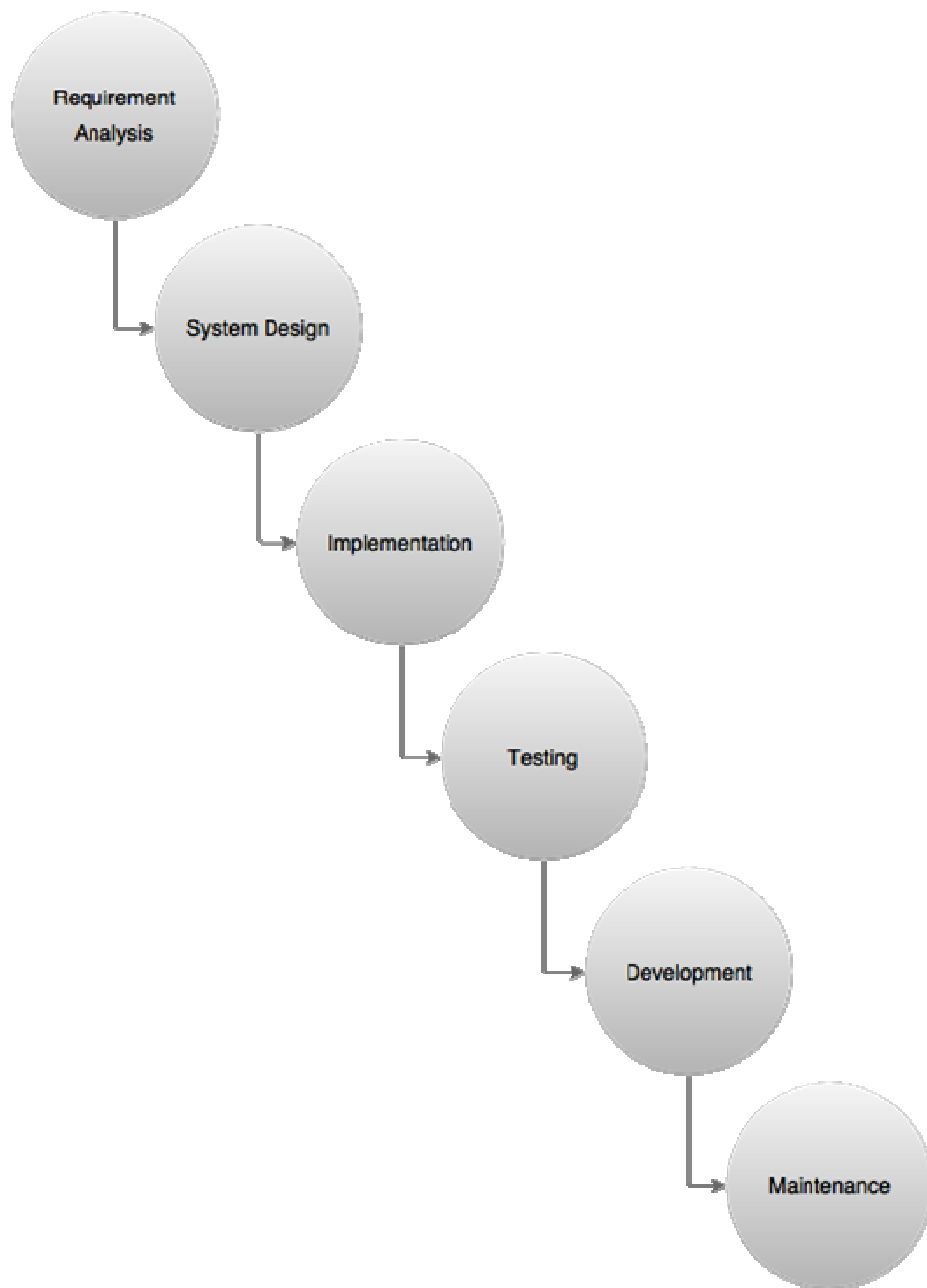

Рисунок 2.1 – Кроки розробки програмного забезпечення в каскадній моделі

Однак, дивлячись на те, як змінилися програмні технології за останні кілька років, можна легко сказати, що ця модель не відповідає вимогам сучасного світу.

Недоліками водоспадної моделі є:

– Робоче програмне забезпечення виробляється лише наприкінці життєвого циклу розробки програмного забезпечення, який у більшості проектів триває близько року;
– Важко проаналізувати ключові етапи проекту;
– Існує величезна кількість невизначеностей;
– Ця модель не підходить для проектів, заснованих на об'єктно–орієнтованих мовах програмування, таких як Java або .NET;
– Ця модель не підходить для проектів, де часто змінюються вимоги, наприклад, веб–сайти електронної комерції;
– Інтеграція виконується після завершення повної фази розробки. В результаті команди дізнаються про проблеми інтеграції на дуже пізнішому етапі;
– Немає зворотного відстеження;
– Важко вимірювати прогрес поетапно;

Дивлячись на недоліки водоспадної моделі, можна сказати, що це переважно ця модель підходить для проектів, де:

– Вимоги добре задокументовані та фіксовані
– Фіксована ціна проекту;
– Є достатньо фінансування для підтримки команди управління, команди тестування, команди розробників, команди створення та випуску, команди розгортання тощо;
– Підходить для процедурних мов програмування
– Технологія фіксована;
– Затверджений конкретний чай розробки проекту;
– Немає неоднозначних вимог.

2.2.Вибір сервера безперервної інтеграції

Як сервер для безперервної інтеграції був вибраний Jenkins. Нижче в підрозділі описані плюси інструмента Jenkins.

На ринку є ряд інструментів безперервної інтеграції, таких як Go, Bamboo, TeamCity тощо. Але найкраще в Jenkins те, що він безкоштовний, простий, але потужний і популярний серед спільноти DevOps.

Jenkins підтримує спільнота з відкритим кодом. Усі люди, які створили оригінальний Hudson, працюють над розвитком Jenkins після розриву закриття Hudson.

Для Jenkins доступно понад 300 плагінів, і список постійно збільшується. Плагіни – це прості проекти Maven. Тому будь–хто з творчим розумом може створювати свої плагіни та ділитися ними в спільноті Jenkins.

Бувають випадки, коли кількість запитів на збірку, пакування та розгортання більше, а іноді менше. У таких сценаріях необхідно мати динамічне середовище для виконання збірок. Цього можна досягти шляхом інтеграції Jenkins за допомогою хмарного сервісу, такого як AWS. За допомогою цього налаштування середовища збірки можна створювати та знищувати автоматично за потребою.

Jenkins об’єднує всі інші інструменти DevOps для досягнення безперервної інтеграції. На рисунку 2.2 зображено, як Jenkins спілкується з інструментом контролю версій, інструментом репозиторію та інструментом статичного аналізу коду за допомогою плагінів. Аналогічно, Jenkins спілкується з серверами збірки, тестовими серверами та виробничим сервером за допомогою підлеглого агента Jenkins.

Відповідь на вимоги Jenkins до обладнання є досить складним завданням. В ідеалі для запуску головного сервера Jenkins достатньо системи з Java 7 або вище та

1–2 ГБ оперативної пам’яті. Однак є організації, які використовують до 60+ ГБ оперативної пам’яті лише для свого Jenkins Master Server.

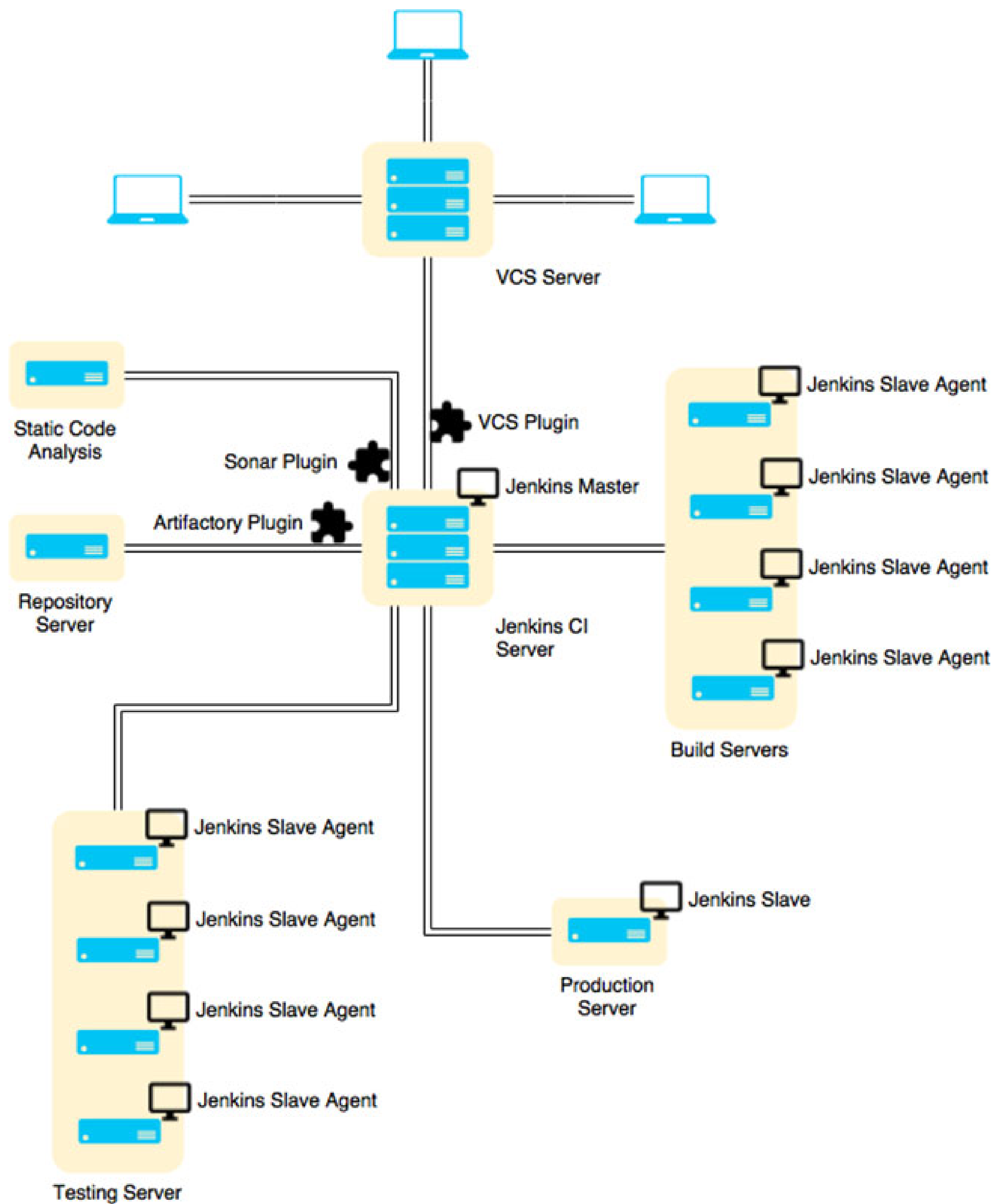

Рисунок 2.2 – Діаграма роботи Jenkins

2.3. Архітектура програмної системи

Архітектура програмної системи є ключовим питанням в роботі. Подальше в розділі описана архітектура програмного комплексу Smart House.

Спроба реалізувати складні програми з використанням традиційної об'єктно–орієнтованої архітектури програмування часто може бути дуже важким завданням. Масивні ієрархії класів, що походять від одного кореневого класу, складні шляхи успадкування та додавання нових типів сутностей можуть зробити кодову базу непотрібно складною.

У той же час знижується гнучкість і можливість повторного використання коду, а також погіршується продуктивність. Інкапсуляція даних в об’єктно–орієнтованих методах також є проблемою, яка вимагає уваги для досягнення хорошої продуктивності під час роботи з великою кількістю об’єктів.

Для боротьби з цими проблемами традиційний об’єктно–орієнтований дизайн можна замінити більш орієнтованим на дані підходом до проектування з використанням моделі «композиція–над–наслідування», де дані та логіка розділені, яка називається Entity–Component–System (ECS).

У цьому підході властивості сутностей можна визначити за допомогою невеликих, багаторазових і загальних компонентів, які не містять ніякої логіки в собі. Замість цього логіка обробляється різними системами, які порівнюються з компонентами і постійно виконують свої внутрішні методи для них у фоновому режимі [8].

На рисунку 2.3 показано UML діаграма класів бібліотеки команд для системи.

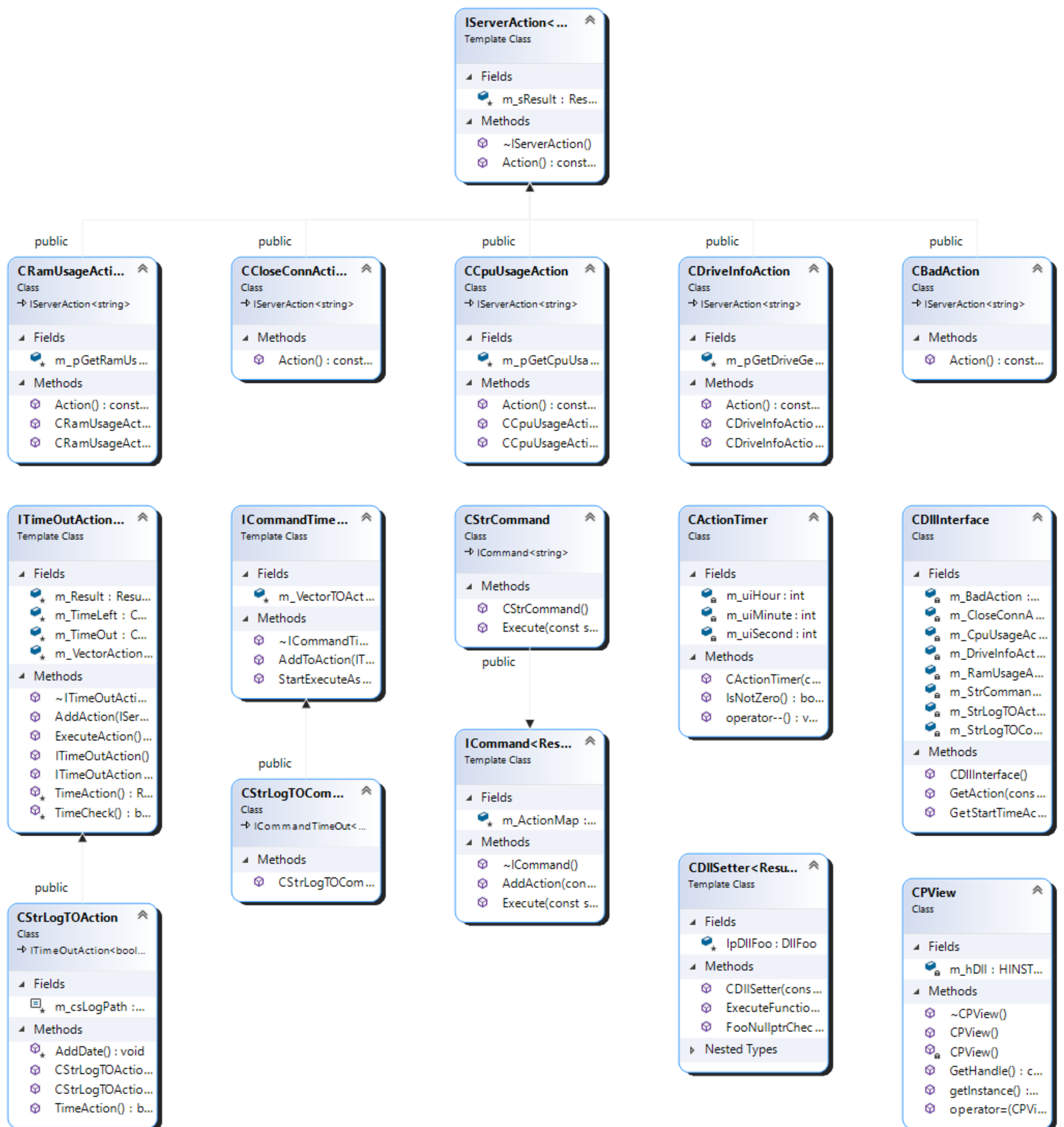

Рисунок 2.3 – UML діаграма класів бібліотеки команд

2.3.1. Entity–component–system

Entity–Component–System – це архітектурний шаблон розробки програмного забезпечення, який слідує принципу композиції над успадкуванням, спрямований на досягнення гнучкого та динамічного способу керування об'єктами у великомасштабних програмах реального часу. Відокремлюючи дані від логіки, ECS досягає модульної системи, яка дозволяє зручно зберігати дані в суміжних областях пам'яті та легко розпаралелювати логіку програми. Завдяки модульності системи сутностей можуть уникнути підводних каменів об'єктно–орієнтованого програмування (ООП), таких як неоднозначність у множинному наслідуванні.

Добре уявити різницю між системами сутностей та об'єктно–орієнтованим програмуванням із самого початку. Хоча системи сутностей часто реалізуються за допомогою об'єктно–орієнтованих мов, спроба інкапсулювати дані та логіку в ті самі об'єкти в середовищі ECS в кінцевому підсумку закінчиться невдачею [7]. Навіть якщо можливо і навіть заохочується використовувати OOP і ECS в рамках одного проекту, необхідно зробити чітке розділення між рівнями, і будь–який рівень повинен бути реалізований за допомогою OOP або ECS [8].

Реалізація системи сутностей може розглядатися як розширення моделі об'єктно–орієнтованої композиції, але її реалізація та використання вимагає від користувача абсолютно нового способу мислення. У цьому підході екземпляри сутності зводяться з об'єктів до просто числових цілочисельних ідентифікаторів, які служать лише з метою об'єднання компонентів в єдину сутність. Компонентами цієї парадигми є прості, невеликі і лишені логіки, представлення властивостей і даних сутності.

Логіку обробляють окремі системи, які завжди шукають активні компоненти. Вони не містять у собі ніяких даних і використовуються лише для перетворення вхідних даних у потрібний спосіб.

Три елементи складаються разом за допомогою менеджера контексту, який дозволяє різним частинам шаблону спілкуватися один з одним. Менеджер контексту відповідає за ведення запису про те, які компоненти належать до яких сутностей, і використовується для передачі необхідних властивостей у відповідні системи.

2.3.2. Сутності в Entity Systems

Ядром парадигми є сутності, які можна розглядати як фундаментальні концептуальні будівельні блоки системи. Кожна сутність має представляти конкретну концепцію, але лише на ідеологічному рівні, оскільки насправді вони самі по собі не є функціональними агентами. Насправді сутність складається лише з унікального ідентифікатора, який відрізняє визначені сутності один від одного [9]. Сутність не містить жодних даних чи методів, а також не є екземпляром класу. В ідеалі сутність також не повинна бути списком чи масивом компонентів. Єдина функція сутностей полягає в тому, щоб скласти пов’язані компоненти разом, щоб сформувати їх у більш конкретного актора [8]. Це робиться шляхом надання компонентам можливості позначатися ідентифікатором, щоб вказати, в яких об’єктах вони активні [7].

У найпростішому випадку нові об’єкти можна створити за допомогою простої функції яка показана на лістингу 3.1 статичного збільшення, яка генерує новий ідентифікатор об’єкта, додавши 1 до останнього ідентифікатора об’єкта.

Лістинг 3.1 – Приклад функції для створення сутності

```
std::size-t generate-entity()
{
static std::size-t entityId = 0;
```

```
return entityId++; }
```

Ідентифікатор об’єкта не обов’язково має бути складнішим за додатне число, іноді наявність певної логіки, що лежить в основі ідентифікації, може бути корисною для структурування даних. Наприклад, як ідентифікатор можна, використовувати як індекс у масиві компонентів, що відображає, які об’єкти мають активний конкретний компонент.

2.3.3. Компоненти

Компоненти – це сегмент даних ECS. Це невеликі, загальні та багаторазові типи, які використовуються для визначення властивостей сутності та того, як вона може взаємодіяти з іншими об’єктами. Вони містять усі дані, що належать їхнім відповідним сутностям. Згідно з визначенням Адама Мартіна, компоненти зберігають дані, але не містять жодної логіки. Кожен компонент представляє інший аспект сутності, надаючи йому дані, необхідні для володіння цим конкретним аспектом [8]. Наприклад, скелет–солдат у відеогрі може мати такі компоненти, як:

- Нежить
- Ворог
- Координати
- Колізії
- Переміщення

Жоден з цих компонентів не містить у собі ніякої логіки. Їх мета полягає в тому, щоб позначити сутність для її властивостей, а також бути окремими контейнерами для даних об’єктів, щоб їх можна було зберігати як прості ідентифікатори.

2.3.4. Системи

Для забезпечення логіки для компонентів використовуються системи або процесори. Вони є агентами з глобальною областю дії, що означає, що на відміну від методу традиційного методу об’єкта, їх можна викликати з будь–якої точки коду. Замість того, щоб безпосередньо орієнтуватися на конкретну сутність, їх цікавлять активні компоненти різних сутностей. Компоненти додаються до контексту, який потім передається відповідній системі як аргумент для визначення меж операції. Функціонуючи окремо за межами структури сутність–компонент, система порівнюється з набором компонентів, що володіють тими ж аспектами, що й ця система, і виконує свої внутрішні методи над компонентами безперервно у фоновому режимі, по одному. Системи не повертають значення, а натомість просто змінюють стани різних компонентів, виконуючи перетворення даних. Приклад системи показаний на лістнигу 3.2.

Лістинг 3.2 – приклад системи

```
void example-system(context& c)
{
c.for-entities-with<a-type, c-type>([&c](auto eid)
{
auto& a-data = c.get-component<a-type>(eid);
auto& c-data = c.get-component<c-type>(eid);

perform-action(a-data, c-data);
}); }
```

Наведена вище система проходить через кожну сутність, що має певний набір компонентів. У рядках 5 і 6 витягуються фактичні дані, що належать компонентам сутності, а в рядку 8 система виконує перетворення цих даних.

Основний цикл програмного забезпечення, що працює на архітектурі ECS, полягає в ітерації всіх існуючих систем. Кожна система порівнюється з підмножиною Entity/Component Binary Large Objects (BLOB). BLOB вибираються із глобального резерву, і дані надсилаються до ЦП разом із логічним кодом.

2.3.5. Складання моделі

Програмування з entity–component–system можна порівняти з програмуванням реляційних баз даних. Інстанс сутності діє аналогічно ключу в базі даних, як і в будь–якій системі керування реляційною базою даних (RBMS). З абстрактної точки зору, доступ до компонента певної сутності є не що інше, як запит до бази даних [8]. Ця концепція показана на таблиці 2.1, де кожен рядок представляє екземпляр сутності, а кожен стовпець представляє інший компонент.

Таблиця 2.1 – Концепція RBMS

	Component A	Component B	Component C
Entity #0	X		X
Entity #1	X	X	
Entity #2			X

Можна побачити, як різні компоненти пов’язані з сутностями, і як екземпляр компонента може бути спільним між кількома сутностями [10, 11]. Менеджер

контексту використовується, щоб бути в курсі того, яка сутність має які компоненти, а доступність компонента можна перевірити за допомогою такого методу який зображений на лістнигу 3.3 [10].

Лістинг 3.3 – Приклад методу перевірки на доступність компонента

```
bool context::has-component<...>(entity-id)
```

Де контекст еквівалентний класу менеджера контексту. Після підтвердження доступності компонента доступ до його даних здійснюється методом, який зображений на лістингу 3.4 [10].

Лістинг 3.4 – Приклад методу доступу до компонента

```
auto& context::get-component<...>(entity-id)
```

Який повертає екземпляр компонента для вказаної сутності.

Нові типи компонентів створюються як структури, що містять змінні відповідного компонента. Для кожного компонента створюється масив для зберігання всіх сутностей, які використовують цей компонент, як це показано на лістингу 3.5.

Лістинг 3.5 – Приклад класу для зберігання компонентів

```
struct component-a { /* ... */ };
struct component-b { /* ... */ };
struct component-c { /* ... */ };

constexpr auto max-entities{10000};

class component-storage
```

```
{
private:
```

Продовження лістингу 3.5

```
std::array<component-a, max-entities> -a;
std::array<component-b, max-entities> -b;
std::array<component-c, max-entities> -c;
public:
template <typename TComponent>
TComponent& get(entity-id eid);
};
```

Менеджер контексту с використовується для передачі необхідних компонентів процесору, який потім може виконувати свої операції над сутностями з необхідними типами компонентів як це показано на лістингу 3.6.

Лістинг 3.6 – Менеджер контексту

```
struct system-ac
{
void process(context& c)
{
c.for-entities-with<a-type, c-type>([&c](auto eid)
{
/* ... */
});
}
};
```

Клас зберігання компонентів ініціалізується як об’єкт і встановлюється в контекст. Контекст безперервно передається системі, яка виконує перетворення даних у кожному циклі.

2.4. Орієнтований на дані дизайн

Під час розробки важкого програмного забезпечення часто виникають проблеми продуктивності, через які час виконання програми для різних завдань зупиняється на неприпустимо довгий час. Причин цього може бути кілька, найочевиднішою з яких є неоптимізовані алгоритми виконання, але це не завжди так. Досить часто сповільнення викликано неефективним управлінням пам’яттю [11, 14]. Щоб мати можливість більш ефективно керувати своїми даними, можна здійснити перехід від об’єктно–орієнтованого мислення до Data–Oriented Design (DOD), підмножиною якого є ECS. DOD прагне повністю перемістити фокус з об’єктів на обробку фактичних даних. Оскільки програмування за визначенням стосується перетворення даних, насправді не потрібно думати, як ізольований об’єкт буде робити речі. Замість цього нехай методи здійснюють перетворення загальними способами та намагаються організувати дані якомога ефективніше для обладнання [11].

У більшому масштабі оптимальне розміщення даних у пам’яті може сильно вплинути на продуктивність програмного забезпечення, тому важливо звернути увагу на типи даних та спосіб їх обробки. В ідеалі дані розміщуються якомога однорідніше та безперервно, щоб їх можна було обробляти послідовно.

Ідеальний дизайн може бути досягнутий шляхом розбиття об’єктів на різні компоненти та групування цих компонентів у пам’яті за їх типом. Це призводить до однорідних і послідовно оброблюваних даних. Цей підхід дуже добре працює з

великими групами об’єктів, на відміну від ООП, яке в основному фокусується на одному об’єкті. З цієї причини ECS дуже ефективний і широко використовується в розробці ігор: ігрові об’єкти зазвичай обробляються групами.

Обробка даних у наборах дозволяє організувати їх так, щоб обробку можна було оптимізувати для роботи з групами одного типу. DOD також дозволяє користувачеві використовувати багатопотокову обробку, а також більший відсоток звернень у кеш. Завдяки модульній природі ECS, парадигма дозволяє створити надзвичайно гнучку ієрархію об’єктів і полегшує мережу та серіалізацію даних [11].

Застосування орієнтованого на дані проектування та ECS до частини програмного забезпечення можна розглядати як ітераційний процес. Вибирається конкретна частина реалізації інкапсульованого об’єкта, яка реалізується орієнтованим на дані способом. Це повторюється до тих пір, поки більшість коду не буде зосереджена на даних. Після того, як предмет, який буде реалізований, вибрано, необхідно визначити входи та результати.

Важливо розглянути, чи є вхідні дані доступними лише для читання, для читання–запису чи лише для запису, оскільки це допомагає визначити, як дані мають зберігатися, і як будуть оброблятися різні залежності.

Дані, доступні лише для читання, можна безпечно розподіляти між кількома потоками, але при обробці даних, що перетворюються, може знадобитися деяке ретельне проектування [11].

Після того, як параметри були продумані, необхідно спланувати реальну реалізацію. Системи повинні бути достатньо загальними, щоб обробляти сотні записів, тому функціональність не може залежати від того, щоб вхідні дані були абсолютно однаковими для кожного виконання. Перетворення даних вимагає ретельного планування щодо того, як їх можна ефективно кешувати та розпаралелювати [11].

2.4.1. Переваги дизайну орієнтованого на дані дизайну

В об’єктно–орієнтованому програмуванні поділ процесу між кількома ядрами може бути тяжкою операцією через помилки синхронізації, викликані різними потоками, які намагаються одночасно отримати доступ до одних і тих же даних. Якщо потоки чекають своєї черги для доступу до даних, це призводить до великої кількості простою, а віддача з точки зору підвищення продуктивності може бути незадовільною. DOD спрощує розпаралелювання. Оскільки дані обробляються групами, їх легко розділити між різними потоками, і код не викличе проблем, пов’язаних із синхронізацією.

Іншою сильною стороною Data–Oriented Design є можливість оптимізованого використання кешу. У сучасному обладнанні ключовим моментом для досягнення високої продуктивності є впорядкування даних у пам’яті, щоб їх можна було ефективно використовувати знову і знову. Якщо дані розміщуються в пам’яті безперервно, їх можна обробляти з майже ідеальним використанням кешу, що призводить до чудової продуктивності. Хоча оптимізація алгоритмів, що використовуються для перетворення даних, безумовно важлива, якщо поглянути назад на те, як обробляються дані, це може дати ще більші результати у великій перспективі [11].

Вже обговорюваною перевагою використання композиції, орієнтованої на дані, є її модульність. Хоча це не приносить програмі додаткової продуктивності, воно значно полегшує процес розробки. Збереження невеликих функцій і уникнення залежностей між різними частинами коду запобігає розгалуженню бази коду, що покращує читабельність коду та значно полегшує його оновлення та переписування. Хоча модульність не обмежується лише проектуванням, орієнтованим на дані, це основний фактор, який слід враховувати, і тому варто згадати [11].

Останньою важливою перевагою DOD є те, наскільки легко можна перевірити код. Оскільки всі функції зосереджені на безпосередньому перетворенні даних, модульні тести просто повинні прийняти якийсь тип вхідних даних, виконати перетворення і перевірити, чи відповідає результатам очікування. Не потрібно турбуватися про залежності між різними частинами коду [11].

3. ПРАКТИЧНА ЧАСТИНА

3.1. Архітектура тестів програмного забезпечення

Тестування програмного забезпечення як концепція існує майже так само довго, як і сама програмна інженерія. Починаючи з простої практики налагодження, вона перетворилася на важливу область перевірки програмного забезпечення і сьогодні є важливою частиною процесу забезпечення якості програмного забезпечення. Оцінка нових і існуючих інструментів і практик тестування програмного забезпечення все ще триває. Поки розробка програмного забезпечення розвиватиметься, тестування та потреба в подальшій оцінці поточних практик тестування та мовних рамок будуть розвиватися.

Тестування необхідне при перевірці логічної коректності та ходу програми в системах і програмах. Система реального часу – це система, призначена для використання в середовищах, де обмеження часу необхідне для виконання операцій, і є критичним критерієм, а також логічна правильність виконання операції. Прикладами такої системи є антиблокувальні гальмівні системи (ABS) в транспортних засобах і мобільні широкосмугові системи в телекомунікаційній галузі. Оскільки функціональні можливості в системах реального часу повинні виконуватися в межах часових обмежень, це може додавати додаткові складності на етапах проектування, розробки та перевірки системи реального часу. Для розробки таких систем можна використовувати інструмент моделювання Rational Software Architect RealTime Edition (RSARTE). Код, згенерований з моделей UML в RSARTE, призначений для виконання в середовищі виконання C++ в режимі реального часу. Важливим будівельним блоком для функціональності реального часу є модель Capsule, розширена версія більш узагальненої концепції моделі Actor від RSARTE. Капсула – це елемент із внутрішньою машиною станів та зовнішнім інтерфейсом,

який визначає поведінку та зв'язок з іншими капсулами. Ця конструкція дуже допомагає при програмуванні додатків реального часу.

RSARTE надає достатню підтримку та функціональність для UML–дизайну, моделювання та генерування коду, але не має належної офіційної підтримки для модульного тестування згенерованих компонентів капсули. Це змушує користувачів, які бажають використати методи тестування програмного забезпечення, розробляти та підтримувати власні фреймворки. Фреймворки для тестування можуть бути інтегровані та використані для тестування. Якщо така можливість існує, це може усунути необхідність розробки власних платформ тестування для виконання модульного тестування на капсулах. Та звільнить багато важливого часу та ресурсів, які можна краще витратити на розробку та перевірку продуктів.

Як частина розробки сучасної системи, тестування програмного забезпечення – це сукупність заходів і методів, які використовуються для дослідження якості програмного продукту або системи. Мета цього процесу подвійна. Перша мета полягає в пошуку та знаходженні помилок і збоїв у системі, щоб їх можна було пізніше виправити і таким чином покращити якість системи. Друга причина полягає в перевірці та валідації, де тестування служить показником того, наскільки система відповідає вимогам і очікуваній функціональності [12]. Тестування не може гарантувати безвідмовну систему, натомість основна мета полягає в тому, щоб сприяти використанню методів і практик тестування розробниками під час процесу розробки, які можуть допомогти максимально зменшити помилки [13].

Існує кілька методів і процесів для тестування програмного забезпечення, які розвивалися протягом багатьох років. На початку розробки програмного забезпечення тестування складалося здебільшого з практик налагодження, спрямованих на вирішення недоліків програмування, або збоїв у міру їх виникнення. Сьогодні чітко розрізняють тестування програмного забезпечення та налагодження. Тестування – це процес пошуку помилок і збоїв, тоді як налагодження виконується

розробниками, щоб ізолювати та виправляти раніше виявлені дефекти та помилки в програмному забезпеченні [14].

Несправності, що виникають через помилку, можуть варіюватися від дратівливої поведінки системи з мінімальним впливом, до втрати великої кількості грошей і часу або навіть бути небезпечними для життя. Оскільки наявність помилок і збоїв у програмному забезпеченні може негативно вплинути на якість і характеристики системи, існує необхідність знайти та зменшити кількість помилок настільки, наскільки це можливо [16].

Оскільки мета тестування полягає в тому, щоб знайти якомога більше помилок і збоїв, існує кілька методів, розроблених для цієї практики. Загальна стратегія при тестуванні полягає в тому, щоб "зламати" програму шляхом систематичного визначення вхідних даних системи та очікуваної поведінки від цього входу. Відмінності в техніках тестування походять від того, як створюються тести, а також від відмінностей у структурі коду, очікуваних або уявних типах помилок, передбачуваного використання системи та специфікацій [17, 18]. Тестування також може виконуватися на різних рівнях системи залежно від того, коли в розробці проводиться тест, і від цілі тесту, яка може варіюватися від окремого або групи модулів до всієї системи [19].

3.2. Метод тестування чорної коробки

Тестування чорної коробки – це класифікація методів тестування з метою визначення того, чи відповідає програма своїм функціональним вимогам [17]. Це робиться шляхом перегляду програми або частини системи, що підлягають тестуванню, як "чорної коробки" метод показаний на рисунку 3.1, не маючи можливості побачити, як вона працює всередині. Надсилаючи вхідні дані до чорної

коробки, результат відповіді можна спостерігати, не знаючи внутрішньої структури коробки. Функціональність може бути оцінена шляхом порівняння спостережуваного результату з очікуваним результатом. Тестеру не потрібно володіти конкретними знаннями про код програми, лише те, що він повинен робити, оскільки зосереджено на перевірці функціональності [20].

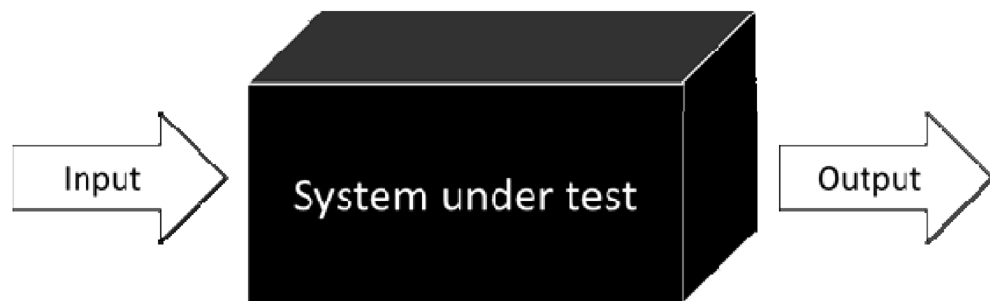

Рисунок 3.1 – Метод тестування чорної коробки

3.3.Рівні тестування

Тестування можна згрупувати в різні категорії залежно від того, де в життєвому циклі розробки проводиться тестування і для яких цілей [19]. Життєвий цикл розробки програми використовується для опису серії етапів у проекті розробки програмного забезпечення. Ці фази визначаються як окремі кроки, які використовуються розробниками для планування, проектування, створення, тестування та доставки програмних продуктів [20]. У сучасній розробці системи тестування має проводитися на кожній фазі життєвого циклу розробки [19].

Поділ на рівні тестування за часом проведення тестування зроблено для того, щоб уникнути накладання або повторення тестів і допомогти знайти відповідні цілі для тестування [19]. Комп'ютерне товариство IEEE визначає в Зводі знань

програмної інженерії (SWEBOK) три цільові рівні для тестів і дванадцять цілей тестування. Цільовими рівнями, визначеними в SWEBOK, є три наступні:

- Unit Testing: тестування на найнижчому рівні. Тести окремих елементів вихідного коду ізольовано.
- Integration Testing: тестування груп об'єднаних програмних модулів, мета – перевірити взаємодію між модулями
- System Testing: тестування цілісної інтегрованої системи з метою перевірки функціональності, продуктивності та нефункціональних вимог [20].

3.4. Unit Testing

Модульне тестування – це метод тестування найменшої окремої частини програмного забезпечення в системі, яка називається блоками. Тестування на рівні одиниці є найнижчим цільовим рівнем тестування. Мета полягає в тому, щоб перевірити функціональність для найменших подільних частин програми та показати, що окремі блоки є правильними. Розмір випробуваних частин може, залежно від ситуації, варіюватися від невеликих окремих функцій до більших композитних модулів, які мають високу згуртованість [20, 21]. Наприклад, з точки зору процедурного програмування модуль або одна процедура були б одиницями відповідного розміру. У той час як в об'єктно–орієнтованому поданні програмування клас або окремий метод можна вважати одиницею відповідного розміру для тестування [22].

При виконанні модульного тестування кожен тестовий модуль не залежить один від одного. Мета полягає в тому, щоб запустити всі тестові випадки ізольовано, щоб попередні тести не впливали на наступні. Виконуючи тести ізольовано, кожен

тест може охоплювати певний набір умов. Це допомагає виявити проблеми на ранніх стадіях, коли розробникам потрібно додатково визначити поведінку та реакцію на помилки пристрою. Рефакторинг коду також полегшується за допомогою модульних тестів, оскільки зміни можна перевірити за допомогою наявних тестових випадків і швидко виявити можливі проблеми [21].

Модульне тестування не може дати повну перевірку поведінки, оскільки воно не може вловити всі можливі збої в результаті помилки програмного забезпечення. Це вимагає тестування всіх шляхів виконання та покриття коду, що швидко може стати нездійсненним через обмежений час для розробки та виконання тестів. Оскільки модульне тестування фокусується на функціональності окремих блоків, воно погано підходить для виявлення помилок інтеграції [21].

Практичний спосіб виконання модульного тестування розробниками – це використання тестових фреймворків і написання тестового коду, який виконується автоматично. Метод модульного тестування формально не вимагає автоматизації, оскільки тестові випадки можуть бути задокументовані та виконані вручну розробниками. На практиці виявляється, що ручне модульне тестування займає багато часу, дороге та схильне до помилок [21].

3.5.xUnit Архітектура

xUnit – це назва для великої кількості платформ модульного тестування, які мають спільну архітектуру тестування для автоматизації скриптових тестів. Першим найбільш відомим із сімейства xUnit є фреймворк JUnit для Java. Описаний Мартіном Фаулером як “... будучи маленьким і простим, він заохочував людей вчитися та використовувати його”. Пізніше він віддав йому належне за те, що він

зіграв велику роль у зміні ставлення до тестування та підтримав розвиток практик, орієнтованих на тестування, та розробки, орієнтованої на тестування [22, 23, 24].

Архітектура платформи тестування xUnit почала свою розробку в Smalltalk, де Кент Бек був провідним учасником в процесі розробки. Кент створив просту структуру, яку можна було б використовувати для організації та проведення модульних тестів. Ця структура була створена з метою полегшити програмістам визначення та запуск тестів, використовуючи їхнє звичайне середовище Smalltalk [23].

Більшість мов програмування, які використовуються сьогодні, мають принаймні одну реалізацію фреймворку з використанням дизайну xUnit. Скриптові тести зазвичай автоматизуються з використанням тієї ж мови програмування, яка використовується для побудови SUT (система під тестуванням) [24]. Тести, написані вручну, або скорочено сценарні тести, мають кілька цілей, наприклад: регресійне тестування програмного забезпечення після його зміни, документування поведінки програмного забезпечення, або визначення поведінки програмного забезпечення до його написання [26].

Ядро архітектури xUnit показане на рисунку 3.2 складається з п'яти компонентів: TestCase, TestRunner, TestFixture, TestSuite і TestResult [25].

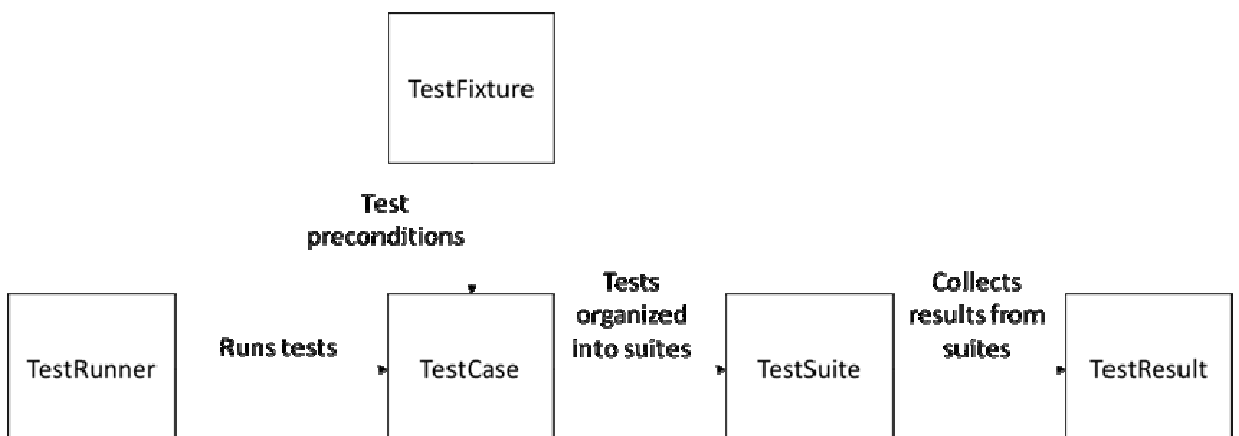

Рисунок 3.2 – Ядро архітектури xUnit

У цьому дизайні окремі модульні тести структуровані в тестові кейси. Коли тестовий кейс виконується, він спочатку встановлює передумови та дані, необхідні для даного тестового прикладу. Потім виконується фактичний тест і перевіряється результат поведінки. Потім передумови в тесті очищаються, щоб наступні тести можна було виконувати ізольовано і не впливати на попередні тести. Необхідні ініціалізації та набір передумов для тесту можуть бути спільними для кількох тестових випадків. Тестовий контекст, такий як цей, називається text fixture. Потім тестові кейси групуються і організовуються в набори тестів. Тестовий запуск – це виконувана програма, яка запускає всі тести та зберігає результати в побудові результатів тесту [27].

3.6.План тестування

Перш ніж розпочинати писати самі тести, потрібно проаналізувати існуючі фреймворки, та вибрати найбільш підходящий для виконання поставлених цілей тестування. Тому в цьому підрозділі будуть описані найбільш відомі фреймворки тестування на мові програмування C++.

3.6.1. Порівняння фреймворків

Boost Test має безліч функцій і підтримки, з автоматичною або ручною реєстрацією тестів і частими оновленнями, хоча з недоліками, наприклад, він завжди запускає тест кейси в групі і не може запускати їх окремо, а також необхідно вручну

реалізовувати форматування для результатів тестування. Документація Boost Test добре розділена на глави. Розділи охоплюють різні області, деякі з яких мають приклади, хоча код можна було б краще прокоментувати. Документація є дещо довгою та дезорієнтованою, і, ймовірно, її можна було б узагальнити краще.

CppUnit не є сильнішим кандидатом через відсутність деяких корисних функції, також відсутність підтримки Mocking Technique та ручну реєстрацію тесту. Хоча його сильні сторони полягають у тому, що він портативний, стабільний і відповідає шаблонам xUnit. CppUnit не має власної домашньої сторінки, де він міг би містити всю необхідну інформацію та вихідні матеріали, що зменшує доступність і кількість документації та прикладів коду.

Google Test має багато функцій і часто оновлюється, і також простий у використанні через доступність документації та автоматично реєструє тести. Хоча одним з його найбільших недоліків може бути те, що Google не випускає оновлення між виправленнями.

Фреймворк xUnit++ не підходить для тестування системи “Smart House”. Йому бракує необхідних функцій і підтримки, і він не може визначити свій власний основний метод. Однією з його найбільш унікальних функцій є рандомізований порядок тестування, який чудово підходить для тестування незалежності, але в даному варіанті потрібна система тестування, де можливо вказати порядок. Це може бути лише тимчасовими недоліками, оскільки xUnit++ дуже молода платформа тестування. Оскільки xUnit++ свіжа платформа тестування, вона може мати потенціал у майбутньому для подальшого розвитку в структуру тестування з більшою зручністю використання. xUnit++, як і CppUnit, не маєть власної домашньої сторінки, а є частиною іншого хоста bitbucket.

Boost Test і Google Test є найсильнішими кандидатами для цього, оскільки обидва фреймворки охоплюють широкий спектр функцій із надійною документацією в поєднанні з живою спільнотою та регулярно обслуговуються. Це надійна основа

для того, щоб залишатися актуальним як фреймворк модульного тестування в майбутньому.

Таблиця 3.1 – Можливості фреймворків

Feature	Boost Test	CppUnit	Google Test	xUnit++
Ассерт	Так	Так	Так	Так
Моки	Так	Ні	Так	Ні
Фікщур	Так	Так	Так	Так
Сют	Так	Так	Так	Так
Макроси	Так	Так	Так	Так
Чекпоінти	Так	Ні	Частково	Ні
Винятки	Так	Так	Так	Так
Дез Тести	Так	Ні	Так	Ні
Шаблони	Так	Ні	Так	Так
Підтримка нових стандартів C++	Так	Так	Так	Так
Підтримка платформи Unix, Windows, OSX	Усі	Усі	Усі	Unix та Windows

Таблиця 3.1 є підсумком та оглядом доступних особливостей тестування для Boost Test, CppUnit, Google Test і фреймворку xUnit++. Різні системи тестування підтримують різні особливості та доступні тести. Виділяються дві системи тестування: Boost Test і Google Test. Незважаючи на те, що ці дві платформи мають найбільшу кількість особливостей серед чотирьох фреймворків тестування, це не повний список можливих особливостей фреймворків тестування.

Дальше в розділі описана конкретна особливість, та як вона імплементована в конкретному фреймворку.

Ассерт – це логічні вирази, які спрямовані на оцінку результатів з конкретних даних у програмі тестування. Тестове Ассерт, яке визначається як вираз, інкапсулює деяку перевірену логіку, визначену щодо цілі, що тестується [29].

Функціональність Ассерт дає змогу виконувати фактичні перевірки в тесті, тому всі вибрані рамки тестування підтримують це. Хоча вони мають схожість у тому, як вони використовуються в кожній структурі. Схожість між Boost Test, CppUnit і Google Test полягає в тому, що функціональні можливості підАссерт надаються за допомогою набору макросів у кожній структурі.

Google Test друкує інформацію консолі, якщо тест пройшов чи не пройшов, інформація автоматично форматується з розташуванням вихідного файлу та номером рядка цього Ассерт разом із повідомленням про помилку [48]. xUnit++ має схожість з Google Test [29]. CppUnit і Boost Test потребує, щоб тестер вручну відформатував інформації на консоль для належного виведення.

Моки використовуються для моделювання поведінки інших об’єктів, використовуються для видалення залежностей у модульних тестах. М має той самий інтерфейс, що й реальний об’єкт, який він імітує та поводиться подібним чином. Моки є корисною функцією, якщо реальні об’єкти непрактично інтегрувати в модульний тест [30].

Boost Test і Google Test підтримують використання фіктивних об’єктів, тоді як CppUnit і xUnit++ цього не роблять. Boost Test підтримує Моки, але віддає перевагу функціональним можливостям, отриманим від бібліотеки Turtle. Google Test отримує можливість створювати Моки через фреймворк Google Mock. У середовищі, де існує можливість використання Google Mock, Google Test є потужним, оскільки він може легко інтегруватися з Google Mock.

Фікщур дозволяє повторно використовувати однакову конфігурацію тесту для кількох різних тестів. Прилад – це відомий набір об’єктів, який служить основою для

набору тестових випадків, якими вони можуть поділитися. Прилади стають дуже зручними при написанні тестових випадків, які використовують подібні ініціалізації та конфігурації тестів [31].

Фікщур є стандартом архітектури xUnit, який є спільним для всіх фреймворків.

Сют – це сукупність тестових випадків, які призначені для перевірки програми на певний набір поведінок. Сют зазвичай містить інструкції або цілі для кожного набору тестових випадків, але також містить інформацію про конфігурацію системи, яка буде використовуватися на етапі тестування. Сют також може містити стани або кроки передумов, а також описи наступних тестів [32].

Завдяки Сют можливо створити свої тести, потрібно запустити. Це корисна функція, і фреймворки тестування підтримують Сют. Для більшості фреймворків тестування це також включає порядок тестів, за винятком xUnit++. Тести в xUnit++ є рандомізованими, тому неможливо запустити свій тест у певному порядку, якщо у немає окремих тестів у кожному наборі тестів. Хоча добре мати набори тестів навіть для рандомізованих тестів.

Макроси – це ключові слова або виклик функції, які представляють фрагмент коду лише за назвою функції. Коли викликається ім’я макросу, замість нього викликається код. Існують два типи макросів: об’єктні та функціональні. Об’єктно–подібні макроси будуть нагадувати об’єкти даних, а функціональні – виклики функцій [33].

Макроси зручні, і всі вибрані фреймворки тестування мають принаймні один власний макрос. Зазвичай вони використовують твердження як макроси, щоб спростити програмісту написання своїх тестів.

Чекпоінти – це збережені позиції, які можна викликати. Чекпоінт – це макрос у Boost Test, який призначений для введення “іменованої” позиції контрольної точки. Повідомлення з позицією контрольної точки потім зберігається та повідомляється функціями реєстрації винятків (якщо такі трапляються), які потім можуть бути

відформатовані з будь–якого компонента, сумісного зі стандартним вихідним потоком.

Google Test підходить до цього таким чином, можливо використовувати тестові події як контрольні точки для впровадження перевірки витоків ресурсів. І CppUnit, і xUnit++ не мають цієї функції [29, 33].

Винятки – це особливість, яка може ловити очікувані невдалі твердження в тестовому випадку. Твердження винятків призначені для перевірки того, що код генерує (або не створює) виняток, як очікувалося. Вони, як і звичайні твердження, зазвичай записуються як макроси [29]. Виняток – це дійсна функція, яка дозволяє тестувати без збоїв, якщо очікується виняток. Усі вибрані системи тестування підтримують цю функцію.

Дез тести – це тести, які використовуються для перевірки того, чи програма завершується очікуваним чином без виключення. Якщо в тесті щось піде не так, наприклад: твердження перевіряє неправильну умову, тоді програма може перейти в помилковий стан, що, у свою чергу, може призвести до гірших наслідків. Як–от пошкодження пам'яті та діри в безпеці. Дез тести – це перевірки передумов, які призводять до закриття процесів у разі виявлення збоїв. Оскільки певні твердження можуть спричинити невдачу через неправильні умови, дез тести діють як перевірка, що гарантує, що програма знаходиться у відомому стані та не пошкоджена [29].

Лише Boost Test і Google Test, підтримують таку особливість. CppUnit і xUnit++ цього не роблять, що робить Boost Test і Google Test сильнішими кандидатами на вибір.

Шаблон – це механізм на мові програмування C++ і, як правило, підтримується сучасними компіляторами. Це дозволяє програмісту використовувати різні типи для класів і функцій. Потім компілятор генерує конкретні класи та функції, коли пізніше надає конкретні типи як аргументи [34].

CppUnit є єдиною з чотирьох тестових фреймворків, яка не підтримує шаблони тестування. За допомогою шаблонної конструкції можна написати функції та класи

для роботи з загальними типами, що дозволить загальним функціям і класам працювати з багатьма різними типами даних без необхідності переписувати код для кожного з них. Економія часу, даних і полегшує читання коду, що робить інші три системи тестування сильнішими кандидатами.

Для того, щоб тестові фреймворки активно використовувалися в майбутньому, їм необхідно постійно йти в ногу з еволюцією мови C++. Усі розробники вибраних фреймворків усвідомили це і продовжують випускати релізи, що підтримують це. Це може бути особливо важливим для Boost, оскільки їх основна мета – бути сховищем найновіших бібліотек C++ [29].

Щоб платформа тестування була привабливою для широкого кола користувачів, вона повинна охоплювати кілька платформ. Усі вибрані рамки тестування, крім xUnit++, підтримуються для розробки в Windows, OS X та Unix [35, 36, 37].

Для тестування був вибраний саме Google Test фреймворк, оскільки для нього не потрібно завантажувати додаткових бібліотек з пакету Boost.

3.7.Розробка тестів з використанням Google Test фреймворка

Веб–сторінка групи Google для Google Test збирає всю документацію в різні документи з власним змістом: у них є Primer, який охоплює основні поняття; Advance, який охоплює більше функціональних можливостей і розширених концепцій, і Samples, який містить тестові приклади коду. Google Test також має FAQ–документацію та документацію, яка охоплює Xcode, який можна використовувати на Mac. І останнє, але не менш важливе, у них є посібник з розробки, і pump посібник, який описує, як Google сам створює деякі вихідні файли Google Test [26, 37, 38].

Приклади Google Test часто добре коментовані та охоплюють найпростіші варіанти написання тестів. Вони досить добре описують свої функції за допомогою панелі функцій, яка показує виклик функції та короткий опис того, що вона робить. Далі вони розповідають про функції більш детально та як їх можна використовувати [39].

У Google Test є група Google, яка виступає як форум для розробки платформи тестування. Тут кожен може зареєструватися та взяти участь у дискусіях. Дискусії розділені щодо конкретних проблем, з якими стикалися різні люди. Відповідь на поширені проблеми можна знайти в FAQ або на форумах [26].

Нижче наведено процес виконання модульного тесту в стилі чорного ящика на ізольованій капсулі:

– Налаштування тестового випадку
– Налаштувати час затримання
– Надіслати тестове повідомлення до капсули
– Дочекайтеся завершення відправки повідомлення
– Отримати відповідь на повідомлення від капсули
– Перевірити відповідь

Тестовий приклад налаштовується шляхом створення контролера та налаштування середовища, в якому запускається капсула, з подальшим створенням самої капсули. Зазвичай верхня капсула в системі викликається під час запуску системи і існує в середовищі, установленому основним контролером. Потрібно ізолювати капсулу, щоб вона існувала лише в межах тестового випадку. Необхідно внести зміни, щоб видалити створення верхньої капсули в головному контролері та делегувати це завдання кожному тестовому випадку.

Загальний випадок тестування капсули полягає в тому, щоб надіслати їй визначене повідомлення, дочекатися відповіді, а потім перевірити, що повідомлення відповіді є очікуваним. Оскільки капсула існує в області тестового випадку з непідключеними портами, контролер повинен обробляти це спеціально під час

надсилання тестового повідомлення з тестового випадку до капсули. Рішення полягає в тому, щоб безпосередньо приєднати повідомлення до капсули, а потім викликати машину стану капсули за допомогою методу behavior() з наданими даними з повідомлення і, таким чином, уникнути необхідності реалізації подальшої обробки помилок для незв'язаних портів у контролері. Це допомагає розрізняти обидві помилки в конструкції та прийнятний незв'язаний стан тестової капсули.

Коли викликається метод behavior(), він може встановити ланцюжок повідомлень, які необхідно надіслати та відправити, перш ніж капсула надішле повідомлення–відповідь на початковий виклик поведінки. Між відправленням тестового повідомлення та отриманням відповіді контролер повинен виконати цикл відправлення повідомлень, доки не будуть оброблені всі повідомлення, а його черга повідомлень не стане порожньою. Цей сценарій виникає лише у випадках, коли тестована капсула містить інші субкапсули, з якими їй потрібно зв'язатися, перш ніж надіслати остаточну відповідь назад.

Коли контролер зустрічає повідомлення–відповідь під час свого процесу диспетчеризації, він зберігає повідомлення–відповідь і видаляє його з черги. Це повідомлення розпізнається шляхом оголошення в тестовому випадку параметрів очікуваного повідомлення–відповіді. Ці параметри – це капсула–відправник, сигнал, який надсилається, і на який порт він надсилається. Після відправлення всіх повідомлень і завершення роботи капсул, повідомлення можна отримати з контролера. Для повідомлення тестової відповіді звичайний процес надсилання та відправлення буде пропущено, щоб повідомлення було доступним у тестовому випадку.

Оскільки повідомлення–відповідь доступне тестовому прикладу через попередній процес, його можна перевірити за допомогою тестового Ассерт або методів порівняння, наданих структурою модульного тестування. Нарешті, усі налаштування, які були зроблені в тестовому випадку, слід очистити, щоб не впливати на наступні тестові випадки.

3.7.1. Google Test

Google Test вибраний як фреймворк тестування оскільки він підтримує багато функцій та має велику докуменатцію.

Google C++ Testing Framework, або скорочено Google Test, – це платформа модульного тестування для мов C і C++. Створення Google Test виникло з потреб розробки в Google, які не могли задовольнити існуючі рамки модульного тестування [28]. Спочатку розроблений в Google, Google Test – це проект із відкритим кодом, який вперше випущено в середині 2008 року [28].

Приклад використання Google Test показаний в лістингу 2.1

Лістинг 2.1 – Приклад тесту з використанням Google Test

```
#include <gtest/googletest>
TEST(ExampleTestCase, ExampleTest) {
int i = 1; EXPECT-EQ(i == 1); }
```

3.8. Інтерфейс системи

Як інтерфейс для взаємодії з програмним комплексом використовується ThingSpeak. Дана платформа дозволяє будувати додатки на основі даних і чудово підходить для контролю за будинком. ThingSpeak API дозволяє не тільки

відправляти, зберігати і отримувати доступ до даних, але і надає різні статистичні методи їх обробки. На рисунку 3.3 показаний приклад каналу з даними.

ThingSpeak має інтегровану підтримку програмного забезпечення для чисельних обчислень MATLAB від MathWorks, що дозволяє користувачам ThingSpeak аналізувати та візуалізувати завантажені дані за допомогою MATLAB, не вимагаючи придбання ліцензії MATLAB у MathWorks.

ThingSpeak™ Channels ▾ Apps Community Support ▾

ESP8266lightsens

Channel ID: 359160
Author: vasyapupkin1337
Access: Private

Private View | Public View | Channel Settings | Sharing | API Keys | Data Import / Export

Add Visualizations | Data Export

Channel Stats

Created: 3 minutes ago
Updated: 3 minutes ago
Entries: 0

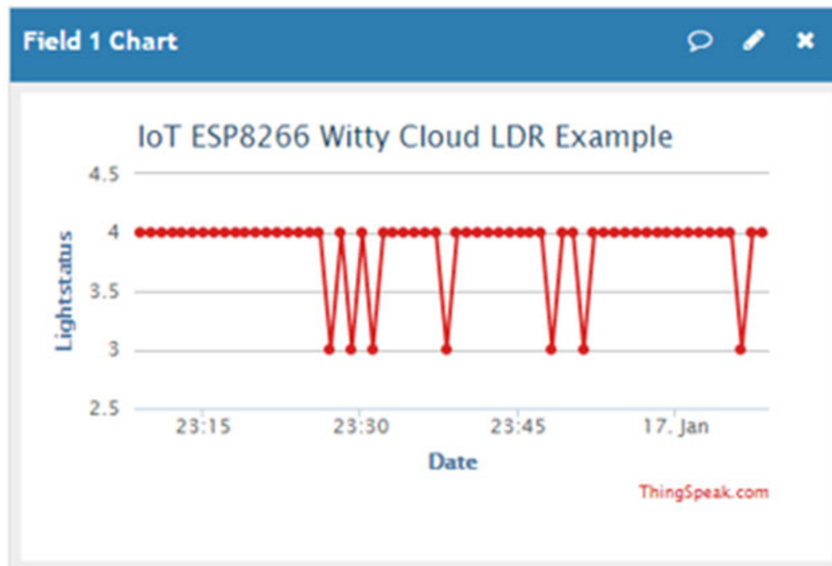

Рисунок 3.3 – Вікно робочого каналу

4. ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1. Охорона праці

Оскільки програмний комплекс Smart House призначений для використання лише на серверному обладнанні, та може виконувати функції аварійного сигналізатора, питання безпечного використання системи, інфраструктури та обладнання серверних приміщень є ключовим для роботи.

Дальше у розділі будуть описані ключові вимоги до інженерної інфраструктури технологічних приміщень (серверних) для розміщення серверного та телекомунікаційного обладнання ВЦТМ.

Серверні мають бути оснащені такими системами:

- система контролю доступу;
- система відеоспостереження;
- система автономного електроживлення; система пожежогасіння;
- система клімат–контролю (вентиляція, температура, вологість); структурованої кабельної системи;
- системи гарантованого електропостачання.

У серверному приміщенні має бути розташовано щонайменше одну серверну стійку вистою 42U для монтажу 19 дюймового мережевого, кросового та комунікаційного обладнання.

Розміри серверного приміщення визначаються в залежності від кількості серверних стійок та іншого габаритного обладнання. Проходи перед та позаду обладнання мають забезпечувати вільний доступ і складати щонайменше 80 сантиметрів. Площа серверного приміщення має складати від 20 м кв. Перебування в серверному приміщенні людей на постійній основі (використання приміщення як

робочого кабінету) заборонено. Також заборонено використання серверних приміщень у якості складських приміщень для зберігання документів, інструменту, будівельних матеріалів, засобів телекомунікацій та інформатизації тощо.

Приміщення серверної згідно з ДБН В.2.5–56:2014 обов'язково підлягають обладнанню системою пожежної сигналізації, тип застосовуваного обладнання системи та місця установки датчиків визначається на етапах технічного проектування відповідно до вимог ДБН В.2.5–56:2014, ДСТУ CEN/TS 54–14:2021, ДСТУ EN 54–13:2014.

Приміщення серверних, які не підлягають обладнанню згідно з ДБН В.2.5–56:2014 автоматичними системами газового пожежогасіння, мають бути оснащені первинними засобами пожежогасіння (пересувними або переносними газовими вогнегасниками).

Кліматичні умови в серверних приміщеннях характеризуються такими показниками: температура повітря та відносна вологість повітря.

У серверному приміщенні необхідно контролювати температуру і вологість, щоб вони постійно перебували в межах рекомендованих робочих діапазонів відповідно ДСН 3.3.6.042–99 і ANSI/TIA–942–A:

температура повітря: від 20ºC (68ºF) до 25ºC (68ºF); відносна вологість повітря: від 40 % до 60 %.

Система кондиціонування повинна забезпечувати безперебійність її роботи у разі зникнення первинного електроживлення серверної чи в момент переключення вводу або запуску резервної бензо або дизель–електростанції. Для цього всі елементи системи кондиціонування мають живитися від системи гарантованого електропостачання.

Рівень резервування системи кондиціонування повинен бути не менше ніж N+1 (бажано – 2N+1), тобто система повинна складатися як мінімум з двох незалежних кондиціонерів, кожен з яких розрахований на самостійне забезпечення заданого мікроклімату в приміщенні.

Система електропостачання серверної за своїм функціональним призначення відноситься до електроприймачів критичної (особливої) групи з неперервним режимом роботи та за ступенем надійності електропостачання належить до електроприймачів першої категорії згідно з ДБН В.2.5–23:2010.

Систему електропостачання необхідно забезпечувати електроенергією від двох незалежних взаєморезервуючих джерел живлення. Переривання їх електропостачання, в разі порушення електропостачання від одного з джерел живлення, можна допускати лише на час автоматичного відновлення живлення.

Для електроприймачів особливої групи першої категорії надійності електропостачання необхідно передбачити додаткове живлення від третього незалежного взаєморезервованого джерела живлення, що забезпечує електропостачання визначеної тривалості. Таким джерелом живлення можуть бути згідно з ДБН В.2.5–23:2010 джерела безперебійного живлення або дизельна (бензинова) електростанція.

Електропостачання серверної згідно з ДБН В.2.5–23:2010 повинно виконуватися від мережі з глухозаземленою нейтраллю 380/220 В та із системою заземлення типу TN–S, яка забезпечує найвищий рівень електробезпеки людей і обладнання.

Опір заземлювального пристрою не повинен перевищувати 4 Ом і 8 Ом відповідно для лінійних напруг 380В і 220В джерела трифазного струму або 220В і 127В джерела однофазного струму згідно з Правилами улаштування електроустановок, затвердженими наказом Міністерства енергетики та вугільної промисловості України від 21.07.2017 № 476. Цей опір необхідно забезпечувати з урахуванням використання всіх заземлювачів, приєднаних до робочого заземлення.

Таким чином розроблена система є безпечною при дотримані всіх вимог.

4.2.Безпека в надзвичайних ситуаціях

Оскільки керування програмним комплексом Smart House відбувається за допомогою персонального комп’ютера, в розділі описані фактори що впливають на функціональний стан користувачів комп’ютерів.

Перед тим як описати фактори впливу, потрібно визначити що таке функціональний стан.

Функціональний стан – інтегральний комплекс різних характеристик, процесів, властивостей і якостей людини, які прямо або побічно обумовлюють виконання діяльності.

Реальний рівень функціонального стану є результатом складної взаємодії багатьох факторів, внесок яких визначається конкретними умовами діяльності індивіда.

Аналіз функціонального стану працюючої людини в умовах реальної діяльності виходить за рамки тільки фізіологічних уявлень і передбачає розробку психологічних і соціально–психологічних аспектів цієї проблематики. Кожен конкретний стан людини можна описати за допомогою різноманітних проявів:

- зміни у функціонуванні різних фізіологічних систем;
- зрушення в протіканні основних психічних процесів;
- суб’єктивні переживання (втома, млявість, безсилля, нудьга, апатія, сонливість, тривога, нервозність, переживання небезпеки і страху);
- зміни на поведінковому рівні (кількісні показників виконання певного виду діяльності, продуктивність праці, інтенсивність і темп виконання роботи, число збоїв і помилок та ін.)

Поняття функціонального стану вводиться для характеристики ефективнісної сторони діяльності або поведінки людини. Цей аспект розгляду проблеми передбачає насамперед вирішення питання про можливості людини, що знаходиться в тому чи іншому стані, виконувати конкретний вид діяльності.

Тому стан людини не є простою зміною в протіканні окремих функцій чи процесів, а є складною системною реакцією індивіда.

Визначення та вивчення факторів, що впливають на функціональний стан користувачів комп'ютерів дозволить виділити основні причини виникнення станів напруженості, стомлення, стресу і здійснити відповідні профілактичні заходи.

Трудова діяльність користувачів комп'ютерів відбувається у певному виробничому середовищі, яке впливає на їх функціональний стан. Найбільш значимі – фізичні фактори виробничого середовища, до яких належать електромагнітні хвилі різних частотних діапазонів, електростатичні поля, шум, параметри мікроклімату та ціла низка світлотехнічних показників. Вплив хімічних та, особливо, біологічних факторів виробничого середовища на користувачів комп'ютерів – значно менший.

Трудовий процес суттєво впливає на психофізіологічні можливості користувачів комп'ютерів, оскільки їх діяльність характеризується значними статичними фізичними навантаженнями; недостатньою руховою активністю; напруженнями сенсорного апарату, вищих нервових центрів, які забезпечують функції уваги, мислення, регуляції рухів. Окрім того, трудовий процес користувачів комп'ютерів відзначається значними інформаційними навантаженнями.

Професійні якості та виробничий досвід, які визначають внутрішні засоби діяльності, обумовлюють надійну та безпомилкову діяльність користувачів комп'ютерів, дозволяють знаходити безпечні методи розв'язання виробничих завдань навіть у нестандартних ситуаціях.

Зовнішні засоби діяльності, які в основному визначаються ергономічними показниками щодо організації робочого місця, форми та параметрів його елементів, просторового розташування основного і допоміжного устаткування, можуть суттєво

знизити фізичні та психофізіологічні навантаження, що діють на користувачів комп'ютерів.

Оскільки робота користувачів комп'ютерів частіше за все проходить за активної взаємодії з іншими людьми, то виникають питання раціоналізації міжособових відносин. Цей комплекс питань порушує як психологічні, так і соціально–психологічні аспекти трудових взаємовідносин, які також є факторами "ризику", що відчутно впливають на функціональний стан користувачів комп'ютерів.

Таким чином, на користувача комп'ютера впливає комплекс факторів. Урахування ступеня та якості впливу цих факторів на функціональний стан дозволяють розробити заходи та засоби щодо забезпечення безпеки, підвищення працездатності та збереження здоров'я користувачів комп'ютерів.

ВИСНОВКИ

В процесі виконання кваліфікаційної роботи розроблено програмний комплекс “Smart House”, систему для зручного спостереження за приватною власністю.

Проект написаний на мові програмування C++ з використанням WinSock та бібліотеки Googletest. Також написана пояснювальна записка до роботи.

В аналітичній частині описується описані дослідження доцільності програмної системи, обґрунтування актуальності теми проекту, постановка завдання, опис ключових варіантів використання та автоматизація процесу розробки. Також описана методологія Agile та життєвий цикл програмного забезпечення.

В теоретичній частині описується модель розробки програмного забезпечення, вибір сервера безперервної інтеграції та архітектура програмної системи, де описана модель Entity–Component–System її особливості та недоліки, також.

В практичній частині описано методи тестування, проаналізовано фреймворки для написання тестів та показано інтерфейс системи.

Основні питання охорони праці та техніки безпеки розглянуто в четвертому розділі.

Під час виконання роботи, було виявлено позитивні та негативні сторони проектування програмного комплексу та вибраних інструментів розробки. Вибраний метод проектування дозволяє легко розширювати систему та забезпечує високу продуктивність, при цьому створюючи великий та не простий до розуміння інтерфейс і вимагає високої кваліфікації до програміста.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. “Buildings Energy Data Book”. US Department of Energy, March 2012 http://buildingsdatabook.eren.doe.gov/TableView.aspx?table=2.1.5
2. “Annual Energy Review 2011”. U.S. Department of Energy, September 2012 http://www.eia.gov/aer
3. “Energieverbrauch der privaten Haushalte für Wohnen”. Statistisches Bundesamt, Wiesbaden, November 2012 https://www.destatis.de/DE/ZahlenFakten/GesamtwirtschaftUmwelt/Umwelt/UmweltoekonomischeGesamtrechnungen/EnergieRohstoffeEmissionen/Tabellen/EnergieverbrauchHaushalte.html
4. “Final energy consumption, by sector.” Eurostat European Commission, April 2012 http://epp.eurostat.ec.europa.eu/portal/page/portal/energy/data/main_tables
5. Angelo Baggini, Lyn Meany. “Application Note Building Automation and Energy Efficiency: The EN 15232 Standard”, European Copper Institute, May 2012 http://www.leonardo-energy.org/good-practice-guide/building-automation-and-energy- efficiency-en-15232-standard
6. Botta A, de Donato W, Persico V, Pescapé A. Integration of cloud computing and internet of things: A survey. Future Generation Computer Systems. 2016;:684–700
7. Soliman M, Abiodun T, Hamouda T, Zhou J. Lung C–H.Smart home: Integrating internet of things with web services and cloud computing. In: International Conference on Cloud Computing Technology and Science; IEEE. 2013
8. A. Martin, Entity Systems are the future of MMOG development, 3 September 2007, available (referred to 03.02.2019): http://t-machine.org/in-

dex.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/.

9. T. Stein, The Entity-Component-System - C++ Game Design Pattern, 21 November 2017, available (referred to 20.1.2019): https://www.gamedev.net/articles/programming/general-and-gameplay-programming/the-entity-component-system-c-game-design-pattern-part-1-r4803/.
10. V. Romeo, Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library, avail- able (referred to 11.2.2019): https://github.com/SuperV1234/bcs_thesis/blob/master/final/rev1/web_version.pdf, Italy, 2015, pp.145.
11. M. Infantino, Entity-Component Systems & Data Oriented Design In Unity, 19 August 2018, available (referred to 28.4.2019): https://github.com/LifeIsGoodMI/ECS-And-DoD-In-Unity/blob/master/ECS_Unity_Article.pdf.
12. N. Llopis, Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot with OOP), 4 December 2009 [Online], available (referred to 28.4.2019): http://gamesfromwithin.com/data-oriented-design.
13. Pan, J. 2015. Software Testing. [ONLINE] Available at: http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/. [Accessed 13 October 2015].
14. Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A. (2007, September) Contract Driven Development = Test Driven Development – Writing Test Cases. Paper presentet at Proceedings of ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2007. Dubrovnik, Croatia. doi:10.1145/1287624.1287685
15. Gelperin, D.; B. Hetzel (1988, June). The Growth of Software Testing. Communications of the ACM Volume 31. ISSN 0001-0782. doi:10.1145/62959.62965.

16. Pfleeger, S. Atlee, J. 2010. Software Engineering. 4th Edition. Pearson. ISBN: 0-13-814181-9.
17. Woods, A. 2015. Operational Acceptance Test - White Paper. Capgemini. [ONLINE] Available at: http://www.scribd.com/doc/257086897/Operational-Acceptance-Test-White-Paper-2015- Capgemini. [Accessed 13 October 2015].
18. Luo, L. (n.d) Software Testing Techniques. Carnegie Mellon University [ONLINE] Available at: http://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf. [Accessed 28 september 2015].
19. Bourque, P. Fairley, R. 2004. Guide to the Software Engineering Body of Knowledge. Version 3.0. IEEE Computer Society. Pages: 4-1, 4-2, 4-5, 4-6, 13-24, 13-26
20. Williams, L. 2006. Testing Overview and Black-Box Testing Techniques. [ONLINE] Available at: http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf. [Accessed 16 June 2015].
21. P. Bourque, R.E. Fairley. 2014. Guide to the Software Engineering Body of Knowledge. Version 3.0. IEEE Computer Society. Pages: 4-1, 4-2, 4-5, 4-6, 13-24, 13-26
22. Kolawa, A. Huizinga, D. 2007. Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press. ISBN: 9780470042120.
23. Xie, T. (2007, May). Towards a Framework for Differential Unit Testing of Object-Oriented Programs. Paper presented at Automation of Software Test, 2007. AST '07. Second International Workshop on. Minneapolis, USA. doi:10.1109/AST.2007.15
24. Fowler, M. 2006. Xunit. [ONLINE] Available at: http://www.martinfowler.com/bliki/Xunit.html. [Accessed 30 July 2015].

25. Meszaros, G. 2007. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley Professional. ISBN: 0131495054.
26. Hamill, P. 2004. Unit Test Frameworks. First Edition. O'Reilly Media. ISBN: 0596006896.
27. Sommerlade, E. Feathers, M. Lacoste, J. Lepilleur, B. Bakker, B. Robbins, S. 2015. CppUnit Documentation. [ONLINE] Available at: http://people.freedesktop.org/~mmohrhard/cppunit. [Accessed 12 October 2015].
28. Mohrhard, M. 2013. cppunit framework. [ONLINE] Available at: http://lists.freedesktop.org/archives/libreoffice/2013-October/056919.html. [Accessed 12 October 2015].
29. Mohrhard, M. 2013. Cppunit 1.13.2 released. [ONLINE] Available at: https://mmohrhard.wordpress.com/2013/11/12/cppunit-1-13-2-released/. [Accessed 21 May 2015].
30. Google Project Hosting. 2015. googletest FAQ - Tips and Frequently-Asked Questions about Google C++ Testing Framework [ONLINE] Available at: https://code.google.com/p/googletest/wiki/FAQ. [Accessed 11 May 2015].
31. moswald – Bitbucket. 2015. xUnit++ wiki — Home. [ONLINE] Available at: https://bitbucket.org/moswald/xunit/wiki/Home. [Accessed 11 May 2015].
32. moswald – Bitbucket. 2015. xUnit++ wiki — Issues. [ONLINE] Available at: https://bitbucket.org/moswald/xunit/issues?status=new&status=open. [Accessed 10 February 2016].
33. moswald – Bitbucket. 2015. xUnit++ wiki — Downloads. [ONLINE] Available at: https://bitbucket.org/moswald/xunit/downloads. [Accessed 10 February 2016].
34. moswald – Bitbucket. 2015. xUnit++ wiki — Overview. [ONLINE] Available at: https://bitbucket.org/moswald/xunit/overview. [Accessed 10 February 2016].
35. moswald – Bitbucket. 2015. xUnit++ wiki — Tests. [ONLINE] Available at: https://bitbucket.org/moswald/xunit/wiki/Tests.wiki. [Accessed 11 May 2015].

36. Tutorialspoint – Simply Easy Learning. 2015. Assertion Testing. [ONLINE] Available at: http://www.tutorialspoint.com/software_testing_dictionary/assertion_testing.htm. [Accessed 28 May 2015].
37. Google Project Hosting. 2015. googletest V1_7_Primer - Getting started with Google C++ Testing Framework. [ONLINE] Available at: http://code.google.com/p/googletest/wiki/V1_7_Primer. [Accessed 11 May 2015].
38. moswald – Bitbucket. 2015. xUnit++ wiki — RunningTests. [ONLINE] Available at: https://bitbucket.org/moswald/xunit/wiki/RunningTests.wiki. [Accessed 28 May 2015].
39. Chaffe, A. Pietri, W. 2015. Unit testing with mock objects. IBM. [ONLINE] Available at: http://www.ibm.com/developerworks/java/library/j-mocktest/index.html. [Accessed 11 May 2015].

додатки