НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

УДК

«ПОГОДЖЕНО»	**«ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ»**
Декан факультету інформаційних технологій	Завідувач кафедри комп'ютерних наук
Болбот І.М., д..н., професор	Голуб Б.Л., к.т.н., доцент
_____	_____
____ _____ 202_ р.	**____ _____ 202_ р.**

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему Програмне забезпечення системи управління автономними динамічними об'єктами

Спеціальність Комп'ютерні науки 122
(код і назва)

Освітня програма Інформаційні управляючі системи та технології
(назва)

Орієнтація освітньої програми _____
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

_____ _____ _____
(науковий ступінь та вчене звання) (підпис) (ПІБ)

КЕРІВНИК МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ

д.т.н., професор _____ Хиленко В.В
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ **Виноградов Д.О.**
(підпис) (ПІБ студента)

КИЇВ-2024

Програмне забезпечення системи управління автономними динамічними об'єктами

Вступ

Упродовж останніх років логістика стала визначальним фактором успішності у глобально розподілених виробничих процесах завдяки своїй міжсекторній функції. Зростання тривалості життєвого циклу продуктів, швидкі зміни у структурі компаній та інформаційних потоках змінюють вимоги до логістичних процесів. Зі зменшенням вертикальної інтеграції виробництва і тенденцією до глобально розподілених виробництв, значимість логістики та проєктування логістичних процесів зростає. Як наслідок, важливість логістики зростає, а також виникає потреба у нових концепціях планування та контролю логістичних процесів.

З одного боку, збільшення складності міжорганізаційних структур і відносна нестача логістичної інфраструктури призводять до посиленої експлуатації наявних логістичних процесів. З іншого боку, спостерігається спеціалізація та інтермодалізація транспортних шляхів і відповідних перевізників. Ці фактори, у поєднанні зі змінами умов на ринку споживачів, мають суттєвий вплив на планування та контроль логістичних процесів в такому динамічному середовищі.

Результуюча динамічна і структурна складність логістичних мереж ускладнює своєчасне надання всієї необхідної інформації для центрального планування та контролю як під час фази планування, так і для реагування на вхідну інформацію під час фази виконання. Один із можливих підходів до подолання цих викликів полягає у розробці автономних логістичних процесів, які мають здатність і можливості для децентралізованої координації та прийняття рішень.

Автономні суб'єкти діють автономно і керуються індивідуально раціональними цілями. Базовим припущенням є те, що вони виступають як раціональні ухвалювачі рішень у сенсі теорії ігор, причому кожен намагається максимізувати свою власну функцію корисності.

У подальшому ми глибше дослідимо питання складності та динаміки в сучасних логістичних системах і можливості, які виникають завдяки впровадженню автономії. Одним із підходів до вирішення викликів у наявних і прийдешніх проблемах у логістиці є концепція автономних логістичних процесів, представлених автономними логістичними об'єктами. Автономія в логістичних процесах визначається як здатність логістичних об'єктів обробляти інформацію, приймати рішення та здійснювати їх самостійно. Автономне управління логістичними процесами може бути реалізоване через децентралізовані системи контролю, які вибирають альтернативи автономно або за певної логічної підтримки, приймаючи рішення в межах визначених цілей.

Актуальність дослідження. Актуальність проблеми управління автономними динамічними об’єктами обумовлена стрімким розвитком сучасних технологій і зростаючими вимогами до ефективності та безпеки таких систем. Автономні транспортні засоби, роботи та інші динамічні об’єкти знаходять все ширше застосування у різних галузях, від логістики та промисловості до військових технологій та космічних досліджень. Складність їх функціонування та взаємодії з навколишнім середовищем потребує розробки надійних і ефективних алгоритмів управління. Крім того, врахування факторів непередбачуваності та адаптації до змінних умов підкреслює важливість системного аналізу цієї проблеми.

Тема дослідження. Системний аналіз проблеми управління автономними динамічними об’єктами.

Мета дослідження. Метою дослідження є розробка та вдосконалення методів системного аналізу і управління автономними динамічними об’єктами, що дозволяє підвищити ефективність їх роботи в умовах невизначеності та змінного зовнішнього середовища. Це включає дослідження алгоритмів самонавчання, адаптивного управління та моделей прогнозування поведінки об’єктів у реальному часі.

Практичне значення дослідження. Результати дослідження можуть бути застосовані в різних сферах, таких як розробка автономних транспортних засобів, безпілотних літальних апаратів, роботизованих систем, а також у військових та космічних технологіях. Використання запропонованих методів системного аналізу та управління дозволить підвищити точність, надійність та безпеку функціонування автономних систем у складних умовах, що може мати значний економічний і соціальний ефект.

1 Системний аналіз проблеми управління автономними динамічними об’єктами

1.1 Поняття і види автономних динамічних об’єктів, а також особливості процесів управління ними

Складність можна зрозуміти як взаємодію між складністю та динамікою. Оскільки описані підходи щодо складності стосуються лише окремих аспектів, таких як структура розглянутого об'єкта, вони здаються недостатніми для повного розуміння терміна "складність" у контексті логістичних систем, особливо виробничих систем.

Існують різні підходи, які спрямовані на пояснення складності, що також застосовуються до логістичних систем. Ми підсумовуємо їх у дві основні категорії: підходи, що базуються на елементах або відносинах, як-от підхід Бар-Яма, який вказує на (1) елементи (і їх кількість), (2) взаємодії (і їхню силу), (3) формування/роботу (і їх часові масштаби), (4) різноманітність/варіативність, (5) середовище (та його вимоги), і (6) діяльності (та їх цілі) як характерні властивості складної системи, тоді як Оттіно, як представник класу, котрий визначає складність через її властивість виникнення, зазначає: «Складна система - це система з великою кількістю елементів або агентів, які можуть організовуватись без застосування зовнішнього принципу організації» [1].

Динаміка логістичної системи характеризується її часовою поведінкою. Така система зазнає постійних змін на мікро- та мезо-рівнях, хоча може перебувати у сталому стані на макро-рівні. Кількість можливих станів, що виникають через події, які впливають на систему, та взаємодію елементів, що в неї вбудовані, є важливим показником для оцінки динаміки логістичної системи. Це настільки ж важливо для планування та реалізації логістичних процесів, як і складність самої системи.

Філіпп та інші вважають, що необхідно визначити різні категорії складності та пов'язати їх одна з одною, щоб отримати комплексний опис

складності. Вони розділяють складність на три категорії: організаційна складність, що відображає складність елементів і відносин; часова складність, яка переосмислює динаміку як форму складності; і системна складність, що встановлює межу між системою та середовищем. Однак у цьому підході не враховано аспект виникнення складності.

Тим не менш, складність логістичної системи і її динаміка, разом із загальною поведінкою системи, не дозволяють зробити висновки щодо чутливості системи до несправностей окремих елементів або їхніх зв'язків, а також до ненавмисних або зловмисних втручань ззовні.

Відповідно до теорії Лумана, що лише комплексність системи може зменшити складність, до якої система піддається, ми надаємо можливість логістичній системі вирішувати зростаючу складність у своєму середовищі, збільшуючи складність самої системи та керуючи цією складністю за допомогою технологій. Цей підхід зміцнюється розробкою нових інформаційно-комунікаційних технологій (ІКТ).

Швидкий і безперервний розвиток сучасних ІКТ, таких як телематика, мобільна передача даних і транспондерні технології, відкриває нові можливості для створення інтелектуальних логістичних систем, здатних виконувати вимоги автономних логістичних процесів. Однак для забезпечення керованості динамічною логістичною системою технологічний розвиток має не тільки забезпечувати швидкі автономні заміни стандартних логістичних операцій, але й враховувати, що впровадження автономії вплине на операційне та стратегічне управління логістичними послугами [2].

Оскільки динамічна і структурна складність логістичних мереж значно ускладнює надання всієї необхідної інформації для централізованого планування та контролю, автономність логістичних одиниць є перспективним підходом.

Ця автономність може бути реалізована шляхом розробки адаптивних логістичних процесів, що включають автономні можливості для

децентралізованої координації автономних логістичних одиниць у гетерархічній структурі. Автономія описує процеси децентралізованого прийняття рішень у гетерархічних структурах. Вона передбачає, що взаємодіючі одиниці мають здатність і можливість самостійно приймати рішення в недетерміністичних системах. Автономія дозволяє і вимагає нових стратегій керування та автономних децентралізованих систем управління логістичними процесами. У цьому контексті такі аспекти, як гнучкість, адаптивність і реактивність на динамічні зовнішні впливи, при збереженні цілей, є центральними.

Інтеграція стратегічного і тактичного планування у поєднанні з великою кількістю актуальних даних та можливої комунікації між одиницями системи дозволяє системі діяти автономно і, можливо, компенсувати тимчасову чи постійну несправність однієї з одиниць або відносин між кількома одиницями. Наслідком автономних дій задіяних одиниць є зміщення відповідальності за реалізацію рішень від центральної системи прийняття рішень — чи вона технічна, чи людська — до окремої логістичної одиниці. Це потрібно враховувати при розробці концепції управління автономними логістичними об'єктами та складності всієї системи.

Поняття і види автономних динамічних об’єктів, а також особливості процесів управління ними

Особливості управління автономними динамічними об'єктами:

1. Самоадаптація: Об'єкти можуть адаптувати свої дії в залежності від умов середовища. Наприклад, робот-автомобіль автоматично коригує свій маршрут на основі змін дорожньої обстановки.
2. Реакція на непередбачувані фактори: Автономні об'єкти мають здатність оперативно реагувати на зміни або непередбачувані події, наприклад, під час польоту дрона він може уникнути перешкод, яких немає на попередніх картах.

3. Оптимізація процесів: Управління автономними системами базується на постійному пошуку оптимальних рішень для досягнення мети. Це може стосуватися як ефективного використання ресурсів, так і максимізації результату при обмежених ресурсах.
4. Інтелектуальне управління: Багато автономних об'єктів використовують штучний інтелект, що дозволяє їм навчатися на основі досвіду і поліпшувати свої результати у майбутньому. Наприклад, промислові роботи можуть оптимізувати процеси на основі аналізу історичних даних.

Автономні системи створюють нові підходи до управління завдяки тому, що працюють без постійного людського контролю, що дозволяє зменшувати вплив людського фактора і підвищувати ефективність їх роботи.

Найгірший приклад автономного робота

Верстати на автомобільних складальних лініях часто плутають з роботами. Вони можуть виглядати як роботи, але насправді це фрезерні верстати з комп'ютерним управлінням, які називаються верстатами з ЧПУ.

На відміну від справді автономних роботів, ці промислові машини попередньо запрограмовані на виконання повторюваних дій. Вони не можуть реагувати.

Наприклад, що станеться, якщо один із так званих роботів, відповідальних за те, щоб покласти запасне колесо в багажник автомобіля, випадково зіткнеться з ситуацією, коли багажник буде зачинений? Чи припинить цей «робот» класти шину?

Напевно, ні. Навпаки, він продовжив би виконувати запрограмоване завдання і міг би знищити шину, врізавшись у кришку багажника. Якби машина була справжнім автономним роботом, вона б не встановлювала шини і знала, що багажник зачинений.

Чому Roomba — справжній автономний робот

Щоб повністю зрозуміти автономних мобільних роботів, корисно побачити їх у дії. Найвідомішим автономним роботом, доступним сьогодні, є Roomba, який відомий своєю незалежністю та продуктивністю. Roomba має функції, корисні у складському та промисловому середовищі, які роблять AMR більш доступною технологією.

Roomba може приймати рішення, уникати перешкод і діяти відповідно до того, що він сприймає в навколишньому середовищі. Бот працює в приміщенні самостійно і без оператора.

Датчики допомагають Roomba розуміти своє оточення, приймати рішення і діяти відповідно до них. Складські роботи працюють за тими ж правилами.

Складські роботи діють за тими ж правилами. Якщо є перешкоди, наприклад, палети, вони уникають їх і об'їжджають. Рішення для автоматизації продовжують працювати без втручання людини.

Простіше кажучи, автономні роботи - це роботи, які самостійно вирішують, яку дію зробити на основі інформації, яку вони сприймають, щоб підвищити продуктивність. Щоб дізнатися більше про автономних роботів та їх численні застосування, зв'яжіться з нами вже сьогодні. Якщо ви не впевнені, як робототехніка може допомогти вам, погляньте на ці вісім чудових прикладів автономних роботів.

Застосування робототехніки в автономних системах

Автономні мобільні роботи для переміщення матеріалів

Хоча можливості AMR продовжують розвиватися, найпростішим і найпоширенішим застосуванням є переміщення матеріалів. Щодня AMR можуть переміщувати незліченну кількість замовлень на склади і в пункти дистрибуції. Переміщення є трудомістким завданням у ланцюгу поставок, і використання роботів для цього - один з найпростіших способів звільнити працівників для більш важливих завдань, не порушуючи при цьому робочий процес [3].

Мобільні автономні роботи для електронної комерції

AMR для додатків електронної комерції можуть мати різні форми, наприклад, мобільні візки, мобільні операції і т.д. Оскільки платформа AMR може включати в себе цілий ряд аксесуарів, її гнучкість робить її ідеальною для різних застосувань, навіть для спеціалізованого використання, наприклад, для обробки і сортування.

Сьогодні ми бачимо, що AMR використовуються в таких завданнях, як:

Виконання замовлення

Обробка повернень і обробки матеріалів

Транспортування та сортування сировини

Сортування посилок

Управління запасами

AMR для складського господарства

Сучасні склади та розподільчі центри величезні - деякі з них мають площу понад мільйон квадратних метрів - і AMR ідеально підходять для автоматизації складських операцій, таких як обробка важких вантажів. Використання AMR для виконання основних складських завдань скорочує час на поїздки працівників і дозволяє їм зосередитися на більш важливих завданнях.

Однією з унікальних особливостей AMR є їхня здатність «бачити» і знаходитися у відкритому просторі: вони використовують системи технічного зору, такі як лазери, для сканування навколишнього середовища, а вбудовані системи аналізують дані з датчиків для виявлення перешкод і безпечної навігації. Однак на багатьох величезних складах відсутні фіксовані елементи, такі як стіни і стовпи, які потрібні AMR для ефективної навігації. У таких умовах безпілотники з навігаційними системами, розробленими спеціально для складських операцій, мають важливе значення.

Ще одним важливим завданням при складуванні та дистрибуції є штабелювання палет. Це монотонна і повторювана робота, яка піддається

автоматизації. Щоб прискорити це завдання і звільнити працівників для виконання інших завдань, AMR тепер застосовується для палетування: За допомогою платформ AMR, підйомних платформ і роботизованих маніпуляторів палетування можна майже повністю автоматизувати. Роботи-палетайзери можуть виконувати всі етапи процесу - завантаження, переміщення і розвантаження - автономно, ефективно і точно [4].

AMR та мобільні маніпулятори для виробництва

Універсальність AMR робить їх ідеальними для постійно мінливого і динамічного світу виробництва. Розроблені для легкого встановлення та використання операторами на існуючих підприємствах, AMR дозволяють компаніям будь-якого розміру використовувати потужність маніпуляторів для нескінченного різноманіття завдань.

Крім обробки деталей і готової продукції, AMR, інтегровані з такими аксесуарами, як конвеєри і маніпулятори роботів, можуть допомогти у виробничому процесі. Наприклад, AMR з роботизованими маніпуляторами можуть додавати можливість динамічного переміщення в різні місця для сортування, відбору та пакування продукції.

Статичні конвеєри вже давно використовуються на лініях для прискорення виробництва та сортування, а додавання конвеєрів до AMR означає, що функція конвеєра стає гнучкою та мобільною. AMR з вбудованими конвеєрами можуть бути підключені до статичних конвеєрів для більш ефективного переміщення продукції по підприємству.

AMR з навісним обладнанням, яке може піднімати візки і підключатися до візків, дозволяють роботу завантажувати і розвантажувати вантажі, а в деяких випадках підключатися до візків без участі людини, але поєднання переміщення візків і завантаження/розвантаження в одному AMR є відносно новою функцією, потенційне застосування автономних роботів буде розширюватися і надалі.

AMR для центрів обробки даних

Безпечне та автономне транспортування є невід'ємною частиною роботи центрів обробки даних і дослідницьких центрів, що створює нові можливості для застосування AMR. Автономні роботи, обладнані ящиками і шафами, можуть використовуватися для безпечного транспортування цінних матеріалів і забезпечення дотримання відповідних протоколів CoC. Вони також можуть миттєво, точно і легко документувати процеси.

AMR в охороні здоров'я

З розвитком можливостей і простоти використання AMR багато галузей знаходять інноваційні способи застосування роботів. Автономні роботи зараз все частіше використовуються в секторі охорони здоров'я для виконання різноманітних завдань.

Перш за все, AMR є корисним інструментом для оптимізації поводження з матеріалами та медикаментами в медичних установах. Це особливо важливо в інфекційних відділеннях, де медсестри уникають частого контакту з потенційними забруднювачами, але при цьому забезпечують належне лікування пацієнтів. По-друге, медичні AMP також можуть використовуватися для санітарної обробки. Роботи можуть бути оснащені ультрафіолетовими лампами для знищення вірусів і дезінфікуючими спреями для очищення приміщень і територій, не наражаючи людей на потенційну небезпеку [5].

AMR в біотехнології

В умовах швидкозростаючого біофармацевтичного ринку біотехнологічні компанії змушені дотримуватися суворо регламентованих виробничих процесів, що означає значні витрати часу. Наприклад, підтримка процесів відбору зразків і культивування клітин є трудомістким процесом і вимагає безперервного моніторингу в режимі 24/7. Автономні мобільні роботи в поєднанні з роботизованими маніпуляторами можуть бути використані для перевірки цінних вхідних даних процесу, виконання

регулярних завдань моніторингу та безпечного управління видаленням відходів з виробничої лінії [6].

Завдяки тому, що AMR виконує повторювані завдання, оператори можуть зосередитися на критично важливих етапах біофармацевтичного виробництва, таких як моніторинг параметрів росту, безперервне тестування і внесення необхідних коригувань по мірі розвитку.

AMR для досліджень і розробок

AMR використовується в дослідженнях і розробках для мінімізації виснажливого виконання повторюваних тестувань та інших інженерних вимог. Крім того, AMR все частіше інтегрується в самі дослідження.

Наприклад, однією з найгарячіших сфер інновацій є розробка сенсорних і роботизованих маніпуляційних технологій. У міру розвитку цих досліджень дослідники шукають способи мобілізувати ці технології, але багато організацій не мають часу або фінансування для створення власних платформ. За допомогою гнучкої AMR датчики і маніпулятори, що використовуються в дослідженнях, можуть бути легко інтегровані в мобільну платформу, забезпечуючи просту у використанні автономну мобільність цих нових технологій і значно скорочуючи час і вартість процесу розробки для компаній і дослідницьких організацій.

Базові компоненти автономних ботів

Основні компоненти автономної поведінки, описані вище, включають три ключові поняття: сприйняття, прийняття рішень і активація.

Сприйняття

У людському сприйнятті основну роль відіграють наші п'ять органів чуття: зір, слух, дотик, смак та нюх. Це досягається через зорові, слухові та механічні рецептори, що взаємодіють із навколишнім світом. Подібним чином у роботів сприйняття здійснюється за допомогою численних сенсорних пристроїв. Лазерні сканери та стереокамери функціонують як аналоги людських очей, датчики удару нагадують відчуття дотику шкірою,

сенсори сили і крутного моменту моделюють напругу м'язів, а спектрометри виконують роль аналогів нюху. Ці пристрої введення дозволяють роботам орієнтуватися у своєму середовищі. Однак, як і у випадку з людьми, варто розглянути можливість віддаленого збору інформації, наприклад, з Інтернету. Цю концепцію можна порівняти з Інтернетом речей, який постає як безмежний океан сенсорів з довгими кабелями, що надають дані роботам, здатним їх використовувати.

Рішення

Для людини основну частину рішень ухвалює мозок; в інших випадках це може бути наш "інтуїтивний відчуття" або навіть нервова система. Мозок відповідає за високорівневі рішення, як-от вибір місця призначення. Проте, інколи наша біологія випереджає мозок, і тіло реагує ще до того, як свідомість усвідомлює ситуацію. Такі рефлекторні дії, як закривання повік для захисту ока від сміття, відбуваються швидше та незалежно від згоди мозку, щоб захистити нас.

Автономні роботи побудовані за схожою схемою прийняття рішень. Їх "мозком" є комп'ютер, який ухвалює рішення на основі поставлених завдань та отриманих даних. Водночас ці роботи мають функціонал, подібний до нервової системи людини: їхні системи безпеки діють швидко й незалежно від "мозку". Насправді у роботів "мозок" працює під контролем системи безпеки. У контексті автономних роботів цю "неврологічну" функцію називають вбудованою системою, яка діє швидше й має більше повноважень, ніж комп'ютер, який займається реалізацією плану і аналізом даних. Завдяки цьому робот може миттєво зупинитися при виявленні перешкоди на шляху, при виникненні внутрішньої проблеми або у випадку натискання кнопки екстреної зупинки.

Приведення в дію

У людському організмі м'язи виконують роль основних механізмів руху, буваючи різноманітних форм та виконуючи широкий спектр функцій

— від простих дій, як-от захоплення кавової чашки, до складних фізіологічних процесів, таких як серцебиття і перекачування крові. Аналогічно, роботизовані системи теж оснащено приводами, за роботу яких відповідає певний вид двигуна. Незалежно від того, чи це колісний механізм, лінійний привід або гідравлічний циліндр, завжди наявний двигун, що перетворює енергію на механічний рух.

Наш повністю оснащений парк автономних мобільних роботів обладнано сенсорами з використанням технологій LiDAR і комп'ютерного зору, системами прийняття рішень і приводами у формі колес. Вони здатні функціонувати в будь-якій зоні вашого складу, забезпечуючи ефективне переміщення матеріалів. Це усуває непродуктивний час, що витрачається на переміщення людьми, значно підвищує продуктивність та скорочує тривалість робочих циклів [7].

1.2 Існуючі види програмного забезпечення для управління автономними динамічними об’єктами

Програмне та апаратне забезпечення, що забезпечує функціонування автономних транспортних засобів, є вельми складним і багатогранним. Завдяки використанню складних алгоритмів машинного навчання, зокрема нейронних мереж та штучного інтелекту, інженери зуміли відкрити дорогу до безпілотних автомобілів.

Ступені автономності

Згідно з даними SAE International, існує шість рівнів автоматизації водіння:

Рівень 0 – функції обмежуються наданням попереджень та короткочасної допомоги, але водій все ще повинен контролювати транспортний засіб.

Рівень 1 – забезпечення підтримки кермування або гальмування чи прискорення. До функцій цього рівня належать центрування в смузі або адаптивний круїз-контроль.

Рівень 2 – незважаючи на те, що водій продовжує контролювати автомобіль, функції рівня 2 забезпечують одночасну підтримку кермування та гальмування або прискорення. Наприклад, автомобіль, який має системи центрування у смузі руху та адаптивного круїз-контролю, належить до транспортних засобів рівня 2 за класифікацією SAE.

Рівень 3 – з переходом на цей рівень водій більше не управляє транспортним засобом. Однак ви повинні бути готові взяти на себе керування у специфічних ситуаціях і умовах.

Рівень 4 – наступний етап до повністю автономного транспортного засобу включає всі функції, що більше не потребують участі водія. Прикладом може служити безпілотне таксі для коротких поїздок, де кермо та педалі можуть бути відсутніми.

Рівень 5 – на цьому рівні автомобіль здатний пересуватися всюди і за будь-яких умов.

Поринемо в світ технологій: як тільки ви зрозумієте принципи роботи апаратного та програмного забезпечення в автономних транспортних засобах, це допоможе суттєво підвищити безпеку та ефективність вашого бізнесу або автопарку.

Основні процеси для автономії

Хоча термін «автономія» стає дедалі популярнішим, щоб система була дійсно автономною, вона повинна включати певні ключові процеси. Ось чотири з них із прив'язкою до автономних автомобілів:

Сприйняття – це процес, що використовує датчики для збору та обробки даних. У контексті автономних транспортних засобів сприйняття охоплює розпізнавання всього, що відбувається навколо автомобіля, подібно до зору водія.

Локалізація – ці інструменти визначають точне місцеположення автомобіля відносно статичних об’єктів під час руху. Таким чином, безпілотники можуть безпечно пересуватися дорогою та досягати заданої точки призначення.

Планування – передбачає створення маршрутів, за якими транспортний засіб буде рухатися, уникаючи перешкод, таких як дерева, дорожні знаки та інші машини. Цей процес включає розробку траєкторії та профілю швидкості, які контролюють основні функції транспортного засобу, наприклад, швидкість обертів двигуна на холостому ходу чи фази газорозподілу.

Управління - системи управління контролюють і підтримують роботу автономних транспортних засобів за допомогою декількох різних елементів. Наприклад, система керування автоматизованого транспортного засобу спрямовує інші частини автомобіля на ефективну роботу. Сюди входять дросельна заслінка, рульове управління і гальмівна система.

Надаючи можливість автономним транспортним засобам виконувати кожну з цих функцій, компанії пропонують життєздатну альтернативу транспортним засобам з рушієм. Це формує основу для різних пристроїв автономного водіння, які ми розглянемо пізніше.

Обладнання

Для технології автономного керування потрібні такі пристрої

Стереоскопічні камери - камери з двома або більше об'єктивами та окремими датчиками зображення або плівковими кадрами. Стереоскопічні камери імітують людський бінокулярний зір і дозволяють безпілотним транспортним засобам бачити в трьох вимірах.

Радар. Існує два основних типи радарів: частотно-модульовані безперервні хвилі (FMCW) та імпульсні радіолокатори; радіолокаційні імпульси FMCW поширюються безперервно і мають високу роздільну

здатність і діапазон виявлення глибини, що робить їх більш поширеним вибором у сучасних автоматизованих транспортних засобах.

Сонар - сонарна навігація та вимірювання відстані може бути пасивною або активною. Це означає, що ці системи слухають звуки від транспортного засобу або зчитують відлуння, випромінюючи звукові імпульси. Це дозволяє автономним транспортним засобам виявляти інші об'єкти та ефективно спілкуватися з ними.

Тепловізійні камери - тепловізійні камери допомагають автономним транспортним засобам орієнтуватися в оптимальних умовах і дають змогу транспортним засобам рівнів 4 і 5 подолати недоліки інших радіолокаційних технологій. Наприклад, на тепловізійні камери не впливають світло фар, прямі сонячні промені або різкі зміни в освітленні дороги.

Light Detection and Ranging (LiDAR) - одна з новітніх технологій картографування для безпілотних автомобілів. Чотири компоненти, які LiDAR використовує для прийняття рішень щодо автомобіля, - це лазер, приймач, сканер і GPS.

Електронна система курсової стійкості (ESC) - це автоматизована система, яка утримує автомобіль на курсі під час руху. У більшості ситуацій, коли за кермом перебуває людина, ESC допомагає водієві зберігати контроль у складних ситуаціях. Наприклад, ESC може запобігти перекиданню автомобіля.

Камери технічного зору - Камери технічного зору можуть стати технологією наступного покоління для автономних транспортних засобів. Вона має схожі переваги з іншими технологіями, перерахованими тут: Починаючи з 2021 року, Tesla відмовляється від радарів на Model 3 і Model Y. Натомість Tesla Vision використовуватиме більш доступні камери. Ці камери дозволять Tesla оснастити всі свої безпілотні автомобілі технологією, яка уможливлює повністю автономне водіння [8].

Глобальна навігаційна супутникова система (GNSS) - це рішення для точного позиціонування автономних транспортних засобів, технологія GNSS дозволяє автомобілю залишатися в своїй смузі і на безпечній відстані від інших транспортних засобів на дорозі.

Програмне забезпечення

За допомогою правильного програмного забезпечення транспортні засоби можна перетворити на інтелектуальні машини, які можуть підвищити безпеку та ефективність.

Основним алгоритмом, що використовується в безпілотних автомобілях, є згорткова нейронна мережа (CNN). Нижче ми розглянемо, як Tesla, Waymo та Nvidia використовують CNN у своїх автономних автомобілях.

HydraNet від Tesla - HydraNet була розроблена Раві та ін. (2018). По суті, це програмне забезпечення має різні мережі CNN, призначені для конкретних завдань. При класифікації зображень HydraNet використовує функцію спалаху, щоб визначити, які компоненти слід активувати в кожній підзадачі, зменшуючи обчислювальну потужність при збереженні високої точності.

ChauffeurNet від Google Waymo - це нейронна мережа на основі RNN, але CNN все ще є основним компонентом програмного забезпечення; ChauffeurNet - це згорткова мережа ознак, також відома як FeatureNet. Концептуальною основою є імітаційне навчання, яке імітує людську поведінку для виконання певного завдання; ChauffeurNet відрізняється від типового програмного забезпечення для безпілотних автомобілів тим, що сенсорна система є проміжною системою, тобто вона отримує різні рівні вхідних даних від сенсорної системи.

Безпілотні автомобілі Nvidia - Nvidia також використовує CNN в безпілотних автомобілях. Однак ця мінімалістична система використовує меншу мережу для вирішення завдань з меншою кількістю ресурсів за

рахунок самооптимізації; випадки використання для безпілотного автомобіля Nvidia включають навігацію на дорогах без смуг руху та виявлення корисних дорожніх особливостей.

За лаштунками: алгоритми машинного навчання

Перш ніж зануритися у світ науки про дані, важливо зрозуміти, як всі дані поєднуються в автоматизованих машинах. Відповідь лежить у площині мультисенсорного злиття. Мультисенсорний синтез поєднує дані з радарів, LiDAR, камер та ультразвукових датчиків для розуміння умов навколишнього середовища та прийняття рішень.

Після об'єднання цих даних інноватори використовують можливості алгоритмів машинного навчання, таких як згорткові нейронні мережі та імітаційне навчання. Існують й інші алгоритми машинного навчання, які відіграють важливу роль в автоматизованих транспортних засобах. До них відносяться

Навчання з підкріпленням - транспортні засоби навчаються, досліджуючи і взаємодіючи з навколишнім середовищем. Глибоке навчання з підкріпленням (Deep reinforcement learning, DRL) використовується, коли транспортний засіб повинен вибрати певну поведінку в заданому сценарії; трьома змінними в DRL є стан, поведінка і винагорода. Стан відноситься до поточного положення на дорозі, поведінка включає всі можливі дії, які може виконати транспортний засіб, а винагорода відноситься до зворотного зв'язку після виконання дії [9].

Граничні обчислення - це обчислювальне зберігання, управління та аналіз даних. Основна перевага периферійних обчислень полягає в тому, що дані можуть оброблятися в режимі реального часу, щоб інструмент міг негайно реагувати на зміну умов.

Fog Computing - це галузь хмарних обчислень, де дані попередньо обчислюються, зменшуючи таким чином обсяг трафіку, що надсилається на

хмарні сервери. Туманні обчислення використовують зв'язок між транспортними засобами (V2V), а також мережевий зв'язок.

1.3 Дослідження науково-технічних публікацій, присвячених проблемам розробки програмного забезпечення систем управління автономними динамічними об’єктами

Дослідження науково-технічних публікацій, що стосуються розробки програмного забезпечення для систем управління автономними динамічними об'єктами, зосереджуються на кількох ключових аспектах.

Алгоритми управління

Однією з основних проблем є розробка та вдосконалення алгоритмів управління для автономних систем. Це включає методи навігації, стабілізації та адаптації до змін у середовищі. Наприклад, у публікаціях активно досліджуються методи машинного навчання для автономного прийняття рішень у реальному часі, а також системи адаптивного управління, здатні реагувати на непередбачені зміни умов.

Алгоритми управління автономними динамічними об’єктами є ключовим компонентом, який забезпечує їхню здатність діяти незалежно, адаптуючись до змін у середовищі та виконуючи задані завдання.

Базові підходи включають класичні методи управління, такі як пропорційно-інтегрально-диференціальні (PID) регулятори. Ці алгоритми використовуються для стабілізації динамічних систем і забезпечують коригування дій об'єкта на основі відхилень від бажаних параметрів. Хоча ці методи прості й ефективні для лінійних систем, їх недостатньо для складніших нелінійних та мінливих середовищ.

Моделі прогнозного управління (MPC) ці алгоритми базуються на прогнозуванні майбутнього стану системи на основі її математичної моделі. MPC використовує поточну інформацію для розрахунку майбутніх траєкторій та вибору оптимальної стратегії управління. Такий підхід

ефективний для складних динамічних об'єктів, оскільки дозволяє враховувати обмеження та різні фактори в процесі прийняття рішень.

Машинне навчання стає все більш важливим у розробці алгоритмів для автономних систем. Особливо активно використовуються нейронні мережі, глибоке навчання та підкріплювальне навчання. Наприклад, алгоритми підкріплювального навчання дозволяють системі навчатися через проби і помилки, адаптуючи свою поведінку залежно від отриманого зворотного зв’язку. Такі алгоритми підходять для складних середовищ, де неможливо передбачити всі можливі сценарії заздалегідь [10].

Алгоритми дозволяють автономним системам навчатися в режимі реального часу через взаємодію з навколишнім середовищем. Система отримує нагороди або штрафи за свої дії і використовує цю інформацію для коригування поведінки. Це дуже ефективно для задач навігації та прийняття рішень в умовах невизначеності.

Глибокі нейронні мережі використовуються для обробки великої кількості сенсорної інформації, наприклад, зображень або даних з лідара. Вони дозволяють автономним системам аналізувати складні візуальні і просторові дані, приймаючи рішення щодо маршруту, виявлення перешкод та ідентифікації об’єктів у середовищі.

Адаптивні алгоритми дозволяють системам управління змінювати свої параметри відповідно до поточного стану середовища або об’єкта. Наприклад, такі алгоритми можуть змінювати стратегії управління залежно від змін у масі об’єкта, вітрових умов або стану доріг. Це особливо важливо для автономних транспортних засобів та безпілотників, які повинні працювати в непередбачуваних умовах.

У випадку кооперативних систем, таких як рої дронів або автономні транспортні засоби, важливо, щоб кожен об'єкт міг діяти незалежно, але при цьому враховувати інформацію від інших об'єктів. Децентралізовані алгоритми управління забезпечують можливість автономної координації між

об’єктами без централізованого контролю. Вони застосовуються для задач колективного пересування, розподілу ресурсів та уникнення зіткнень.

Алгоритми відповідають за обчислення оптимальних траєкторій руху автономного об'єкта з урахуванням обмежень, таких як перешкоди, обмеження швидкості або енергії. Сучасні підходи включають використання методів пошуку на графах, як-от A*, а також методів математичної оптимізації.

Моделювання і симуляції

Для розробки та тестування програмного забезпечення активно використовуються методи моделювання та симуляції. Це дозволяє дослідникам створювати віртуальні моделі автономних об'єктів (роботів, безпілотників, автомобілів) та навколишнього середовища, щоб перевірити різні сценарії функціонування без необхідності реальних випробувань. Це не лише зменшує витрати, але й мінімізує ризики.

Для моделювання автономних об’єктів необхідно створити математичні моделі, що відображають їхню динаміку. Ці моделі включають параметри, які впливають на поведінку об’єкта, наприклад, масу, швидкість, прискорення, а також зовнішні фактори, як-то сила тяжіння, аеродинамічний опір або тертя. Наприклад, для дронів розробляються моделі, що враховують вплив вітру та зміни висоти.

Ця математична модель дозволяє симулювати рух об'єкта в різних умовах, аналізувати стабільність та адаптивність алгоритмів управління, і таким чином покращувати їх без необхідності ризикувати реальним обладнанням.

Після того як створена модель об'єкта, симуляційне середовище використовують для тестування алгоритмів управління. Наприклад, алгоритми навігації автономного автомобіля можуть бути перевірені у віртуальному місті з реалістичними дорожніми умовами, де симулюються рух інших автомобілів, пішоходів і погодні умови. Це дозволяє тестувати

сценарії, які важко або небезпечно відтворити в реальних умовах, наприклад, раптова поява перешкод на дорозі або зміна покриття під час дощу.

Ключовим елементом є створення реалістичного середовища навколо автономного об’єкта. Для цього використовуються системи моделювання, які можуть імітувати різні типи середовищ – від міських умов для безпілотних автомобілів до відкритого простору для дронів. Важливо, що симуляції можуть включати змінні фактори, такі як зміна освітлення, погоди чи навіть непередбачувані події, як-то несправності в системах або поява непередбачуваних перешкод.

Система управління автономного об'єкта повинна бути здатною адаптуватися до таких змін. Наприклад, віртуальні симуляції можуть перевіряти, як алгоритм реагує на погіршення видимості внаслідок туману або дощу, або як швидко він здатний обчислити нову траєкторію при появі перешкоди.

Симуляція дає змогу не лише перевірити алгоритми, але й провести їхню верифікацію та валідацію. Верифікація полягає в тому, щоб переконатися, що алгоритми працюють відповідно до їхніх технічних специфікацій, а валідація – що вони виконують свої завдання ефективно в контексті реального світу. Наприклад, це дозволяє виявити помилки або недоліки в коді, які можуть призвести до помилок у реальних умовах.

Оскільки системи для автономних об'єктів часто повинні функціонувати в реальному часі, тестування в симуляціях допомагає оцінити, наскільки ефективно алгоритми здатні обробляти великі обсяги даних і приймати рішення в умовах обмеженого часу. Це особливо важливо для роботів або транспортних засобів, де затримка в кілька секунд може призвести до аварії або помилкової поведінки.

Процес моделювання є ітераційним: розробники постійно вдосконалюють алгоритми на основі результатів симуляцій. Наприклад, якщо симуляція показала, що система недостатньо швидко реагує на перешкоди,

розробники можуть змінити алгоритм, зробивши його більш чутливим до зміни середовища, після чого тестування повторюється. Цикл вдосконалення може продовжуватися до того моменту, поки система не покаже стабільні результати в різних симульованих сценаріях.

Хоча симуляції надають величезні можливості, вони мають свої обмеження. Справжні фізичні властивості об’єктів і зовнішніх умов інколи важко точно відобразити в моделі. Тому після успішних симуляцій важливо проводити польові випробування, щоб перевірити, чи відповідає поведінка системи реальним умовам. Проте симуляції суттєво скорочують кількість таких фізичних тестів, що зменшує вартість і ризики.

Таким чином, моделювання та симуляції є невід’ємною частиною процесу розробки, яка дозволяє прискорити створення автономних систем, перевірити їхню надійність і ефективність у складних умовах, і зробити процес тестування безпечнішим та економічнішим.

Проблеми реального часу

Системи управління автономними об'єктами часто мають працювати в умовах реального часу, що вимагає високої продуктивності та надійності програмного забезпечення. Важливою темою є оптимізація обчислювальних ресурсів і зменшення затримок у прийнятті рішень, особливо в контексті роботи з сенсорами та системами зворотного зв'язку.

Основна проблема полягає в тому, що автономний об'єкт (наприклад, дрон або автомобіль) повинен діяти в умовах, коли будь-яка затримка в обробці інформації може призвести до катастрофічних наслідків. Система в реальному часі отримує дані від сенсорів, обробляє їх, аналізує навколишнє середовище та вирішує, як діяти далі. Наприклад, автомобіль, який рухається зі швидкістю 100 км/год, має лише частки секунди на реакцію при виявленні перешкоди. Якщо обробка даних або прийняття рішень затримуються навіть на кілька мілісекунд, це може призвести до аварії [11].

Щоб уникнути цього, розробка таких систем вимагає максимального прискорення всіх етапів – від зчитування даних із сенсорів до виконання команд виконавчими механізмами. У сучасних автономних системах використовуються оптимізовані алгоритми, які дозволяють приймати рішення буквально за мікросекунди, зводячи затримки до мінімуму.

Інша проблема полягає в обмежених обчислювальних ресурсах, доступних на борту автономного об'єкта. Хоча сучасні системи можуть мати потужні процесори та графічні чипи для обробки даних, обсяг інформації, що надходить від різних сенсорів (лідарів, камер, радарів), може бути величезним. Об'єкт має одночасно виконувати кілька завдань: аналізувати простір, розпізнавати об'єкти, планувати траєкторію, стежити за своїм станом і приймати рішення в реальному часі.

Для вирішення цієї проблеми використовуються методи паралельної обробки та розподілу навантаження між різними обчислювальними блоками. Наприклад, окремий блок може бути відповідальним за обробку зображень з камери, в той час як інший займається навігацією. Однак, навіть при цьому важливо, щоб система не перевантажувалася, інакше це також призведе до затримок у прийнятті рішень.

Середовище, в якому працює автономний об’єкт, зазвичай є динамічним і непередбачуваним. Проблеми реального часу часто виникають через необхідність миттєво реагувати на зміни, які неможливо передбачити заздалегідь. Наприклад, дрон, який літає над містом, може раптово стикнутися з сильним поривом вітру або зміненою траєкторією іншого об’єкта.

Алгоритми управління повинні бути досить гнучкими та адаптивними, щоб швидко обробляти такі зміни і коригувати свої дії в реальному часі. Це включає не тільки швидке виявлення проблеми, але й прийняття нового рішення: наприклад, перехід на інший маршрут або коригування швидкості, що вимагає від системи ефективної взаємодії всіх її компонентів.

Системи реального часу повинні працювати безперервно і надійно навіть в екстремальних умовах – від сильних погодних явищ до різних механічних пошкоджень. Наприклад, якщо сенсори автомобіля зазнають впливу дощу або бруду, алгоритми повинні мати можливість або компенсувати це, або швидко перемкнутися на інші джерела даних (наприклад, від камери до лідара). Це означає, що система має здатність негайно діагностувати проблему і адаптувати свою роботу.

У багатьох випадках такі системи розробляються з використанням принципів граничного часу реакції, що означає, що кожна задача має бути завершена протягом певного строго обмеженого часу. Якщо система не встигає виконати завдання в заданий час, це розглядається як критична помилка.

Ще одним важливим аспектом є здатність системи передбачати можливі ситуації та планувати свої дії наперед. Наприклад, автономний автомобіль повинен не тільки реагувати на події в реальному часі, але й передбачати, як можуть змінитися умови в найближчі секунди – наприклад, чи інший автомобіль почне перестроюватися, або ж чи з’явиться пішохід на пішохідному переході.

Такі алгоритми часто використовують методи прогнозного управління або моделювання поведінки інших учасників руху. Це дозволяє їм не просто реагувати на поточні умови, а й діяти проактивно, що зменшує кількість критичних ситуацій, які потребують негайної реакції.

Кібербезпека

Оскільки автономні системи стають все більш складними і інтегрованими в реальні середовища, питання кібербезпеки виходять на перший план. У дослідженнях аналізуються способи захисту програмного забезпечення від зловмисних атак, що можуть вплинути на функціонування автономних систем. Це включає як питання шифрування, так і виявлення та запобігання атакам.

Автономні системи, такі як дрони, автомобілі або роботи, підключені до різних мереж для отримання даних від сенсорів, GPS, а також обміну інформацією з іншими об'єктами або хмарними сервісами. Це створює ризик зовнішніх атак через злом мережі або несанкціонований доступ до даних. Наприклад, хакер може перехопити або змінити команди, які передаються до автономного об'єкта, змусивши його виконати небажані або навіть небезпечні дії.

Щоб запобігти цьому, впроваджуються складні механізми шифрування і аутентифікації. Дані, що передаються, мають бути захищені таким чином, щоб їх не можна було перехопити або змінити під час передачі. Важливим завданням є забезпечення цілісності інформації, яка передається від сенсорів або керуючих систем до автономного об’єкта. Без цього об'єкт може приймати неправильні рішення, що призведе до аварійних ситуацій.

Програмне забезпечення, яке керує автономними об'єктами, може містити уразливості, які можуть бути використані зловмисниками для отримання доступу до системи. Це можуть бути як помилки в коді, так і недоліки в архітектурі системи. Наприклад, уразливості в системі оновлень можуть дозволити хакерам встановлювати шкідливе ПЗ, що змінює поведінку об'єкта або викрадає дані.

Особливо небезпечними є атаки, які спрямовані на управління системами реального часу. В такому випадку зловмисник може спровокувати помилкові рішення в момент, коли система повинна миттєво реагувати на зовнішні фактори. У випадку з автономними автомобілями, наприклад, це може призвести до зміни маршруту або раптової зупинки на швидкості.

Автономні системи покладаються на широкий спектр сенсорів для отримання інформації про навколишнє середовище, таких як радари, лідари, камери та ультразвукові датчики. Ці сенсори можуть стати об'єктами атак, якщо зловмисники зможуть перехопити або підробити дані, які вони

передають. Наприклад, атаки на систему GPS можуть змінити координати, що змусить автономний об'єкт рухатися за неправильним маршрутом.

Ще однією загрозою є підміна або спотворення сенсорних даних, так звані "атаки обману". У таких випадках система може бути обманута фальшивими сигналами, змушуючи об'єкт виконувати небезпечні дії. Для протидії таким загрозам розробляються алгоритми, які здатні виявляти аномальні дані і фільтрувати неправдиву інформацію.

Небезпеку становлять не тільки зовнішні зломи, але й внутрішні загрози. Це може бути випадкове чи навмисне введення шкідливого коду співробітниками, які мають доступ до системи. У автономних системах, які залежать від мережевих з'єднань і хмарних сервісів, можливі випадки, коли хтось із команди або партнерів отримує доступ до критичних даних або можливостей управління.

Для вирішення цієї проблеми в системах впроваджуються строгі політики доступу та контроль над рівнями прав. Це гарантує, що навіть ті, хто має доступ до певних частин системи, не зможуть впливати на її функціонування без належної перевірки та підтвердження.

Оскільки повністю усунути ризик кіберзагроз неможливо, критично важливою є здатність автономної системи відновлюватися після атак або збоїв. Системи повинні мати механізми, які дозволяють швидко повернутися до нормального функціонування навіть після кібернападу. Це можуть бути резервні копії, автоматичні відновлення, а також алгоритми, здатні переконатися, що об’єкт працює належним чином після усунення загрози. Відновлюваність є ключовою рисою кібербезпеки в автономних системах, особливо якщо мова йде про системи, що працюють у критичних умовах, таких як автономні транспортні засоби, медичні роботи або промислові дрони [12].

Інтерфейси та інтеграція

Важливим аспектом є інтеграція програмного забезпечення з апаратними компонентами. Публікації зосереджуються на розробці інтерфейсів, що дозволяють ефективну взаємодію програмного забезпечення з сенсорами, виконавчими механізмами та іншими системами. Також досліджуються методи стандартизації та забезпечення сумісності різних компонентів.

Таким чином, основні напрямки досліджень в розробці ПЗ для автономних систем включають розвиток алгоритмів управління, симуляційне моделювання, роботу в умовах реального часу, кібербезпеку та інтеграцію з апаратними рішеннями. Ці теми формують основу для вдосконалення сучасних автономних технологій.

1.4 Виділення не вирішених раніше частин проблеми та уточнення науково-технічного завдання на дану роботу

Для виділення нерозв'язаних частин проблеми управління автономними динамічними об'єктами (АДО) необхідно глибше зануритися в специфіку завдань, з якими стикаються дослідники.

Однією з ключових проблем є невизначеність динаміки самого об'єкта та середовища його функціонування. Більшість існуючих підходів, орієнтованих на моделювання та контроль руху АДО, ґрунтуються на детермінованих моделях. Однак реальні умови функціонування часто супроводжуються невизначеностями, пов'язаними з мінливими характеристиками середовища або недосконалістю сенсорних систем. Це ускладнює забезпечення стабільного управління, особливо в ситуаціях, коли дані містять шум або є неповними.

Інша невирішена частина проблеми стосується синхронізації та інтеграції рішень на різних рівнях управління. В сучасних підходах часто використовуються розділені системи управління для різних аспектів функціонування АДО — планування шляху, контроль орієнтації, стабілізація

тощо. Відсутність інтегрованої платформи, яка могла б узгоджено працювати на всіх рівнях і забезпечувати реальну адаптивність у режимі реального часу, залишається серйозним викликом.

Також важливою є проблема навчання систем управління. Використання методів машинного навчання, зокрема глибокого навчання, дозволяє автономним системам пристосовуватись до нових умов. Однак проблема забезпечення надійності та стійкості таких рішень залишається невирішеною, особливо коли система працює в режимі, де невідомі умови змінюються динамічно і не передбачені в рамках навчання.

Важливо також враховувати питання енергоефективності та оптимізації ресурсів. Автономні динамічні об'єкти, як правило, мають обмежені ресурси живлення, що вимагає розробки ефективних алгоритмів управління, які б знижували споживання енергії без втрати точності та надійності управління.

Управління автономними динамічними об'єктами має кілька важливих невирішених аспектів, які потребують подальших досліджень. Однією з таких проблем є інтеграція різнорідних джерел даних для прийняття управлінських рішень в умовах невизначеності. Системи автономних об'єктів залежать від численних датчиків, але їх робота часто супроводжується неточностями або збоєм у передачі інформації. Це ускладнює точну оцінку стану середовища та призводить до ризику неправильних рішень, особливо в реальних умовах, де об'єкти повинні діяти оперативно [13].

Крім того, важливим аспектом є потреба в адаптивності систем управління в контексті змінних середовищ. Автономні динамічні об'єкти часто працюють у непередбачуваних умовах, наприклад, в міських середовищах з високою щільністю транспорту або в природних умовах з обмеженим доступом до точної інформації про місцевість. У таких ситуаціях необхідно розробляти моделі, здатні ефективно адаптувати стратегії управління в реальному часі.

Ще одним викликом є створення стійких алгоритмів для оптимізації руху та керування автономними системами з урахуванням складних динамічних обмежень. Наприклад, керування безпілотниками чи роботами в тісно пов'язаних умовах часто потребує не тільки точності траєкторій, але й здатності до маневрування в обмежених просторах з високою швидкістю та мінімальною реакцією на зовнішні зміни.

Також у контексті багатозадачних середовищ виникає проблема координації кількох автономних об'єктів, які взаємодіють між собою або з іншими системами. Існує потреба у розробці методів, які забезпечують безперебійне та безпечне функціонування декількох об'єктів, що працюють одночасно в одній зоні, без виникнення конфліктів і аварійних ситуацій. Це включає не лише проблеми навігації, але й питання комунікації між об'єктами.

Таким чином, основне науково-технічне завдання полягає в створенні стійких, адаптивних систем управління АДО, здатних діяти в умовах невизначеності та забезпечувати оптимальну інтеграцію між різними рівнями управління з урахуванням обмежених ресурсів, проблематика управління автономними динамічними об'єктами вимагає подальшого вивчення для забезпечення надійності, адаптивності та інтеграції різних підсистем управління в умовах динамічних і непередбачуваних середовищ [14].

1.5 Висновки по розділу

Сучасні алгоритми управління для автономних систем є складною комбінацією класичних підходів та новітніх методів, таких як машинне навчання і адаптивні системи. Основна мета полягає в забезпеченні автономності, надійності та гнучкості в складних і непередбачуваних умовах.

Основні проблеми реального часу в системах управління автономними об'єктами стосуються необхідності швидкої і точної обробки даних, обмежених обчислювальних ресурсів і потреби в адаптивності до

непередбачуваних змін. Завдання полягає в тому, щоб мінімізувати затримки, зберігаючи при цьому надійність системи, навіть у найскладніших умовах.

Кібербезпека в системах управління автономними об'єктами має вирішальне значення для забезпечення безпечного і надійного функціонування таких систем. Вона вимагає не тільки захисту від зовнішніх загроз, але й постійного моніторингу, адаптації до нових видів атак і забезпечення стійкості системи у випадку інцидентів.

Щодо управління автономними динамічними об'єктами полягає в тому, що основні виклики зосереджені на адаптивності до умов невизначеності, інтеграції різнорідних систем і ефективному використанні ресурсів. Рішення повинні ґрунтуватися на більш точному врахуванні реальних умов експлуатації, де автономні об'єкти взаємодіють із мінливими середовищами та різними системами. Це вимагає удосконалення алгоритмів для обробки нестабільних даних, створення більш реалістичних моделей поведінки об'єктів та забезпечення надійного функціонування в багатозадачних середовищах. Усе це потребує міждисциплінарного підходу для досягнення стабільної роботи таких систем у реальному світі.

2 Моделювання структури та функціональності програмного забезпечення системи управління автономними динамічними об'єктами

2.1 Виділення загальних особливостей програмного забезпечення, що розглядається

Загальні відмінності В UML можна моделювати як класи, так і об'єкти (Рисунок 2.1). Класи - це абстракції, а об'єкти - їх конкретні втілення. Для графічного представлення об'єктів в UML використовуються ті ж символи, що і для класів, але імена об'єктів підкреслюються. При моделюванні

об'єктно-орієнтованих систем реальність може бути розділена різними способами. На рисунку 2.2 показано клас Customer з трьома об'єктами: «Jan», який явно вказаний як об'єкт класу “Customer”; безіменний об'єкт класу “Customer”; та “Elyse”, який не показаний явно, але вказаний як об'єкт класу “Customer”.

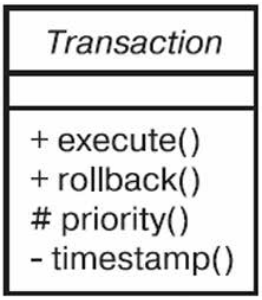


Рис. 2.1 Доповнення

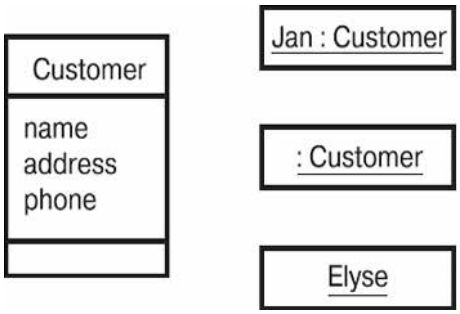


Рис. 2.2 Класи й об’єкти

Майже всі будівельні блоки UML характеризуються подібною дихотомією клас/об'єкт. Наприклад, є варіанти використання та екземпляри варіантів використання, компоненти та екземпляри компонентів, вузли та екземпляри вузлів тощо.

Другий - це розділення інтерфейсу та реалізації. Інтерфейс визначає угоду, а реалізація визначає її конкретну реалізацію і зобов'язується слідувати всій семантиці інтерфейсу; в UML можна моделювати як інтерфейс, так і його реалізацію (рис. 2.3). Тут є компонент spellingwizard.dll, який реалізує два інтерфейси, IUnknown та ISpellling. Йому також потрібен інтерфейс IDictionary, який має бути наданий окремим компонентом. Таким чином, для забезпечення мобільності коду використовується та сама

дуальність інтерфейсу/реалізації. Зокрема, ВВ реалізуються за допомогою координації, а операції - за допомогою методів [15].

Рис. 2.3 Інтерфейси та реалізація

По-третє, їх можна розділити на типи та ролі. Типи визначають клас сутності (об'єкт, атрибут, параметр). Ролі визначають значення сутності в контексті (клас, компонент, співпраця). Сутність, яка є частиною структури іншої сутності (атрибута), має обидві властивості. Вона успадковує певний вміст від свого типу (класу), а інший вміст - від своєї ролі в цьому контексті (рис. 2.4).

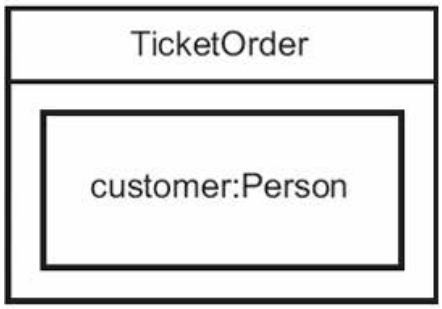


Рис. 2.4 Частина з роллю і типом

Механізми розширюваності. Як стандартна мова для розробки програмних «креслень», UML не завжди може охопити всі можливі нюанси моделі у всіх сферах застосування. Тому UML є відкритою мовою і дозволяє використовувати механізми контрольованої розширюваності, такі як:

□ стереотипи;

□ присвоєні значення;

□ обмеження.

Стереотипи розширюють словниковий запас UML, дозволяючи успадковувати існуючі будівельні блоки і створювати нові типи будівельних

блоків для вирішення конкретних завдань. Наприклад, при роботі з C++ або Java вам потрібно моделювати винятки, які можна розглядати як звичайні класи. Винятки потрібно вміти генерувати та перехоплювати. Винятки моделюються шляхом позначення їх відповідними стереотипами і уподібнення до базових будівельних блоків (класів переповнення) (рис. 2.5).

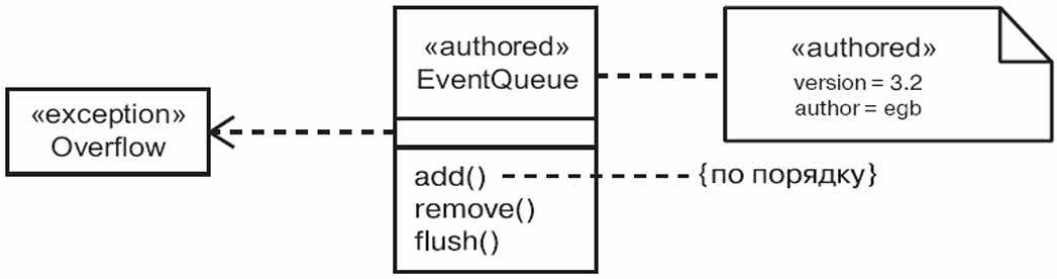


Рис. 2.5 Механізми розширення

Марковані значення розширюють властивості стереотипів UML, дозволяючи включати в специфікацію стереотипу нову інформацію (специфічні параметри деяких важливих абстракцій можна простежити в римі). Такі параметри не є примітивами UML, але можуть бути додані до будь-якого будівельного блоку (класу) шляхом додавання нових значень присвоєння; клас EventQueue розширено шляхом явного зазначення його версії та автора (рис. 2.5).

Обмеження розширюють семантику будівельного блоку UML, додаючи нові правила або змінюючи існуючі правила; клас EventQueue можна розширити, додавши обмеження, яке явно задає таке правило до операції вставки, так що всі події впорядковуються одна за одною і є обмеженими для вставки (рис. 2.5).

Ці три механізми розширення мови можна комбінувати для модифікації UML відповідно до вимог проекту або для адаптації UML до нових програмних технологій (наприклад, розподілених мов програмування). Можна додавати нові будівельні блоки, змінювати властивості існуючих будівельних блоків і модифікувати семантику в звичайних межах.

Початкова стадія освоєння нової теми. Єдиний ефективний метод вивчення нової мови програмування полягає в активному написанні програм на ній, як зазначають Брайан Керніган і Деніс Річі, автори мови С. Аналогічно, для засвоєння UML слід створювати моделі з її використанням. Зазвичай, першою програмою, яку пишуть багато розробників, вивчаючи нову мову програмування, є надзвичайно проста. Вона виконує мінімальну дію, наприклад, друкує рядок на кшталт "Привіт, UML!". Цей банальний додаток є важливим, оскільки демонструє весь процес від компіляції до запуску програми, та забезпечує платформу для подальшого розвитку. Навіть у простоті цього додатка заховані різноманітні механізми, що підтримують його функціонування [16].

```
import java.awt.Graphics;
class SalutUML extends Java.applet.Applet
  {
    public void paint (Graphics g)
        {g.drawstring("Salut, UML!", 10, 10);}
      }
```

Перший рядок коду призначений для доступу до класу Graphics у наступному коді. Префікс java.awt. вказує на пакет Java, в якому знаходиться клас Graphics. Другий рядок коду вводить новий клас SalutUML, який є нащадком класу Applet у пакеті java.applet. Застосування цієї операції викликає іншу операцію drawstring, яка виводить рядок «Salut, UML!» у вказаних координатах. drawstring - це операція над об'єктом g класу Graphics.

Для моделювання цього додатку на мові UML використовується діаграма класів (рисунок 2.6): Клас SalutUML графічно представлено у вигляді прямокутної іконки. Операції розфарбовування показано тут без формальних параметрів, а реалізацію наведено у супровідних примітках. Діаграма класів передає основну ідею реалізації «Salut, UML!», але в той же час залишає багато чого поза увагою. Як видно з коду, в реалізації беруть

участь два класи: Applet і Graphics. Клас Applet використовується як основний клас SalutUML, тоді як клас Graphics використовується для підпису та реалізації операцій малювання [17].

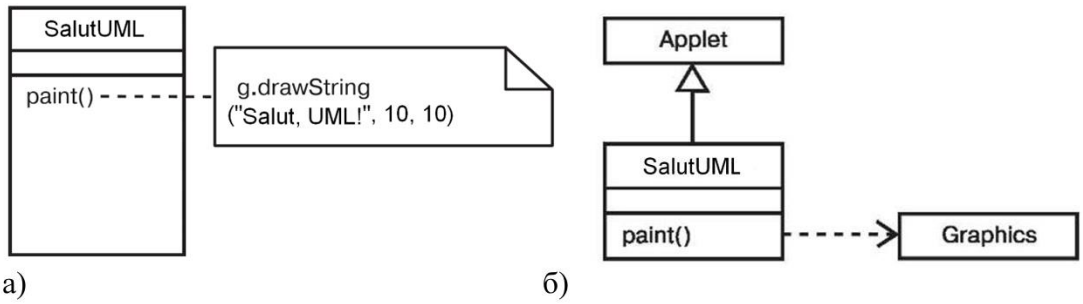


Рис. 2.6 Ключові абстракції класу SalutUML (а) та діаграма класів (б)

Класи Applet та Graphics зображено у вигляді прямокутних символів. Їхні операції не показані; порожня лінія зі стрілкою від SalutUML до Applet - це узагальнення, яке вказує на те, що SalutUML є нащадком Applet; пунктирна лінія від SalutUML до Graphics - це залежність і Graphics. І це далеко не вся структура SalutUML: Як видно з бібліотеки Java, що містить Applet і Graphics, ці два класи є частиною більшої ієрархії. якщо ми подивимося тільки на класи, які розширює і реалізує Applet, можна побудувати окрему діаграму класів (рис. 2.7).

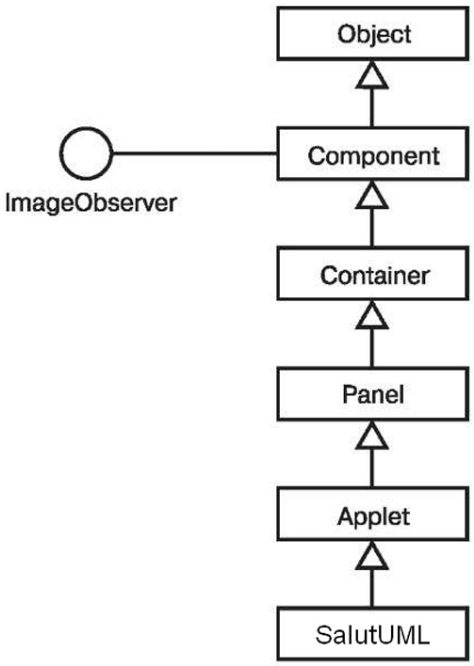


Рис. 2.7 Ієрархія успадкування

Хоча UML не є візуальною мовою програмування в традиційному розумінні, вона забезпечує глибоку інтеграцію з такими мовами програмування, як Java (див. рис. 2.6). UML надає можливість перетворювати моделі в код і навпаки — з коду в модель. Для деяких аспектів, таких як математичні вирази, доречніше використовувати текстові мови програмування, тоді як ієрархію класів доцільно представляти графічно за допомогою UML.

Зв’язок між ImageObserver (Оглядач Зображень) та Component помітно відрізняється від типових зв’язків узагальнення і ця особливість відображена на діаграмі класів (див. рис. 2.7). У контексті бібліотеки Java ImageObserver реалізований у форматі інтерфейсу. Це означає, що він не містить власної реалізації, а натомість вимагає, щоб інші класи реалізовували його функціонал.

Зображення того, що клас Component реалізує інтерфейс ImageObserver, можна здійснити за допомогою суцільної лінії, яка з’єднує прямокутник (Component) та овал, що позначає інтерфейс (ImageObserver). Клас SalutUML безпосередньо взаємодіє лише з двома класами — Applet та Graphics — які є частиною багатої бібліотеки класів Java (див. рис. 2.7).

Виділення загальних особливостей програмного забезпечення, що розглядається

Пакетування. Щоб ефективно керувати великою колекцією елементів, Java організовує свої інтерфейси та класи у структуровану систему пакетів. Основним пакетом середовища Java є java, який містить кілька підпакетів, кожен з яких включає додаткові пакети, інтерфейси та класи. Наприклад, клас Object знаходиться у пакеті lang, тож його повний шлях описується як java.lang.Object.

Схожим чином, такі класи, як Panel, Container і Component, належать до пакета awt, тоді як клас Applet розташований у пакеті applet. Інтерфейс

ImageObserver входить до пакету image, що також є частиною awt, таким чином його повний шлях виглядає як java.awt.image.ImageObserver.

Ця система пакетування може бути візуалізована на діаграмі класів (див. рис. 2.8). У специфікації UML пакети зображаються у формі папок зі спеціальними закладками. Пакети можуть бути вкладеними один в один, а пунктирні лінії зі стрілками вказують на залежності між ними. Наприклад, клас SalutUML має залежності від пакета Java.applet, а останній своєю чергою залежить від Java.awt.

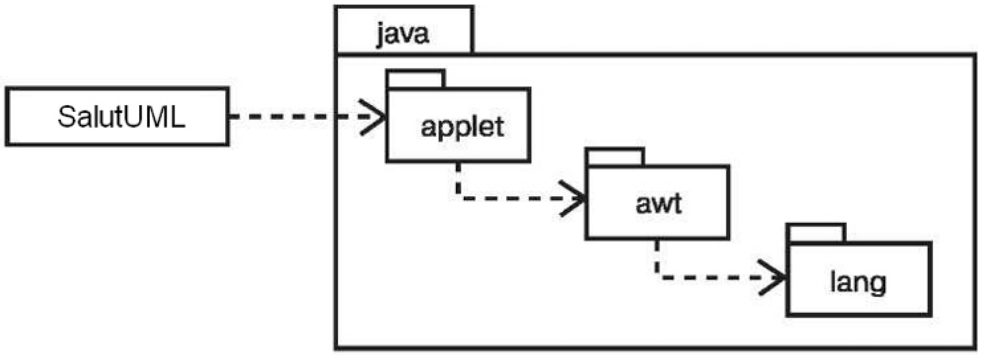


Рис. 2.8 Пакетування SalutUML

Механізми взаємодії. Одним із найбільш складних завдань, що стоять перед розробниками під час освоєння настільки обширної платформи, як бібліотека Java, є розуміння способу взаємодії її компонентів. Наприклад, виникає питання: яким чином викликається метод paint класу SalutUML? Які саме методи потрібно імплементувати або змінити, щоб модифікувати поведінку цього аплета, скажімо, для відображення тексту іншого кольору? Для вирішення таких питань необхідно мати концептуальну модель, яка демонструє взаємодію класів у динаміці. Згідно з архітектурою Java, метод paint від класу SalutUML є успадкованим від класу Component. Проте залишається питання, яким чином цей метод активується. Відповідь полягає у тому, що paint виконується в межах потоку, що керує виконанням аплета (див. рисунок 2.9) [18].

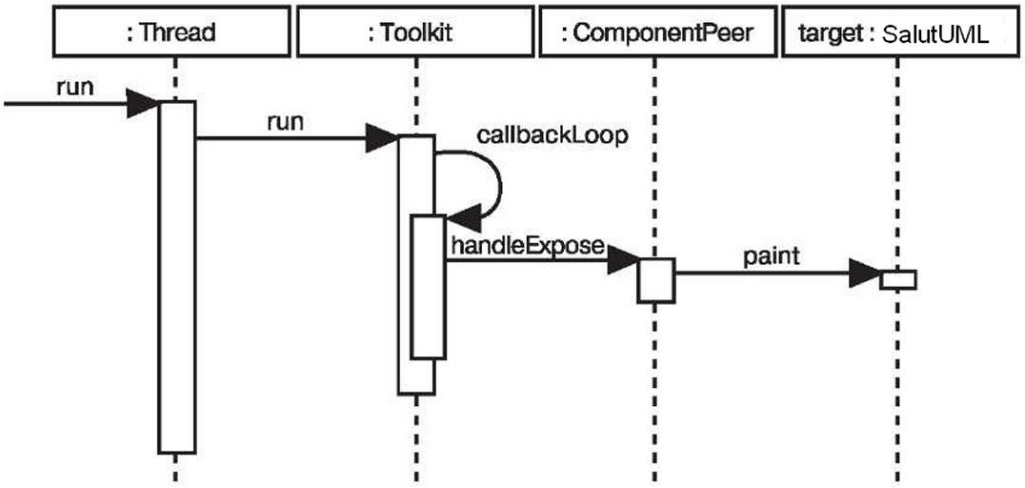


Рис. 2.9 Зображення механізму

Послідовність подій в даному випадку відображає модель, представлена через діаграму послідовності (рис. 2.9). На цій діаграмі процес починається з ініціації об'єкта класу Thread, який викликає метод run об'єкта класу Toolkit. Потім об'єкт Toolkit здійснює виклик одного зі своїх методів, а саме callbackloop, що в подальшому ініціює метод handleexpose класу Componentpeer. Клас Componentpeer передбачає, що його ціллю є екземпляр класу Component, проте в цьому випадку об'єкт належить до підкласу Component - SalutUML. Відтак, метод paint викликається поліморфно, відповідно до класу SalutUML [19].

Артефакти. Програма "Salut, UML!" реалізована у формі аплета, що унеможливлює її автономний запуск і передбачає функціонування виключно в контексті веб-сторінки. Початок роботи аплета відбувається при завантаженні веб-сторінки за допомогою браузерного механізму, який активує виконання об'єкта Thread цього аплета. Водночас, клас SalutUML не є безпосереднім компонентом веб-сторінки — представлена лише його бінарна форма, створена компілятором Java з вихідного коду, що слугує виконуваним артефактом. Такий підхід дозволяє сформулювати нове уявлення про систему: якщо попередні діаграми пропонували логічну

структуру аплета, то на цьому етапі ми спостерігаємо фізичну репрезентацію його артефактів.

Цей зовнішній вигляд можна змоделювати за допомогою діаграми артефактів (Рис. 2.10). Ця діаграма взаємодії показує взаємодію різних об'єктів, включаючи екземпляр класу SalutUML. Інші об'єкти, показані тут, є частиною середовища Java і тому здебільшого знаходяться у фоновому режимі програми, що створюється. Це співпраця між екземплярами багаторазових класів. Кожен стовпчик показує роль у співпраці, тобто частину об'єкта, яка може бути виконана іншим об'єктом за одне виконання В UML ролі представлені так само, як і класи, з тією лише різницею, що ролі визначаються їхнім типом та іменем (назвою об'єкта). Дві ролі в центрі цієї діаграми є анонімними. Це тому, що їхніх типів достатньо, щоб визначити їх як взаємодіючі (хоча двокрапка і підкреслення вказують на те, що вони є ролями). Перший об'єкт Thread називається кореневим, тоді як роль SalutUML називається цільовою (так само, як і ім'я об'єкта) і відома через об'єкт Componentpeer.

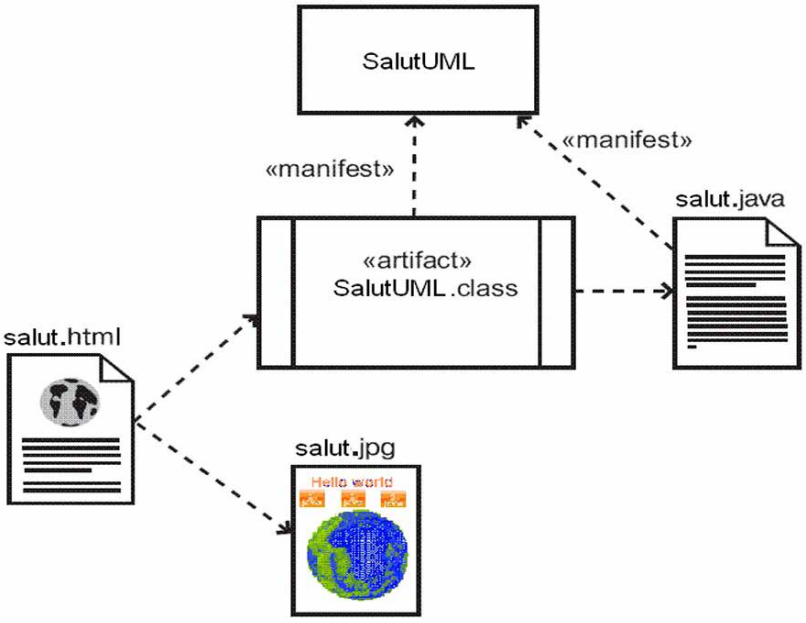


Рис. 2.10 Артефакти

Логічні класи SalutUML представлені прямокутником вище. Всі інші піктограми на діаграмі представляють артефакти UML у вигляді реалізацій

системи. Артефакт - це фізична сутність, наприклад, файл: артефакт з назвою salut.java представляє вихідний код логічних класів SalutUML. Цим файлом можна маніпулювати як у середовищі розробки, так і за допомогою інструментів керування конфігурацією. Вихідний код може бути перетворений у двійкову реалізацію salut.class за допомогою компілятора Java і придатний для виконання у середовищі віртуальної машини Java. Як вихідний код, так і бінарна реалізація фізично реалізують булевий клас. На це вказує пунктирна стрілка, позначена ключовим словом manifest.

Піктограма артефакту - це прямокутник з ключовим словом artefact над його назвою; бінарна реалізація SalutUML.class є варіацією цієї базової піктограми, але з товстою лінією, яка вказує на те, що це виконуваний артефакт (піктограму артефакту salut.java було замінено на спеціальну піктограму, яка представляє текстовий файл). Як видно з діаграми, веб-сторінка містить ще один артефакт, salut.jpg, також представлений спеціальною іконкою. Хоча цього разу це графічний файл, останні три артефакти представлені визначеними користувачем графічними символами, тому їхні назви винесені за межі іконок. Залежності між артефактами позначені пунктирними стрілками.

2.2 Розробка алгоритмічних основ програмного забезпечення, що розглядається

Розробка програмного забезпечення як великої, монолітної, «фіксованої» одиниці ускладнює майбутні зміни. Навіть якщо існуюча система має більшу частину необхідної функціональності, вона, ймовірно, містить багато непотрібних частин, які важко або неможливо видалити. Тому системи повинні бути побудовані з окремих компонентів, які гнучко взаємодіють один з одним і можуть бути відокремлені та додані в міру зміни вимог без шкоди для цілого.

Інтерфейс - набір операцій, які визначають сервіси, представлені класом або компонентом.

Компонент - змінна частина системи, яка відповідає набору інтерфейсів і гарантує їх реалізацію.

Порт - специфічне для інкапсульованого компонента «вікно», яке приймає повідомлення до компонента та від нього відповідно до визначених інтерфейсів.

Внутрішня структура - реалізація компонента, представлена набором частин, з'єднаних між собою певним чином.

Частина - специфікація ролі, яка є частиною реалізації компонента. Екземпляр компонента містить екземпляр, що відповідає частині.

З'єднувач - канал зв'язку між двома частинами або портами в контексті компонента.

Компонент та інтерфейс. Взаємозв'язок між компонентами (що надають послуги) та інтерфейсами є важливим. Всі інструменти та інші програмні системи (COM+, CORBA, Enterprise Java Beans) побудовані на компонентах, які використовують інтерфейси компонентів один з одним.

Проектування компонентно-орієнтованої системи означає декомпозицію системи та визначення інтерфейсів, які представляють основні зв'язки. Наступним кроком є визначення компонентів, які реалізують інтерфейси, та інших компонентів, які отримують доступ до сервісів через ці інтерфейси. Цей механізм дозволяє розгорнути систему, в якій сервіси певною мірою не залежать від місця розташування.

Інтерфейс, реалізований компонентом, називається інтерфейсом, що надає (компонент надає інтерфейс як послугу іншим компонентам). Компонент може оголошувати набір інтерфейсів, що надаються. Інтерфейс, який використовується компонентом, називається інтерфейсом запиту; цей інтерфейс є інтерфейсом, який цей компонент використовує для запиту послуг від інших компонентів. Компонент може підтримувати багато

інтерфейсів запитів. Деякі компоненти мають як інтерфейс надання, так і інтерфейс запиту одночасно [20].

Компоненти зображуються у вигляді прямокутника з двозубою піктограмою у правому верхньому куті (Рисунок 2.11). У центрі прямокутника відображається назва компонента. Компоненти можуть мати атрибути та операції, які часто не відображаються на діаграмах. Компоненти можуть мати мережу внутрішніх структур.

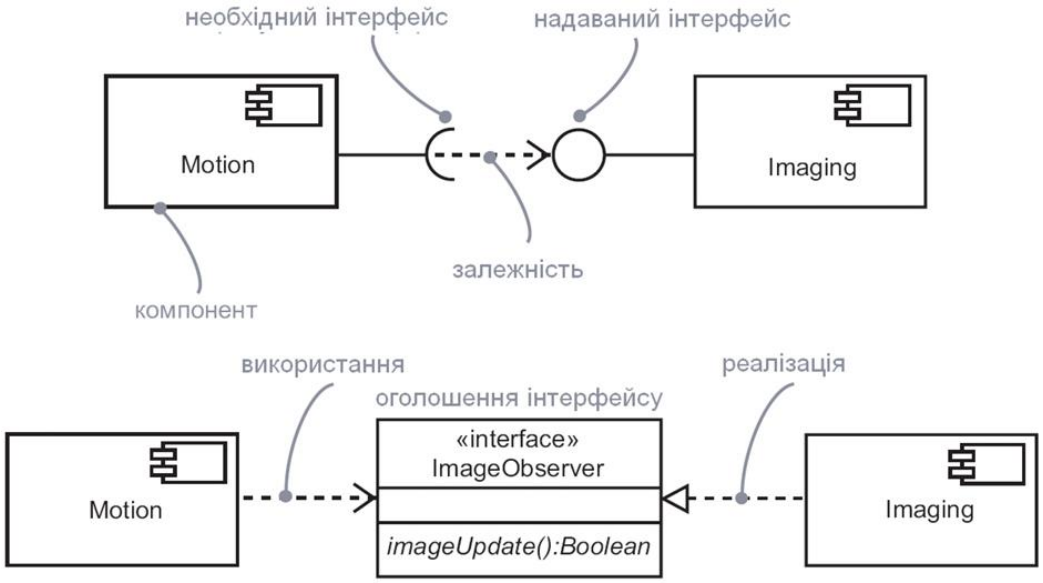


Рис. 2.11 Компоненти та інтерфейси

Зв'язок між компонентами та їх інтерфейсами можна виразити одним з двох способів. У символічній формі наданий інтерфейс виглядає як коло, з'єднане з компонентом лінією («льодяник на паличці» для класу). Необхідний інтерфейс - це півколо, з'єднане з компонентом («сокет»). Назва інтерфейсу з'являється поруч із символом. Компоненти, що реалізують інтерфейс, підключаються до інтерфейсу за допомогою повного з'єднання програми. Компоненти, що реалізують інтерфейс, підключаються до інтерфейсу за допомогою повного з'єднання програми. Компоненти, які можуть отримати доступ до послуг інших компонентів через інтерфейс, зв'язуються з інтерфейсом за допомогою залежності. Деякі інтерфейси надаються одним компонентом і є необхідними для іншого компонента. Оскільки цей інтерфейс знаходиться між двома компонентами, їх пряма

залежність один від одного усувається. Компонент, який використовує цей інтерфейс, працюватиме коректно незалежно від того, який компонент його реалізує. Для використання компонента в цьому контексті всі інтерфейси, необхідні для цього компонента, повинні бути реалізовані так само, як вони надаються іншим компонентом [21].

Сумісність. Основне призначення засобів розробки систем на основі компонентів полягає в тому, щоб дозволити збирати системи з бінарно сумісних артефактів. Такі системи можуть бути спроектовані за допомогою компонентів і реалізовані як артефакти. Систему також можна розширювати, додаючи нові компоненти або модифікуючи старі, без необхідності перебудовувати всю систему. Інтерфейси є важливим способом забезпечення таких можливостей. У працюючій системі можна використовувати будь-який артефакт, який реалізує узгоджені компоненти і надає необхідні інтерфейси. Розширення системи може здійснюватися шляхом створення компонентів, які надають нові послуги через інші інтерфейси, які потім можуть бути виявлені і використані іншими компонентами. Ця семантика пояснює мету визначення компонентів в UML. Компоненти відповідають наборам інтерфейсів і надають реалізації, які можуть бути замінені як у логічному дизайні, так і у фізичній реалізації, що базується на ньому.

Взаємозамінний компонент - це компонент, який може бути замінений в процесі проектування іншим компонентом, що відповідає тому ж інтерфейсу. У сприятливій системі механізм розміщення взаємозамінних артефактів або допускається об'єктною моделлю (наприклад, COM+ або Enterprise Java Beans), яка є прозорою для користувача компонента і вимагає декількох проміжних перетворень, або цей механізм виконується інструментами, які його автоматизують.

Компоненти є частиною системи і рідко використовуються самостійно. Частіше вони поєднуються з іншими компонентами. Це означає, що вони є частиною архітектурного та технічного контексту, в якому

використовуються. Компоненти є логічно та фізично узгодженими і утворюють складову або поведінкову частину більшої системи. Вони придатні для багаторазового використання. Вони є основними будівельними блоками для проектування та побудови систем. Компоненти відповідають набору інтерфейсів і забезпечують їх реалізацію.

Склад компонентів. Компоненти можуть бути організовані у пакети так само, як і класи. Компоненти можуть бути організовані шляхом встановлення залежностей, відношень узагальнення, пов'язаних відношень (включаючи агрегування) та відношень реалізації між компонентами. Деякі компоненти можуть бути побудовані з інших компонентів.

Порти є своєрідним «вікном» для інкапсульованих компонентів. Інтерфейси корисні для опису загальної поведінки компонента, але не мають «індивідуальності». Порти дають більше контролю над програмою.

Усі взаємодії з такими компонентами на входах і виходах відбуваються через точки з'єднання. Зовнішньо виражена поведінка компонента - це сума його портів. Кожен порт є унікальним. Компонент може взаємодіяти з іншим компонентом через певний порт. У цьому випадку зв'язок повністю визначається інтерфейсом, що підтримується портом, навіть якщо компонент підтримує інші інтерфейси. У додатку внутрішні частини компонента можуть взаємодіяти один з одним через певні зовнішні порти, таким чином гарантуючи, що кожен компонент не залежить від вимог інших компонентів. Порти дозволяють розділити інтерфейси компонентів на окремі пакети і використовувати їх окремо. Інкапсуляція та незалежність, яку забезпечують порти, підвищує взаємозамінність компонентів [22].

Схематично порт зображується невеликим прямокутником збоку компонента і є отвором в кінці інкапсуляції компонента. До символу порту можуть бути підключені як інтерфейси, що надають, так і інтерфейси, що запитують. Інтерфейс, що надає, представляє послуги, які можуть бути запитані ззовні через цей порт, в той час як інтерфейс, що запитує,

представляє послуги, які порт повинен отримати від інших компонентів. Кожен порт має назву і може бути ідентифікований за компонентом та назвою. Останнє може використовуватися компонентами всередині для ідентифікації портів, які надсилають та отримують повідомлення. Ім'я компонента та ім'я порту ідентифікують порти, що використовуються іншими компонентами.

Порт - це частина компонента. Екземпляр порту може бути створений або знищений разом з екземпляром компонента, до якого він належить. Також може існувати декілька точок прив'язки. Це означає, що в одному екземплярі компонента може бути декілька екземплярів точки прив'язки. Кожен порт компонента має масив відповідних екземплярів. Всі екземпляри портів у масиві задовольняють один і той самий інтерфейс і приймають один і той самий тип запиту, але вони можуть перебувати у різних станах і мати різні значення даних. Наприклад, кожен екземпляр у масиві може мати свій рівень пріоритету (екземпляр порту з найвищим рівнем пріоритету обслуговується першим).

На рисунку 2.12 показано модель компонента продажу квитків з портами. Кожен порт має ім'я та необов'язковий тип, що вказує на призначення порту. Компонент має порти для продажу квитків, оголошень та транзакцій з кредитними картками. Існує два порти для продажу квитків, один для звичайних клієнтів і один для привілейованих клієнтів. Обидва надають однаковий інтерфейс продажу квитків. Порт для обслуговування кредитних карток має необхідний інтерфейс. Порт «Оголошення» має як стандартний, так і необхідний інтерфейс. Інтерфейс Load Attractions дозволяє театру імпортувати афіші та іншу інформацію про виставу до бази даних, що використовується для продажу квитків. Інтерфейс компонента «Бронювання» дозволяє продавцям квитків запитувати та купувати квитки у театру.

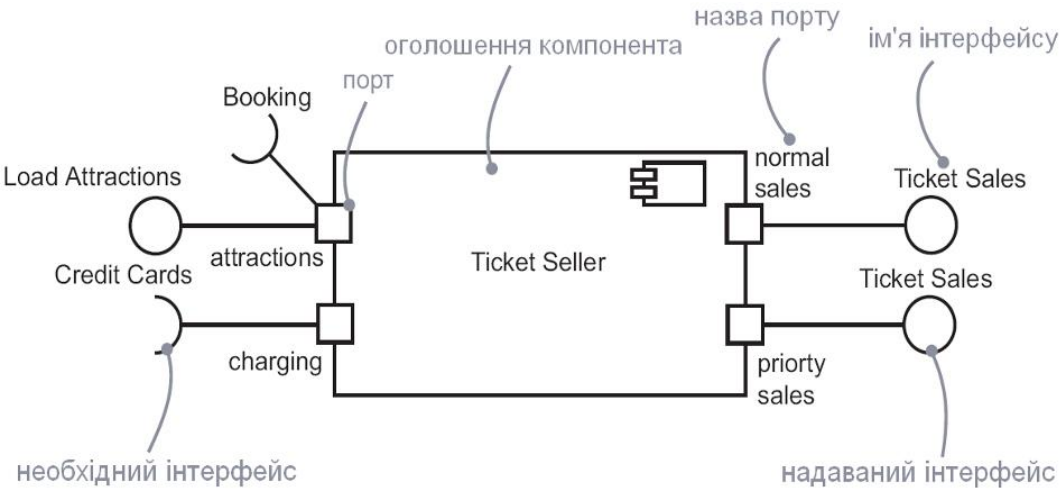


Рис. 2.12 Порти компоненти

Внутрішня структура. Компоненти можуть бути реалізовані як єдиний фрагмент коду, але бажано мати можливість створювати великі компоненти з менших компонентів, які використовуються як будівельні блоки у великих системах. Внутрішня структура компонента включає частини, з яких складається його реалізація. У більшості випадків внутрішні частини можуть бути екземплярами менших компонентів, статично з'єднаних через порти, які можуть забезпечити необхідну поведінку без необхідності моделювальника вказувати додаткову логіку.

Частина - це одиниця реалізації компонента з присвоєним ім'ям та типом. Екземпляр компонента містить один або декілька екземплярів кожної частини заданого типу. Частини мають кратність в межах компонента. Якщо ця кратність більша за одиницю, екземпляр компонента може містити будь-яку кількість екземплярів компонентів цього типу. Якщо кратність не представлена одним цілим числом, кількість екземплярів компонента може бути різною для кожного екземпляра компонента. Екземпляри компонентів створюються з мінімальною кількістю частин (решта додається пізніше, якщо це необхідно). Атрибути класу є різновидом компонента і мають тип та множинність. Кожен екземпляр класу має один або декілька екземплярів такого атрибуту [23].

На рисунку 2.13 показано компонент компілятора, який складається з чотирьох типів компонентів. Це лексичний аналізатор, синтаксичний аналізатор, генератор коду та від одного до трьох оптимізаторів. Повнішу версію компілятора можна налаштувати з різними рівнями оптимізації. У цій версії необхідні оптимізатори можна вибрати під час виконання.

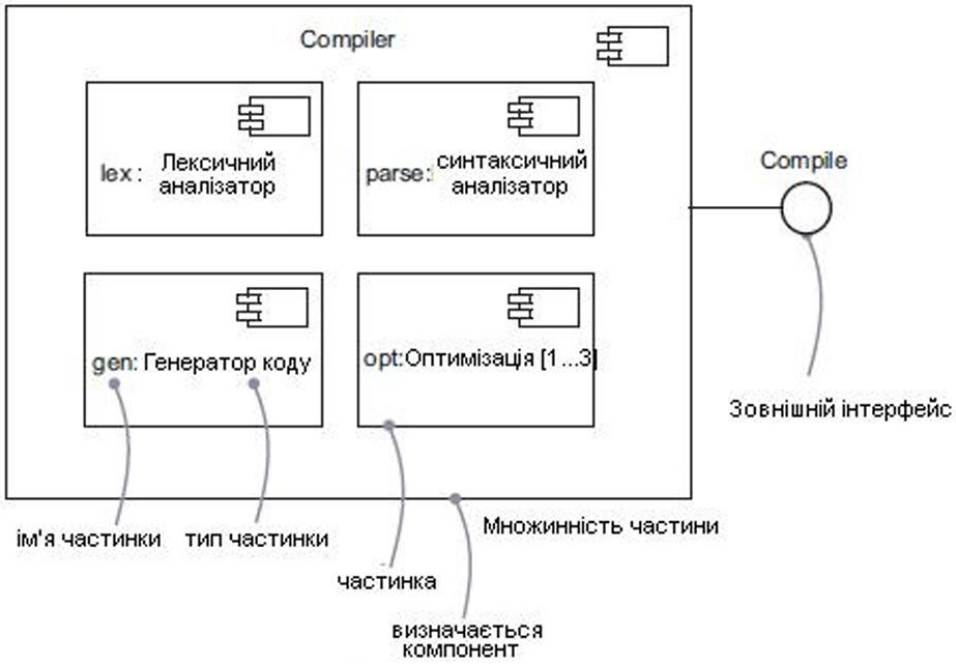


Рис. 2.13 Частини компоненти

Частини відрізняються від класів. Кожна частина ідентифікується за назвою так само, як атрибути розрізняються в класі. У компоненті може бути декілька частин одного типу, які виконують різні функції. Їх можна розрізнити за їхніми назвами. На Рисунку 2.13 2.13 компонент продажу квитків може містити окремі частини продажу для постійних та частих клієнтів. Обидві виконують однакову функцію, але перша обслуговує лише особливих клієнтів, збільшуючи можливість уникнути черг і пропонуючи різні переваги. Оскільки це однотипні компоненти, їх слід розрізняти за назвою: Інші два компоненти типу «Призначення місць» та «Управління запасами» існують як єдиний екземпляр в рамках компонента «Продаж авіаквитків» і тому не потребують назви [24].

Якщо компоненти є компонентами з портами, вони можуть з'єднуватися один з одним через ці порти; два порти можуть бути з'єднані разом, якщо один з них надає цей інтерфейс, а інший вимагає його. З'єднання портів означає, що порт, якому потрібна послуга, викликає порт, який надає інтерфейс, щоб отримати цю послугу. Перевага портів та інтерфейсів полягає в тому, що більше нічого не має значення. Порти можуть бути з'єднані між собою, якщо інтерфейси сумісні. Інструменти можуть автоматично генерувати код для виклику одного компонента з іншого. Вони також можуть підключатися до інших компонентів, які надають такий самий інтерфейс, якщо вони доступні.

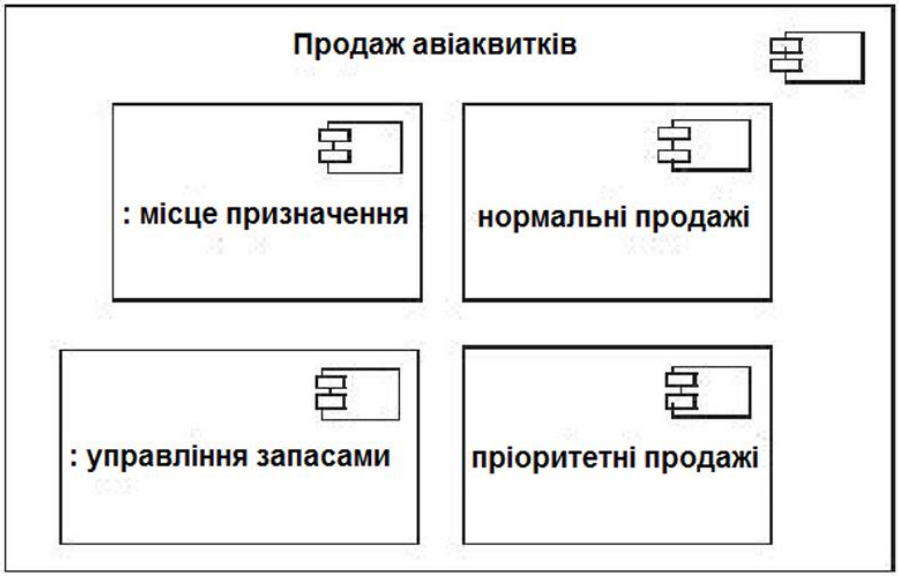


Рис. 2.14 Частини одного типу

Конектор - це «дріт», який з'єднує два порти разом. У прикладі компонента, який він охоплює, це або просте з'єднання, або тимчасове з'єднання.

Просте з'єднання є прикладом звичайного зв'язку.

Тимчасове з'єднання - це зв'язок використання між двома компонентами. Замість звичайного зв'язку, він може бути забезпечений параметрами процедури або локальними змінними, що є метою операції. Перевага портів та інтерфейсів полягає в тому, що два компоненти не

повинні «знати» один про одного на етапі проектування, якщо інтерфейси сумісні.

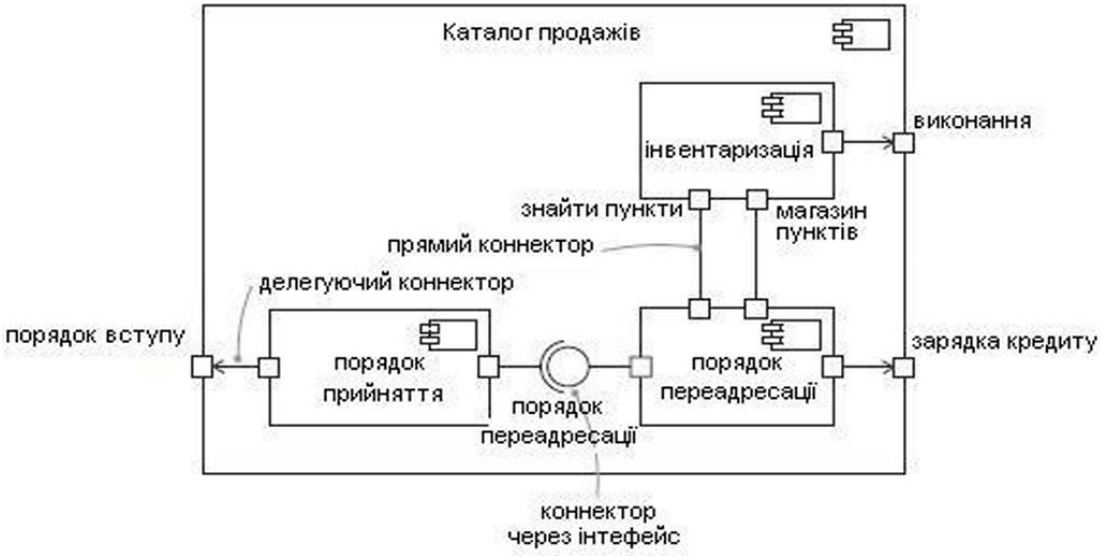


Рис. 2.15 Діаграма компонентів з використанням конекторів

З'єднувачі можна зобразити двома способами (рис. 2.15): якщо два компоненти чітко з'єднані (безпосередньо або через порти), достатньо намалювати лінію між ними або на портах, оскільки ці два компоненти мають сумісні інтерфейси Коли вони з'єднані один з одним, можна використовувати позначення у вигляді кулі та розетки, щоб вказати, що між цими компонентами немає постійного з'єднання, але вони з'єднані всередині компонента, який їх містить. У будь-який час можна замінити інші компоненти, якщо вони задовольняють вимогам інтерфейсу.

Також можна підключити внутрішній порт до зовнішнього порту компонента. У цьому випадку повідомлення із зовнішнього порту делегуються на внутрішній порт, який є з'єднувачем делегата. Таке з'єднання позначається стрілкою, що вказує від внутрішнього порту до зовнішнього. Це можна показати двома способами. По-перше, можна припустити, що внутрішній порт ідентичний зовнішньому, знаходиться в кінці і дозволяє підключення ззовні. По-друге, слід зазначити, що повідомлення, які надходять на зовнішній порт, негайно перенаправляються на внутрішній порт і навпаки. Процес однаковий в обох випадках.

На рисунку 2.15 показано використання внутрішніх портів та різних типів коннекторів: Зовнішні запити до порту OrderEntry передаються на внутрішній порт підкомпонента Ordering. Цей компонент, у свою чергу, надсилає свій вихід на порт Orderhandoff. Останній з'єднаний з підкомпонентом Orderhandling за схемою «куля і розетка». Цей тип з'єднання передбачає, що компоненти не розпізнають один одного: Компонент Orderhandling взаємодіє з компонентом Inventory для отримання товарів зі складу. Ця взаємодія представлена прямою конвекцією. Оскільки інтерфейс не показано, можна припустити, що цей зв'язок є тіснішим, тобто забезпечує сильніший зв'язок.

Як тільки товар знаходиться на складі, компонент Обробка замовлень зв'язується із зовнішнім кредитним сервісом. Як тільки зовнішня кредитна служба відповідає, компонент Orderhandling зв'язується з іншим портом Shipltems компонента Inventory, щоб підготувати замовлення до відправки. Компонент Inventory зв'язується з зовнішньою службою Fullfillment, щоб виконати доставку.

На діаграмі компонентів показано структуру повідомлень компонентів та можливі способи доставки. Однак вона не показує послідовність повідомлень всередині компонента. Послідовності та інші типи динамічної інформації можуть бути представлені за допомогою діаграм взаємодії.

Моделювання структурованих класів. Структурований клас може використовуватися для моделювання структур даних, де елементи мають контекстнозалежні зв'язки, можливі лише в межах цього класу. Звичайні атрибути або асоціації можуть визначати складові класу, однак частини не можуть бути взаємопов'язані на простій діаграмі класів. Клас, чия внутрішня структура представлена частинами та конекторами, дозволяє розв'язати цю проблему.

Для моделювання структурованого класу важливо виконати наступні кроки: визначити внутрішні компоненти класу та їхні типи; надати кожній

частині назву, що відображає її функцію в структурованому класі, а не виключно її тип; зобразити конектори між компонентами, що забезпечують комунікацію або контекстно-залежні зв’язки; при необхідності використовувати інші структуровані класи як типи частин, з урахуванням того, що під’єднання до компонентів всередині іншого класу недоступне – можна підключатися лише до їхніх зовнішніх портів. Об'єкт "покупець" є прикладом класу "Людина" і може мати або не мати статус "пріоритету", тому частина "пріоритет" має множинність 0..1. Конектор від "покупця" до "пріоритету" має таку ж множинність. Оскільки кожен клієнт має право бронювати одне або кілька місць, об'єкт "місце" теж має множинність. Немає необхідності показувати конектор від "покупця" до "місць", оскільки ці об'єкти вже знаходяться в одному структурованому класі. Звернімо увагу на клас "Категорія", обведений пунктирною рамкою. Це означає, що цей компонент є посиланням на об'єкт, який не належить структурованому класу. Посилання створюється і знищується разом із екземпляром класу "Замовлення квитків", але екземпляри "Категорії" незалежні від цього класу. Частина "місце" підключена до посилання "категорія", бо замовлення можуть включати місця різних категорій, і кожне бронювання повинно відповідати певній категорії місць. З множинності видно, що кожне резервування місця строго підключене до одного об'єкта "Категорія" [25].

Моделювання прикладних програмних інтерфейсів (API) є важливим аспектом при розробці систем, що складаються з компонентних частин. Ці інтерфейси забезпечують зв'язок між складовими системи. API можна уявити як програмні з’єднання, що моделюються за допомогою інтерфейсів і компонентів. Вони реалізуються різними компонентами, і для розробника важливо знати, які саме компоненти виконують функції інтерфейсу. Однак з точки зору управління конфігурацією системи, часто немає потреби візуалізувати всі реалізації одночасно, оскільки багато з них можуть бути досить зрозумілими та повторюваними.

Основним завданням є залишення операцій на периферії моделей та використання інтерфейсів як дескрипторів, які дають змогу знайти всі відповідні набори операцій. При створенні реальних систем на базі API, важливо деталізувати моделі настільки, щоб інструменти розробки могли компілювати код згідно з властивостями цих інтерфейсів. Поряд зі сигнатурою кожної операції може бути потрібне відображення пояснюючих матеріалів щодо правильного використання інтерфейсу.

Щоб ефективно змоделювати API, необхідно: визначити з'єднання в системі та змоделювати їх як інтерфейси, об'єднуючи атрибути і операції, що їх утворюють. Відображайте тільки ті властивості інтерфейсу, які важливі для візуалізації в конкретному контексті, а зайві деталі сховуйте у специфікації інтерфейсу для подальшого використання. На рисунку 2.16 показано API компоненти анімації, що включає чотири інтерфейси, які є частиною API: Iapplication.

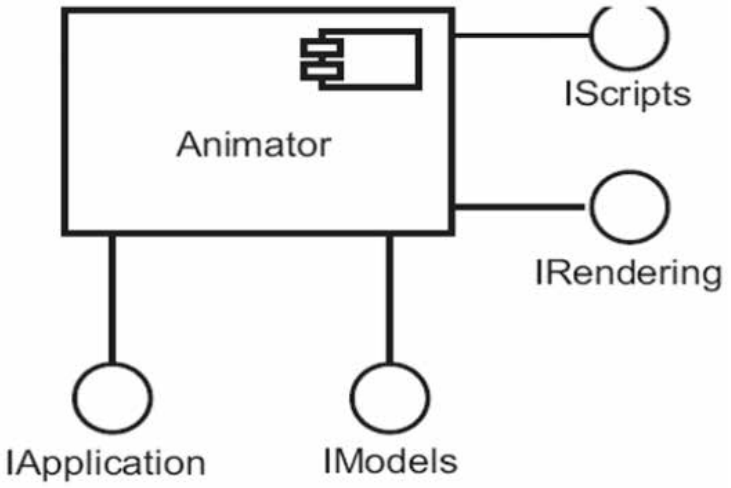


Рис. 2.16 Моделювання API

Компоненти служать засобом для інкапсуляції окремих частин системи, що сприяє зменшенню кількості залежностей між її елементами та робить ці залежності більш очевидними. Це також підвищує можливість взаємозамінності та адаптивності, особливо коли виникає необхідність у модернізації або зміні системи в майбутньому. Високоякісна компонента повинна володіти такими характеристиками:

 інкапсулює сервіс із добре окресленим інтерфейсом і межами;

 має внутрішню структуру, яка допускає можливість її описування;

 не комбінує незв’язаної функціональності в межах однієї одиниці;

 організовує зовнішню поведінку, використовуючи інтерфейси й порти в невеликій кількості;

 взаємодіє тільки через оголошені порти.

Коли потрібно продемонструвати реалізацію компонента з використанням вкладених підкомпонентів, важливо уникати надмірного поділу на підкомпоненти. Якщо їх стає настільки багато, що вони не вміщуються на одній сторінці, слід застосувати додаткову декомпозицію для деяких із них. Переконайтеся, що підкомпоненти взаємодіють лише через визначені порти та коннектори; ідентифікуйте підкомпоненти, які прямо контактують із зовнішнім середовищем, і моделюйте їх з допомогою делегуючих конекторів. При зображенні компонентів у UML важливо дати їм назви, які чітко відображають їх призначення, так само іменувати інтерфейси. Називайте підкомпоненти та порти, якщо їх функції не можна визначити за типами або ж коли модель містить багато частин одного типу. Приховуйте зайві деталі реалізації на діаграмі компонентів; демонструйте динаміку компонентів за допомогою діаграм взаємодії.

2.3 Дослідження функціональних особливостей програмного забезпечення системи управління автономними динамічними об'єктами

Дослідження функціональних особливостей програмного забезпечення для управління автономними динамічними об'єктами зосереджується на кількох ключових аспектах, які визначають ефективність та надійність системи в умовах реального світу. Програмне забезпечення має бути здатне

до гнучкої адаптації залежно від мінливих параметрів середовища, таких як зміни траєкторії, перешкоди або інші об’єкти, що можуть вплинути на динаміку руху. Ця гнучкість забезпечується завдяки алгоритмам, здатним працювати в реальному часі, швидко обробляючи сенсорні дані та коригуючи дії системи.

Крім того, важливим компонентом є забезпечення стійкості програмного забезпечення до невизначеностей і нестабільності вхідних даних. Умови експлуатації автономних систем часто характеризуються високою динамічністю та неоднорідністю, що вимагає особливих підходів до обробки та фільтрації інформації. Це означає, що програмне забезпечення повинне не тільки ефективно обробляти великий обсяг даних, але й здатне компенсувати недоліки або неточності інформації від сенсорів, що може виникати через різні фактори, включаючи погоду, технічні збої або інші зовнішні впливи.

Ще одним критичним аспектом є інтеграція різних підсистем управління, де кожен модуль має виконувати свою функцію, але діяти синхронізовано з іншими. Програмне забезпечення повинно забезпечувати безперебійну взаємодію між модулями, відповідальними за навігацію, орієнтацію, енергоспоживання та комунікацію з іншими системами або об'єктами. Це особливо важливо для складних сценаріїв, коли автономний об'єкт має взаємодіяти з іншими системами в реальному часі, приймаючи рішення на основі інтегрованих даних.

Дослідження функціональних особливостей програмного забезпечення для управління автономними динамічними об'єктами потребує глибокого розуміння того, як система забезпечує стабільність та точність у прийнятті рішень у реальному часі. Програмне забезпечення має виконувати низку складних обчислень, які пов'язані з прогнозуванням поведінки об'єкта та потенційних сценаріїв його взаємодії з навколишнім середовищем. Це означає, що алгоритми повинні мати високу продуктивність для швидкого

реагування на непередбачувані зміни в оточенні, наприклад, появу нових перешкод чи зміну ландшафту.

Важливу роль відіграє здатність програмного забезпечення не тільки контролювати фізичний рух об'єкта, але й оптимізувати його дії з точки зору енергоефективності та раціонального використання ресурсів. Така оптимізація має враховувати складні взаємозалежності між різними параметрами системи, такими як швидкість, точність руху, та витрати енергії. Це вимагає від програмного забезпечення глибокого аналізу в режимі реального часу, щоб вибрати оптимальні стратегії з урахуванням поточних умов і наявних обмежень [26].

Ще одним важливим аспектом є здатність програмного забезпечення до автономного навчання. Оскільки автономні системи часто функціонують у нових і непередбачуваних середовищах, вони повинні постійно оновлювати свої моделі, використовуючи нові дані. Це дозволяє об'єктам адаптувати свої стратегії управління відповідно до нових умов, що є критично важливим для тривалої автономної роботи. У цьому контексті програмне забезпечення повинно підтримувати механізми машинного навчання та самовдосконалення, що забезпечує постійну оптимізацію процесів управління.

Таким чином, програмне забезпечення для автономних динамічних об'єктів є не просто інструментом для контролю, а складною інтелектуальною системою, яка здатна адаптуватися, навчатися та вдосконалюватися в процесі експлуатації, забезпечуючи надійну та ефективну роботу в умовах складної і мінливої реальності.

Отже, функціональні особливості програмного забезпечення для управління автономними динамічними об'єктами визначаються здатністю до швидкої адаптації, обробки великих обсягів даних в умовах невизначеності та інтеграції різних модулів для забезпечення надійної роботи в складних і динамічних середовищах.

2.4 Розробка структури інформації, що обробляється у проектованому програмному забезпеченні

Розробка структури інформації для програмного забезпечення, що управляє автономними динамічними об'єктами, ґрунтується на необхідності забезпечення безперервної обробки даних у режимі реального часу. Одним із ключових аспектів є формування ієрархічної моделі, в якій кожен рівень інформації виконує свою специфічну роль, але при цьому є інтегрованим у загальну структуру. Програмне забезпечення має працювати з даними, що надходять із зовнішнього середовища, з внутрішніх систем контролю об'єкта та взаємодії з іншими автономними елементами.

Важливим є визначення типів даних, які система повинна отримувати, обробляти та зберігати. Це не лише фізичні параметри руху або навігаційна інформація, а й метадані, що відображають стан системи в кожен момент часу. Такі дані повинні бути структуровані так, щоб забезпечити їх швидке використання для прийняття управлінських рішень. Це може включати комплексні зв'язки між вхідними та вихідними сигналами, алгоритмічними блоками обчислень, що контролюють зміну стану об'єкта.

Далі, інформація повинна бути організована таким чином, щоб дозволити ефективний доступ до критичних даних у будь-який момент роботи системи. Це вимагає, щоб структура оброблюваної інформації була достатньо гнучкою для швидкого масштабування під нові умови або зміни в задачах, які виконує система. Наприклад, якщо автономний об'єкт змінює своє середовище роботи, інформаційна структура повинна дозволяти швидке підключення нових джерел даних і їх інтеграцію в загальну систему обробки без потреби кардинальних змін в архітектурі.

Окрім цього, важливо забезпечити механізми для відстеження та аналізу історичних даних, щоб система могла враховувати попередній досвід

для покращення своїх рішень у майбутньому. Це дозволяє створити більш адаптивну та ефективну систему, яка вчиться на власному досвіді і оптимізує роботу в реальному часі, реагуючи на зміни в середовищі або зміни параметрів об'єкта.

Таким чином, структура інформації в проектованому програмному забезпеченні повинна бути організована таким чином, щоб забезпечити ефективну, гнучку та адаптивну обробку даних для забезпечення надійного управління автономними динамічними об'єктами.

Процес розробки програмного забезпечення включає кілька ключових етапів: визначення вимог, планування та проектування структури й функціональних можливостей програмного забезпечення, написання та кодування, а також тестування, що підтверджує відповідність роботи програмного забезпечення запланованим специфікаціям. Завершальним етапом є розгортання програмного забезпечення для його використання.

Аналогія між процесом розробки програмного забезпечення та будівництвом будинку на цифровому полотні є досить помітною, оскільки кожен етап розвитку програмного продукту відповідає важливим аспектам будівельного процесу. Не є дивним, що розробники часто отримують назву "архітектори".

На початку вам необхідно мати чіткий план або концепцію, подібно до архітекторської розробки проекту будинку. Цей план визначає, які функції програмне забезпечення виконуватиме та хто буде його кінцевим користувачем. Далі розпочинається процес написання коду розробниками, які виконують роль цифрових будівельників, адже саме код становить базові елементи програмного забезпечення. Після завершення створення програмного коду проводиться його ретельне тестування, яке можна порівняти з перевіркою новозбудованого будинку на відповідність запланованим характеристикам. За необхідності виявлені проблеми усуваються [27].

Цей комплексний процес є критично важливим у бізнес середовищі, оскільки забезпечує створення ефективних інструментів і програм, що мають на меті оптимізацію та підвищення ефективності діяльності компаній і окремих осіб.

Етапи процесу розробки програмного проекту

Життєвий цикл розробки програмного забезпечення (SDLC) для програмних проектів зазвичай включає кілька важливих етапів. Він починається з аналізу вимог, коли команда визначає та аналізує, які завдання має виконувати програмне забезпечення згідно з потребами користувачів або бізнесу. На етапі дизайну ці вимоги перетворюються в конкретний план, який описує вигляд і функціонування програмного продукту.

На етапі впровадження розробники пишуть код, реалізуючи задумане в дизайні. Після цього програмне забезпечення піддається тестуванню, де його ретельно перевіряють на наявність помилок і проблем. У разі успішного тестування відбувається розгортання, коли програма стає доступною для використання.

Завершальний етап — це технічне обслуговування, що передбачає регулярні оновлення та виправлення для забезпечення стабільної та безпечної роботи програми. Цей цикл є ітераційним, тобто може повертатися до попередніх стадій у випадку появи нових вимог чи виявлених недоліків, що гарантує безперервне вдосконалення програмного проекту.

Розглядаючи розробку програмного забезпечення глибше, виділяємо вісім основних етапів, що складають основу будь-якого успішного процесу створення ПЗ. Від початкового планування до остаточного розгортання кожен крок грає ключову роль в перетворенні ідеї на функціональний продукт протягом усього циклу розробки.

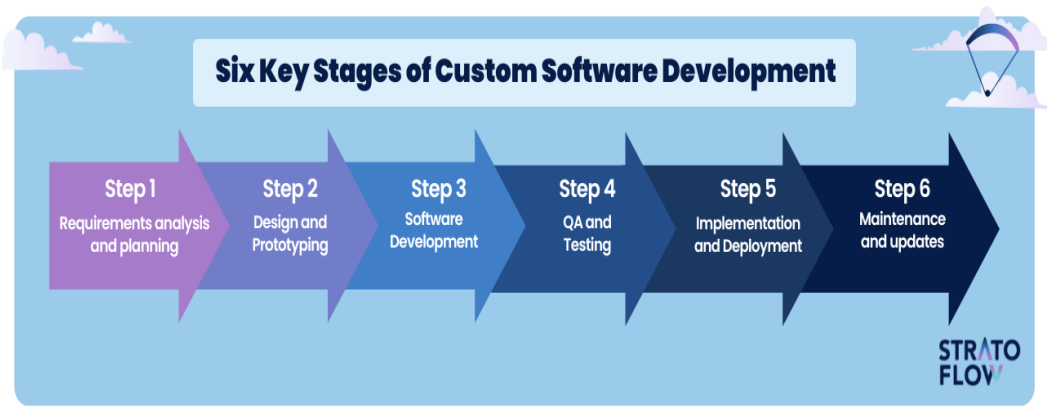


Рис. 2.17 Шість ключових етапів процесу розробки програмного забезпечення

Рис. 2.18 Аналіз вимог і планування

Ця перша фаза життєвого циклу розробки продукту пов'язана з розумінням того, що має робити програмне забезпечення.

Процес розпочинається з ретельного аналізу специфічних потреб та очікувань організації та її користувачів. Це означає активну взаємодію з усіма учасниками, щоб зібрати всебічні вимоги, що можуть охоплювати функціональні, технічні та операційні аспекти. Мова йде не лише про те, які завдання має виконувати програмне забезпечення, але й про те, як воно повинно функціонувати в різних контекстах. На цьому етапі приймаються важливі рішення щодо вибору методології та підходу до процесу розробки програмного забезпечення.

Цей вибір має істотний вплив на перебіг проекту. Обрана методологія формує основу для роботи команди, регулюючи такі елементи, як співпраця, етапи розробки та підхід до управління змінами. Вибір оптимальної методології є критичним завданням, оскільки вона повинна гармоніювати з

цілями проекту, командною динамікою та загальною стратегією бізнесу. Які ж методології розробки програмного забезпечення можуть взяти на озброєння команди розробників? Існує вісім популярних варіантів. У продовженні цієї статті ми детально розглянемо деякі з них.

- Гнучкі методології
- Scrum підхід
- Водоспадна методологія розробки ПЗ
- Методологія розробки ощадливої системи
- Канбан підхід
- DevOps
- Екстремальне програмування (XP)
- Швидка розробка додатків (RAD)

Рис. 2.19 Дизайн і прототипування

На другому етапі формуються концептуальні ідеї та вимоги.

На цьому етапі команда розробників створює детальну архітектуру програмного забезпечення, яка визначає технічні специфікації, потік даних, дизайн інтерфейсу користувача та функціональність системи. Це важливий етап, на якому приймаються стратегічні рішення щодо загальної структури та поведінки програмного забезпечення.

На цьому етапі команда розробників повинна вибрати технологічний стек і допоміжні технології, такі як Java для ядра, Kafka для обробки даних IMDG, Hazelcast і Oracle Coherence.

Два ключові компоненти, які зазвичай включаються в етап проектування, - це перевірка концепції (POC) і мінімальний життєздатний продукт (MVP).

POC - це невелике дослідження для перевірки певної концепції або теорії, яка може бути реалізована в процесі розробки програмного забезпечення. По суті, це прототипи, які розробляють інженери-програмісти, щоб продемонструвати здійсненність певного аспекту проекту. Створення POC дуже важливе, оскільки допомагає перевірити технічні припущення, зрозуміти потенційні проблеми та оцінити реалістичність ідеї до того, як інвестувати значний час і ресурси. Це надзвичайно важливо.

З іншого боку, MVP - це версії програмного забезпечення з достатньою функціональністю для перших користувачів, що дозволяє їм надавати зворотній зв'язок для подальшого розвитку продукту. Це стратегія швидкого та кількісного тестування продуктів і функцій на ринку; MVP мають вирішальне значення на етапі проектування, оскільки вони зосереджуються на основній функціональності та допомагають командам зрозуміти, що є найважливішим для користувачів.

Інтегрування концептуального доказу (Proof of Concept, POC) та мінімально життєздатного продукту (Minimum Viable Product, MVP) на стадії проектування є надзвичайно важливим для зменшення ризиків, оптимізації використання ресурсів і забезпечення того, що програмне забезпечення стане конкурентоспроможним та цінним на ринку. Ці компоненти направляють процес розробки, щоб створити продукт, який є не лише технічно досконалим, але й готовим до виходу на ринок та орієнтованим на потреби кінцевого користувача [28].

Рис. 2.20 Розробка програмного забезпечення

Кодування зазвичай вважається основою процесу розробки програмного забезпечення, і розробники втілюють дизайн в життя, пишучи власне програму.

На цьому етапі розробники перетворюють плани на працююче програмне забезпечення шляхом написання коду. Ця фаза є дуже важливою, оскільки вона перетворює заплановану концепцію на реальний продукт.

Ефективна комунікація із зацікавленими сторонами проекту на цьому етапі є ключем до забезпечення відповідності програмного забезпечення бізнес-цілям та очікуванням користувачів.

Регулярна взаємодія з клієнтами та зацікавленими сторонами дозволяє отримати негайний зворотній зв'язок і внести корективи, необхідні для приведення проекту у відповідність до бачення замовника.

Особливо ефективним може бути застосування методології, яка наголошує на коротких спринтах розробки та ітеративному підході до роботи, наприклад, гнучкої розробки. Такі спринти зазвичай тривають від одного до чотирьох тижнів і передбачають щоденну комунікацію між командою розробників і замовником.

Такий підхід дозволяє розробляти програмні продукти швидко, невеликими частинами та інкрементально.

Це гарантує, що проект постійно рухається в правильному напрямку, а будь-які відхилення або проблеми з управлінням проектом швидко вирішуються. Така часта співпраця має вирішальне значення для розробки

програмного забезпечення, його адаптації до мінливих вимог і забезпечення того, щоб кінцевий продукт дійсно відповідав потребам користувачів.

Основний висновок полягає в тому, що в процесах створення програмного забезпечення існують три фундаментальні цінності, які мають слугувати керівним напрямом усієї діяльності:

- інновації,
- співпраця,
- Спритність.

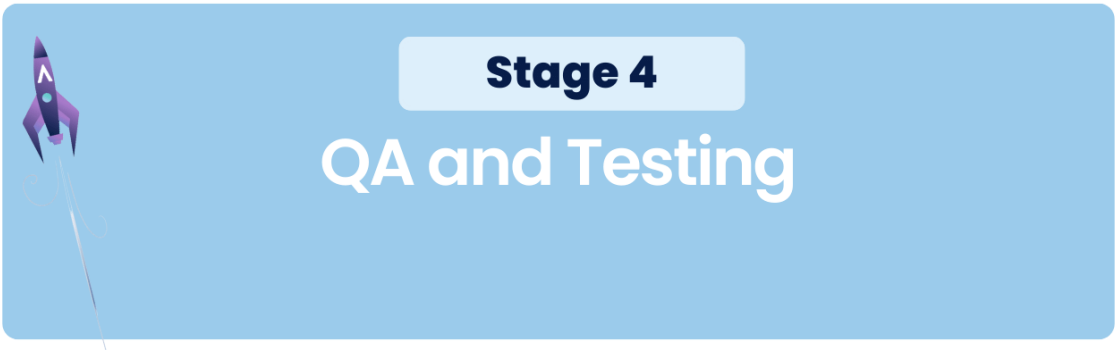


Рис. 2.21 Контроль якості та тестування

Тестування забезпечує високу якість.

На цьому етапі розробки програмного забезпечення здійснюється ретельне тестування для виявлення та виправлення будь-яких помилок чи недоліків у функціонуванні системи.

Значення тестування полягає у його здатності підтвердити відповідність програмного продукту встановленим вимогам та його надійну роботу за різних умов. Це є визначальним етапом у процесі постачання програмних продуктів високої якості, що підкреслює їхню надійність, зручність і ефективність.

Крім того, тестування сприяє покращенню продуктивності програмного забезпечення та підвищенню впевненості і задоволеності користувачів, гарантуючи безперебійну та безпомилкову роботу системи.

Основні типи тестування програмного забезпечення охоплюють:

• Модульне тестування: забезпечує перевірку правильної функціональності окремих компонентів або частин коду.

• Тестування продуктивності: оцінює швидкість, реакцію та стабільність програмного забезпечення за умови певного робочого навантаження для гарантії його швидкої та ефективної роботи.

• Інтеграційне тестування: забезпечує взаємну сумісність різних модулів або служб, що використовує програма.

• Тестування безпеки: критично важливо для виявлення вразливостей у програмному забезпеченні, щоб запобігти потенційним атакам або зламам, захищаючи дані користувачів та цілісність системи.

• Системне тестування: перевіряє повністю інтегрований програмний продукт, щоб гарантувати його відповідність усім визначеним вимогам.

• Тестування на зручність використання: оцінює, наскільки зручний та інтуїтивно зрозумілий інтерфейс програмного забезпечення для кінцевих користувачів.

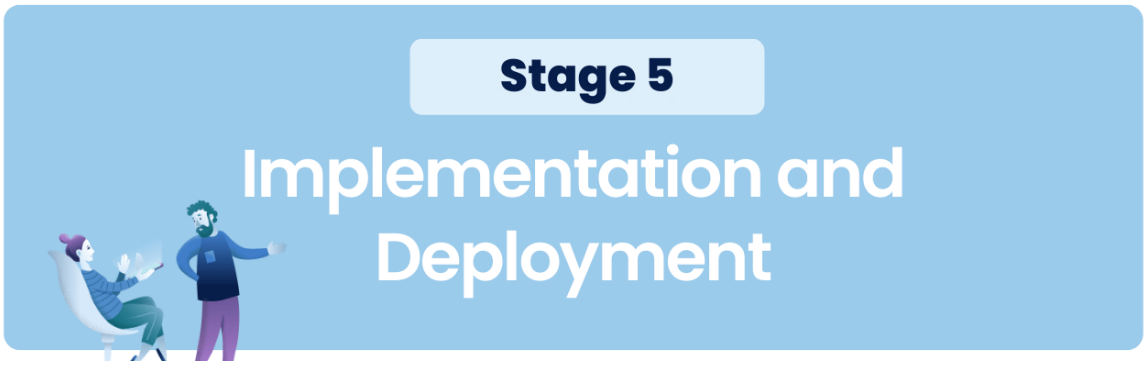


Рис. 2.22 Впровадження та розгортання

Реалізація та розгортання - це етапи, на яких програмне забезпечення починає набувати форми в реальному світі.

Подумайте про це як про урочисте відкриття цифрового продукту.

Впровадження передбачає інтеграцію програмного забезпечення в середовище користувача, що передбачає встановлення його на сервері, налаштування для використання, а іноді й перенесення даних зі старої системи в нову.

З іншого боку, розгортання - це процес надання програмного забезпечення користувачам. Це означає, що програмне забезпечення стає доступним для завантаження на загальнодоступній платформі, доступним через Інтернет або розповсюджується всередині організації.

Ця фаза є критично важливою, оскільки програмне забезпечення вперше використовується в реальному середовищі і може виявити проблеми, які не існують в тестовому середовищі. Це також етап, на якому необхідне ретельне планування, щоб забезпечити плавний перехід з мінімальними перешкодами для бізнесу користувача.

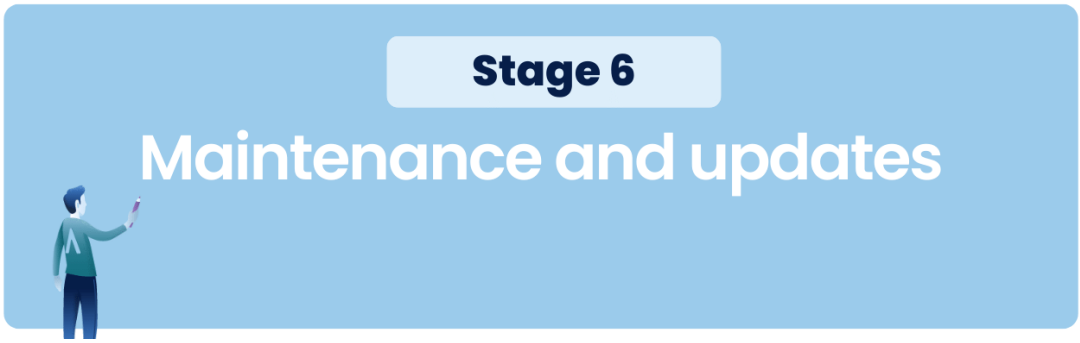


Рис. 2.23 Технічне обслуговування та оновлення

Після розгортання програмного забезпечення настає фаза його обслуговування.

Це передбачає регулярний моніторинг програмного забезпечення на наявність проблем, виправлення помилок і внесення необхідних змін для підвищення продуктивності або додавання нових функцій.

Ця фаза вимагає регулярних оновлень не лише для покращення програмного забезпечення, але й для захисту його від вразливостей безпеки.

Регулярні оновлення допомагають підтримувати відповідність програмного забезпечення технологіям, що розвиваються, і потребам користувачів, що змінюються, а також запобігають його застаріванню. Це особливо важливо, щоб уникнути проблем, спричинених застарілим програмним забезпеченням, яке є дорогим і неефективним в обслуговуванні.

На цьому етапі також важлива хороша технічна документація. Чітке документування змін, оновлень і поточного стану програмного забезпечення робить майбутню підтримку і вдосконалення простішими та ефективнішими. Ця документація є цінним ресурсом для нових членів команди та розробників, які можуть працювати над програмним забезпеченням пізніше.

Як прискорити процес розробки програмного забезпечення

Розробка програмного забезпечення на замовлення забезпечує індивідуальне рішення, яке ідеально відповідає унікальним потребам компанії, але не секрет, що цей підхід може бути дорогим і трудомістким.

Значні витрати часу та ресурсів, необхідні для розробки програмного забезпечення на замовлення, часто обмежують доступність цього підходу і створюють величезний тягар для багатьох компаній, особливо тих, що мають невеликі або обмежені бюджети.

Ця проблема змушує організації шукати альтернативні способи скоротити час і витрати, зберігаючи при цьому переваги кастомного програмного забезпечення.

Найкращі методології розробки програмного забезпечення, які варто знати

У цьому розділі ми детальніше вивчимо провідні методології розробки програмного забезпечення, що вплинули на сучасні практики, від Agile до DevOps. Знання цих методологій дозволяє ефективно управляти проектами та швидко адаптуватися до змінних вимог технологічного світу.

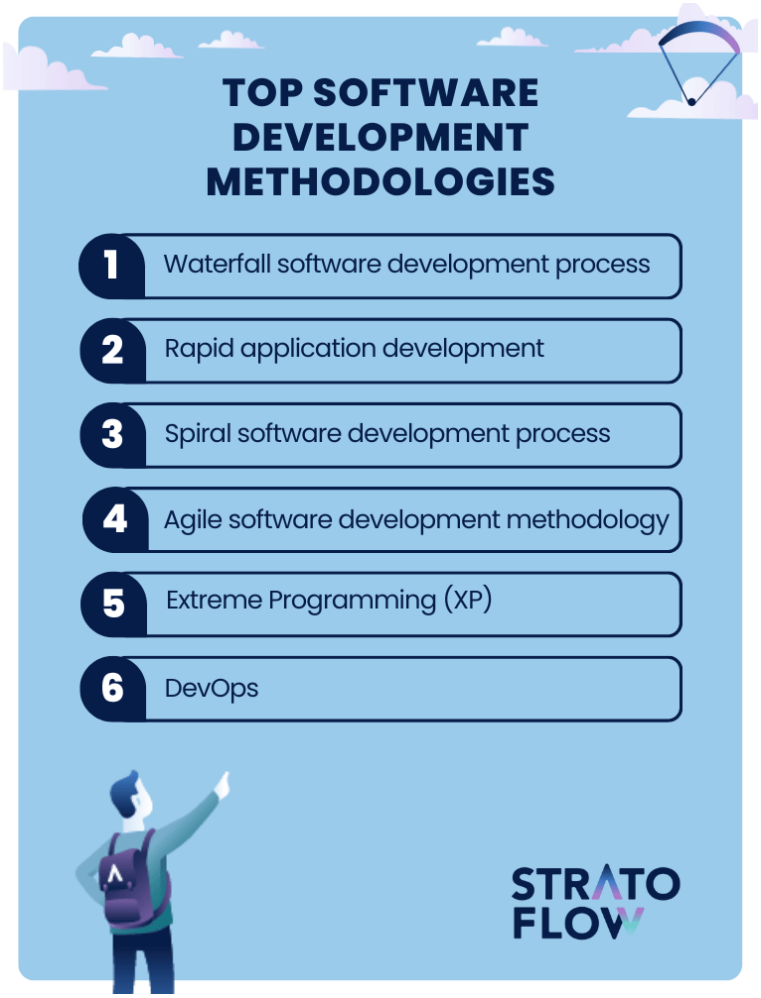


Рис. 2.24 Найкращі методології розробки програмного забезпечення

Процес розробки програмного забезпечення за моделлю водоспаду

Модель водоспаду, найстаріша з усіх методологій, є лінійним і послідовним підходом до розробки програмного забезпечення, в якому кожна фаза завершується до того, як починається наступна. Це схоже на покроковий процес, де ви не можете повернутися до попереднього етапу, не почавши з самого початку. Метод починається з етапу детального планування та збору вимог, за яким слідують проектування, реалізація, тестування, розгортання та супровід. Модель водоспаду проста і зрозуміла, що робить її придатною для невеликих проектів з чітко визначеними вимогами. Однак, вона може бути невигідною в динамічних середовищах через її негнучкість до змін, якщо проект вже запущено [29].

2.5 Висновки по розділу

З точки зору виявлення основних аспектів проблеми та визначення науково-технічних завдань автономного динамічного керування об'єктами, основні виклики пов'язані з ефективною адаптацією до невизначеності системи, підвищенням точності сенсорних даних та інтеграцією різних рівнів керування. Невизначеність навколишнього середовища та мінливість зовнішніх умов вимагають розробки нових підходів до обробки даних та прийняття рішень в режимі реального часу. Крім того, розробка алгоритмів підвищення надійності, гнучкості та енергоефективності є вирішальним фактором забезпечення безпеки та автономності таких установок. Інтеграція сенсорної інформації, модулів управління та прогнозування майбутньої поведінки об'єктів залишається значним науково-технічним викликом.

Підсумовуючи нашу подорож через процес розробки програмного забезпечення, ми сподіваємося, що ви отримаєте краще розуміння різних етапів, методологій та інноваційних підходів до розробки програмного забезпечення.

Від ретельного планування на ранніх етапах до гнучкості, яку пропонують гнучкі методології та сучасні рішення, такі як Openkoda, цей посібник має на меті надати вам знання, які допоможуть зорієнтуватися у складному процесі розробки програмного забезпечення.

3 Розробка програмного забезпечення системи управління автономними динамічними об'єктами

3.1 Вибір архітектурних особливостей продукту

Автономні системи, такі як безпілотні літальні апарати, як у повітрі, так і на суші чи на морі, стають дедалі складнішими і мають кілька рівнів управління та автоматизації (Рис. 3.1). Зі збільшенням вимог до маневреності та експлуатаційних характеристик, електроніка стає складнішою, а інтеграція апаратного і програмного забезпечення перетворюється на серйозний виклик. Ключовими елементами електроніки транспортного засобу є сенсорні підсистеми для позиціонування і розвідки, виконавчі підсистеми для руху транспортного засобу і підсистеми управління для навігації, наведення і виконання специфічних для місії завдань. Наприклад, використовуються дедалі складніші датчики з функціями, що перекриваються і дублюються. Застосовуються складні алгоритми об'єднання декількох сенсорів і підтримується декілька режимів роботи системи. Різні способи зондування працюють разом з алгоритмами управління на декількох абстрактних рівнях завдань для виконання завдання.

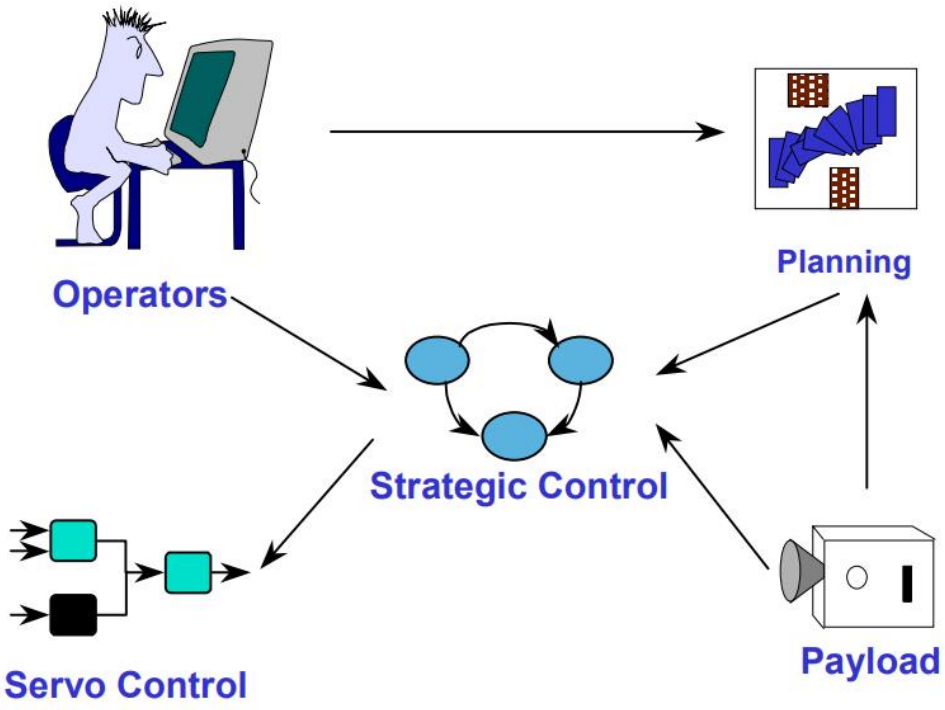


Рис. 3.1 Елементи типової автономної системи

Крім того, програмне забезпечення в автономних системах розробляється на різних рівнях функціональності. На найнижчому рівні знаходяться драйвери пристроїв, які взаємодіють із апаратним забезпеченням. Цей прошарок абстрагує апаратне забезпечення в такий спосіб, що стає зручним для використання на вищих рівнях функціональності. Драйвери пристроїв повинні бути високоефективними і зазвичай передбачають ручне написання коду мовами C або C++. Іноді драйвери надаються виробником апаратного забезпечення і потребують обгортання в спеціальний код для інтеграції з рештою системи. Наступний рівень функціональності охоплює низькорівневі управління сервосу, необхідні для досягнення первинних управлінських поведінок. Програмне забезпечення на цьому рівні характеризується періодичним циклом обробки даних, що виконується з фіксованими частотами. Як правило, це програмне забезпечення також написане вручну на мовах C або C++. Наступний рівень функціональності базується на примітивних сервосистемах і додає вищий рівень інтелекту та управління. Тут програмне забезпечення зазвичай має подієво-орієнтовану природу, з умовною логікою і прийняттям рішень на основі даних сенсорів. Розробка цього рівня може передбачати різноманітні підходи, включаючи умовну логіку при ручному написанні коду на мовах C або C++, графічні кінцеві автомати та використання інструментів моделювання, таких як MATLAB і Simulink, для аналізу та проектування загальної продуктивності системи. Таким чином, програмна архітектура повинна забезпечити «опорядження», необхідне для інтеграції програмного забезпечення на різних рівнях абстракції в автономній системі [30].

Сучасні автономні системи за своєю природою є розподіленими, що включає кілька обчислювальних вузлів. Крім того, вони зазвичай географічно розосереджені та забезпечені наземними станціями для моніторингу та керування місіями, а також GPS-станціями для визначення місця розташування. У майбутньому може бути розгорнуто ескадрильї

транспортних засобів, які діятимуть як група, де транспортні засоби повинні будуть спілкуватися і координувати свої дії для виконання завдань. На сьогодні таке програмне забезпечення для зв'язку розробляється з використанням різноманітних технік та проміжного програмного забезпечення для з'єднання. Вони варіюються від спеціалізованих саморобних методів і власних рішень до готових об'єктно-орієнтованих проміжних рішень, таких як CORBA®, та орієнтованих на дані систем публікації-підписки, таких як NDDS®. Таким чином, програмна платформа для автономних систем має також надавати інфраструктуру зв'язку для забезпечення розгортання розподілених систем.

У подальшій частині статті розглядаються такі питання: програмні платформи та платформи для автономних систем разом з їх характеристиками та перевагами використання; приклади застосування програмних платформ для автономних систем; висновки та короткий огляд майбутніх тенденцій у цій сфері.

ПРОГРАМНІ ФРЕЙМВОРКИ ДЛЯ АВТОНОМНИХ СИСТЕМ

Програмні фреймворки починають активно використовуватися в розробці програмного забезпечення для безпілотних транспортних засобів та роботизованих систем. Прикладом таких фреймворків з використанням у безпілотних транспортних засобах є Open Control Platform (OCP), яке фінансується DARPA і було розроблене компанією Boeing для підтримки дослідницьких проектів за Допомогою Програмного Забезпечення (SEC). Ще одним прикладом фреймворків є ControlShell®, що спочатку розроблявся дослідниками з Лабораторії Аерокосмічної Робототехніки Стенфордського університету і наразі продовжує вдосконалюватися компанією Real-Time Innovations, Inc.

Діяльність в рамках OCP включає участь багатьох учасників проектів SEC, серед яких дослідницькі платформи Каліфорнійського технологічного інституту, Технологічного інституту Джорджії, Орегонського Інституту

Вищої Освіти, Університету Міннесоти та інших провідних академічних установ. Як інструмент розробки з комерційною підтримкою COTS*, ControlShell та проміжне програмне забезпечення для розподілу даних, NDDS, знайшли широке застосування у сфері безпілотних транспортних засобів, серед яких:

• Автономний підводний апарат OTTER розроблений в Лабораторії аерокосмічної робототехніки при Стенфордському університеті та Монтерейському інституті досліджень акваріумів.

• Телекерований підводний апарат QUEST від Alstom Schilling Robotics.

• Система Robonaut, що розробляється НАСА в Космічному центрі імені Джонсона.

Використання програмного каркаса може значно скоротити час розробки під час створення надійного та гнучкого програмного забезпечення. Однією з вагомих переваг програмних каркасів є надання чітко визначеного механізму інтеграції, який сприяє зменшенню часу розробки. Такі механізми забезпечують інтеграцію низькорівневих драйверів, рукописного чи кастомізованого коду, телеметрійного та комунікаційного програмного забезпечення, а також інтелектуальних моделей у стабільний спосіб. Зі швидким прогресом моделей розробки та автоматизованої генерації коду виробничої якості, програмні каркаси стають все більш цінними для архітекторів [31].

Наступні підрозділи описують ключові характеристики програмних каркасів та їхні переваги у розробці безпілотних систем транспортних засобів. Для ілюстрації розглянемо розробку системи круїз-контролю транспортного засобу з використанням програмної платформи. У цьому простому прикладі (Рис. 3.2) людський оператор (водій) подає команди круїз-контролеру для увімкнення або вимкнення автоматичного режиму, відновлення швидкості, прискорення чи руху накатом. Водій також може безпосередньо керувати транспортним засобом, наприклад, натискаючи

педаль газу, гальмуючи чи перемикаючи передачі. Круїз-контролер використовує систему зворотного зв'язку для досягнення бажаної швидкості руху, заданої оператором.

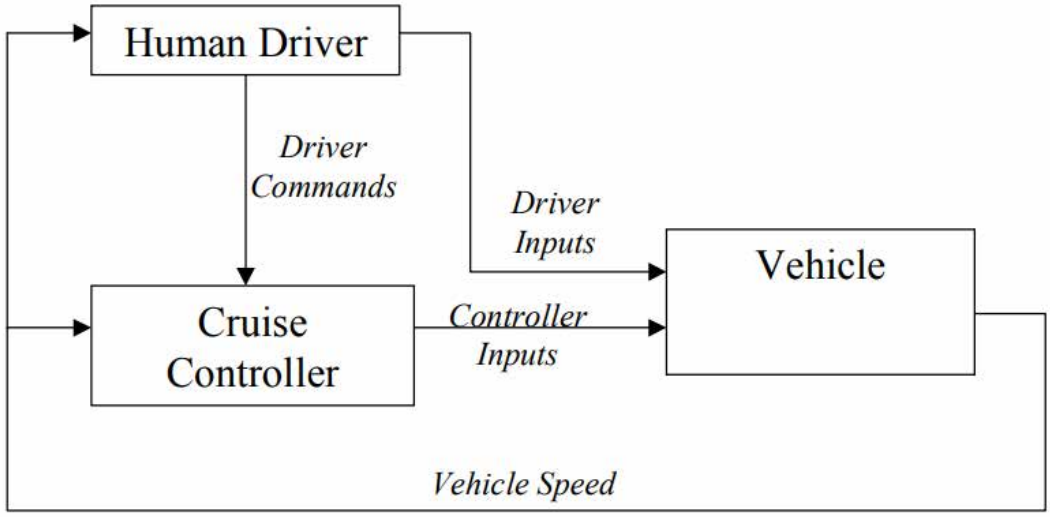


Рис. 3.2 Приклад системи круїз-контролю автомобіля

КОМПОНЕНТНО-ОРІЄНТОВАНИЙ ПІДХІД

Програмні бази для автономних систем є компонентно-орієнтованими. Програмний компонент є інкапсульованим блоком програмного забезпечення, що виконує певну чітко визначену функціональність. Наприклад, на рис. 3.3 представлено графічне зображення компонента зворотного зв'язку для контролера круїз-контролю. Цей компонент має інтерфейс, що включає вхідні дані (швидкість), вихідні дані (дросель круїзу), а також довідкові дані (зміщення швидкості, зміщення прискорення). Крім того, він має порт (команди круїз-контролю) для прийому команд від водія.

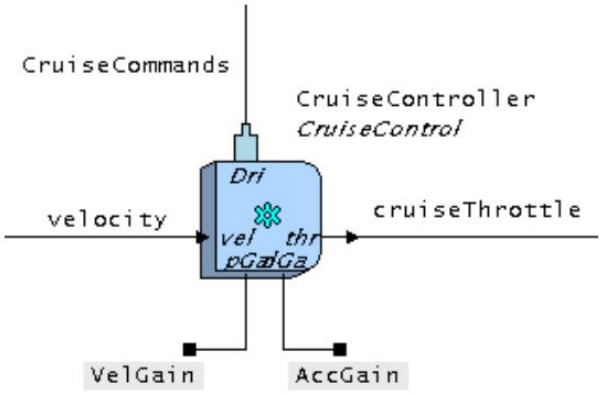


Рис. 3.3 Програмна складова круїз-контролера

Крім портів для передачі даних, компоненти можуть також забезпечувати послуги та внутрішньокомпонентне спілкування на основі викликів методів і обміну повідомленнями. Програмна платформа керує з'єднаннями та розподіляє механізм спілкування між компонентами. Архітектура та організація програмного забезпечення легко відображаються у системі компонентів. Переваги підходу, заснованого на компонентах, включають:

• Повторне використання: компоненти можуть бути розроблені, протестовані та верифіковані один раз, а потім використовуватися в різних місцях.

• Ясність: компоненти виділяють ключові структурні елементи програмної реалізації і часто відповідають фізичним компонентам.

• Переконфігурованість: компоненти можуть легко "перепідключатися" у різній топології з'єднань.

• Взаємодія: компоненти з сумісними інтерфейсами можуть обмінюватися даними через ці інтерфейси.

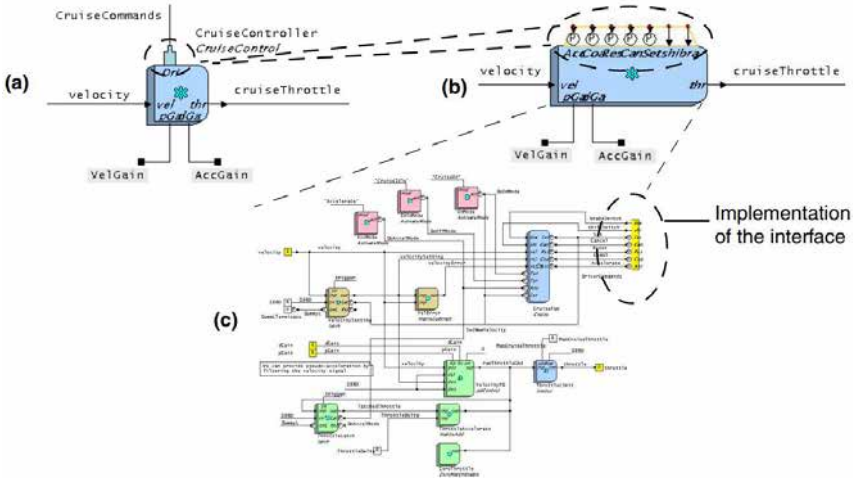


Рис. 3.4 Компонент Cruise Controller, що реалізує програмний інтерфейс Cruise Controller: (a) компактний зовнішній вигляд компонента, (b) розширений зовнішній вигляд компонента, (c) ієрархічно розширений внутрішній вигляд компонента, що розкриває деталі впровадження

ІНТЕРФЕЙСИ ВІДДІЛЕНІ ВІД ІМПЛЕМЕНТАЦІЇ КОМПОНЕНТІВ

У програмних каркасах чітко розмежовуються інтерфейси та їх реалізації. Інтерфейси можуть виступати як самостійні елементи в програмному каркасі. Вони визначають дані і методи, які мають підтримуватися компонентом, що реалізує цей інтерфейс. Інтерфейси можуть бути ієрархічно складеними. Реалізації можуть надаватися різноманітними компонентами.

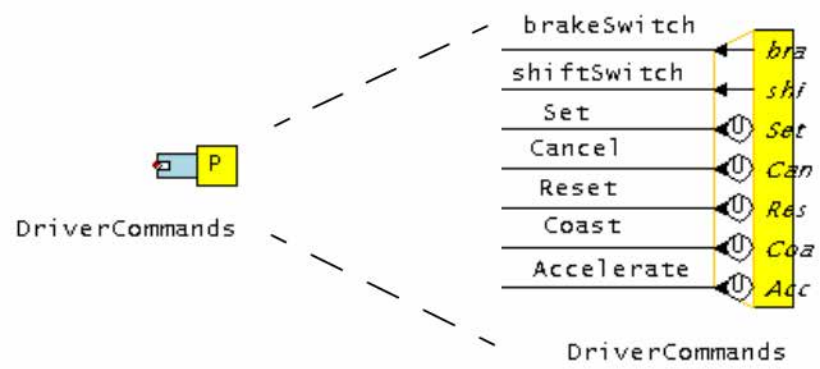


Рис. 3.5 «Програмний інтерфейс» круїз-контролера. Компактний вигляд можна розгорнути, щоб побачити деталі

Основні переваги розділення інтерфейсів компонентів від їх реалізацій включають в себе:

• Заміна: реалізацію компонента легко можна замінити іншою, якщо вона дотримується того самого інтерфейсу.

• Розширюваність: вже існуючий компонент можна вдосконалити для надання додаткових функцій шляхом впровадження нових інтерфейсів компонентів.

• Командна розробка: інтерфейси можуть бути поділені між членами команди, які забезпечують різні реалізації компонентів, що повинні взаємодіяти.

• Використання експертизи в цій галузі: кожен рівень функціональності системи може вимагати різного набору навичок і знань. Завдяки загальній програмній архітектурі, що охоплює всі рівні функціональності, чітко

визначені інтерфейси можуть бути створені та реалізовані експертами в цій області.

Ієрархічність

Програмні структури для автономних систем, як правило, мають ієрархічну організацію. Компоненти можуть бути систематизовані ієрархічно шляхом взаємозв'язку інших елементарних одиниць (рис. 3.4(в)). Це дозволяє створювати складну функціональність шляхом комбінування простих базових компонентів у різноманітні способи.

Первинні базові компоненти зазвичай реалізуються за допомогою таких текстових мов програмування, як C або C++. Деякі фреймворки також надають графічні редактори для збирання компонентів у більш складні функціональні одиниці (рис. 3.4). Можна створювати користувацькі компоненти та поєднувати їх із уже вбудованими, що дає можливість швидко створювати робочі програми.

Основні переваги ієрархічних компонентів включають:

- Управління складністю: програмні елементи можна оформити у формі узгоджених компонентів, які виконують чітко визначене завдання.

- Програмування на високому рівні: складний компонент з новою функціональністю може бути створений шляхом простого комбінування наявних компонентів.

- Масштабованість і багаторівневоість: вищі рівні абстракції можуть бути реалізовані шляхом поєднання базових компонентів нижчого рівня.

СПЕЦИФІЧНІ ГАЛУЗІ ПОВЕДІНКИ

Структура програмного каркаса для автономних систем включає в себе області виконання, які особливо підходять для програмних архітектур багатошарових систем управління. Два основні домени виконання для автономних систем — це синхронізований потік даних у дискретному часі та поведінка подійної скінченної автомати (фінал-анальний автомат).

Синхронізований потік даних (Рис. 3.6) представляє собою відомі блок-схеми для систем управління, які передбачають періодичне виконання з певною швидкістю у визначеному порядку. Поведінка скінченної автомати (Рис. 3.7) охоплює логіку прийняття рішень на високому рівні, яка активується різноманітними подіями як всередині, так і поза системою.

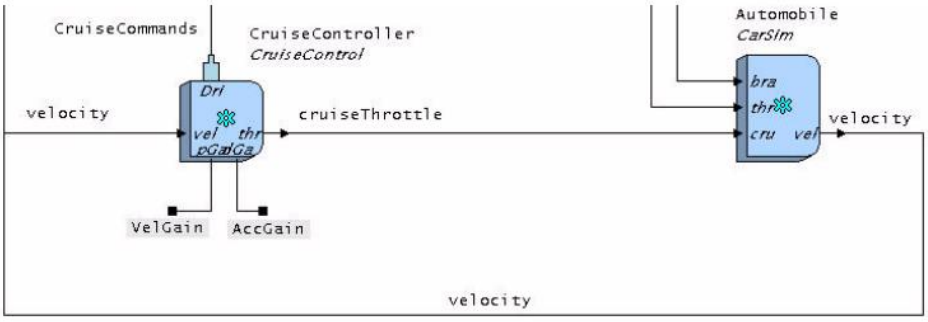


Рис. 3.6 Для керування транспортним засобом зі зворотним зв'язком потрібне виконання періодичного потоку вибіркових даних

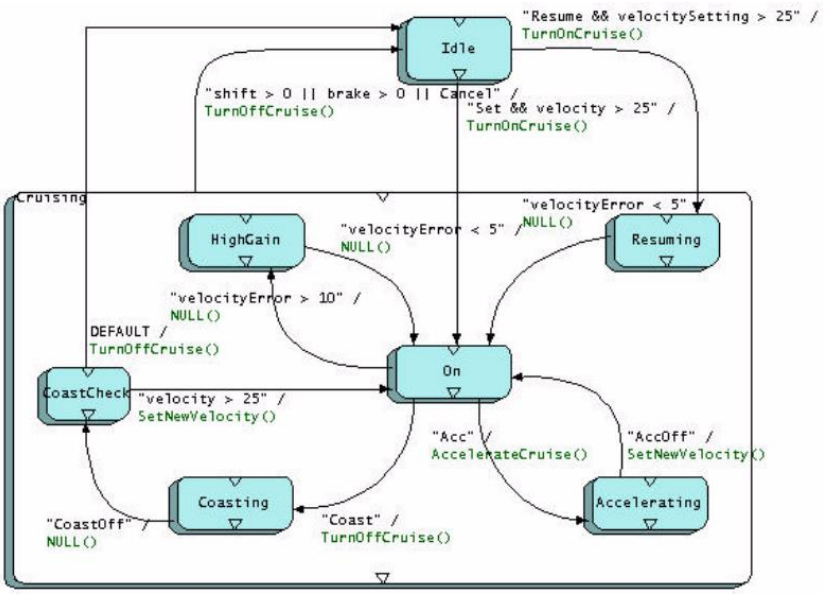
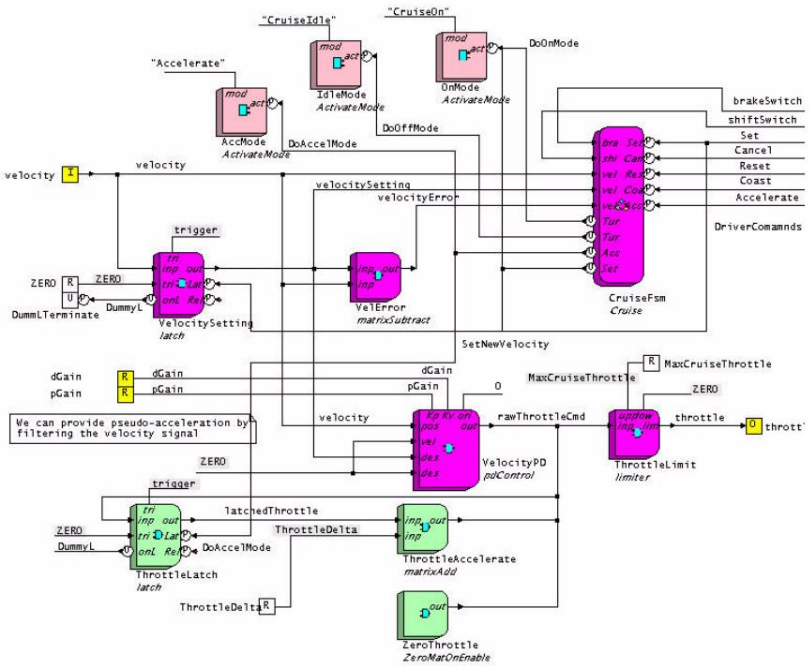


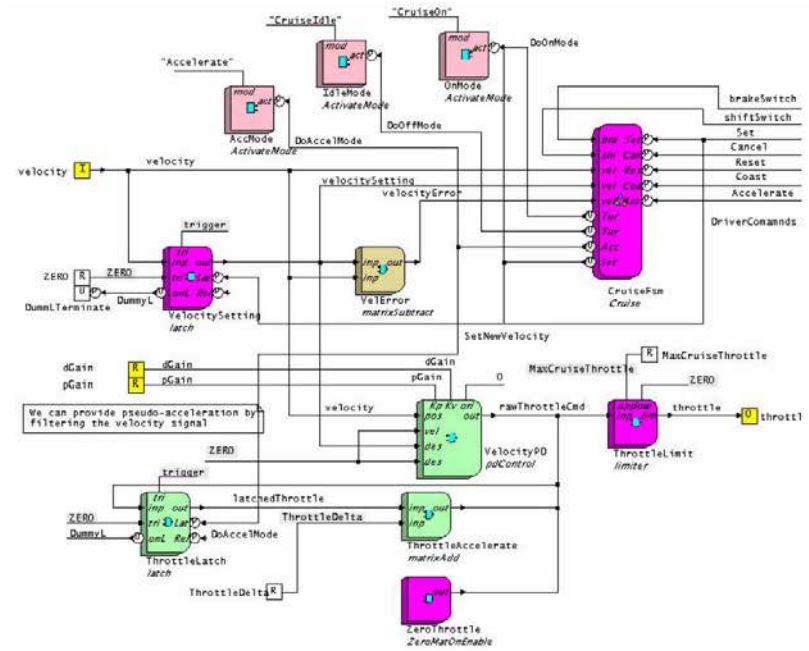
Рис. 3.7 Скінченний автомат, керований подіями, можна використовувати для перемикання між різними режимами контролера

Фреймворки, призначені для автономних систем, забезпечують вбудовані поведінки для вирішення специфічних вимог керування, які зазвичай зустрічаються в таких системах. Наприклад, програмний фреймворк може мати вбудовані можливості для створення модальних систем і перемикання певних компонентів в залежності від режиму роботи. Крім того,

у програмних фреймворках часто присутня наперед визначена модель потоків, яка підтримує різні періодичні частоти виконання потоку даних і інші моделі виконання задач.



(a) Круїз-контроль увімкнено



(b) Круїз-контроль вимкнено

Рис. 3.8 Перемикання режиму компонента круїз-контролера з увімкнення на вимкнення перемикає набір активних компонентів (виділено)

Переваги використання поведінки, специфічної до певної галузі, як частини фреймворку включають:

• Швидший розвиток: часто повторювані шаблони поведінки не потребують повторної реалізації.

• Зрозуміла поведінка: фреймворк надає шаблон поведінки, що забезпечує спільне розуміння серед розробників.

• Оптимізована поведінка: фреймворк може надавати оптимізовані реалізації повторюваних шаблонів поведінки.

ПЛАТФОРМИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОНОМНИХ СИСТЕМ ПРОГРАМНІ ФРЕЙМВОРКИ ТА ПЛАТФОРМИ

Програмний фреймворк — це система реалізації, яка заздалегідь визначає загальні поведінкові моделі для набору програмних додатків. Він спрощує розробку додатків, що використовують ці моделі. Фреймворк може бути представлений як вихідний код з документацією щодо його використання. Програмна платформа включає фреймворк і доповнюється інтегрованим набором інструментів для розробки та управління. Таким чином, як показано на Рисунку 3.9:

Platform = Framework + Integrated Tools

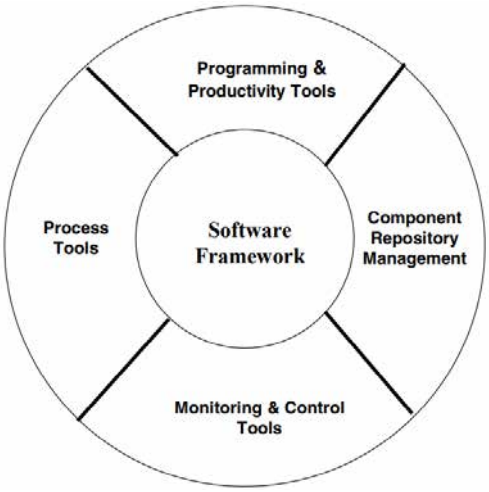


Рис. 3.9 Платформа програмного забезпечення включає структуру програмного забезпечення та інтегровані інструменти для розробки, розгортання, налагодження, моніторингу та контролю

МОЖЛИВОСТІ ІНТЕГРАЦІЇ

Програмні фреймворки здійснюють інтеграцію на всіх рівнях функціональності, що зустрічаються в програмному забезпеченні для автономних систем, надаючи архітектуру для об'єднання функцій різної деталізації. Це включає компоненти низького рівня, написані на C та C++, а також застарілий код, який необхідно інтегрувати у фреймворк. Також це охоплює компоненти високого рівня, розроблені за допомогою інструментів програмування на основі моделей, таких як MATLAB та Simulink. Програмний фреймворк може надавати підтримку для телеметрії та комунікацій шляхом вбудованих механізмів інтеграції зі стандартними середовищами сполучення, такими як CORBA16 та NDDS [32].

На рисунку 3.10 зображено систему круїз-контролю з рисунка 3.2, реалізовану за допомогою програмного фреймворку.

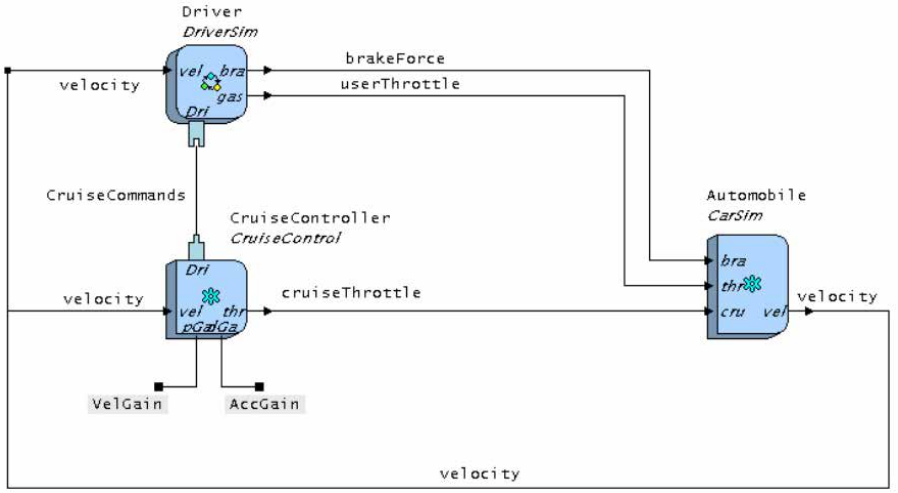


Рис. 3.10 Система круїз-контролю: круїз-контролер інтегровано з компонентами програмного забезпечення для автомобіля та водія

Автомобільний компонент може функціонувати як симуляція динаміки транспортного засобу або репрезентувати реальне автомобільне обладнання. Моделі симуляції динаміки транспортного засобу зазвичай розробляються з використанням MATLAB/Simulink. Реальні апаратні компоненти можуть включати драйвери пристроїв, написані на мовах програмування C та C++. Передача даних між компонентом Круїз-контролю і Автомобільним

компонентом може здійснюватися через локальну спільну пам'ять або мережу. Компонент Водія може слугувати інтерфейсом людина-машина (HMI), який симулює дії водія та можливо написаний на Java. Зв'язок між водієм і круїз-контролером може також реалізовуватись через мережу, наприклад, за допомогою CORBA у симуляційному середовищі. Програмна платформа здатна забезпечити інтеграцію різнорідних програмних компонентів, які можуть бути створені на різних рівнях абстракції, написані різними мовами програмування із застосуванням різних інструментів та іноді мають комунікувати через мережі, використовуючи різноманітні види проміжного програмного забезпечення [33].

Переваги інтеграційних можливостей вбудованих у фреймворк включають:

• Інтеграція навичок: члени розробницької команди можуть мати різні спеціалізації, але використовують єдиний програмний фреймворк.

• Автоматизація програмування інфраструктури: важлива для успіху проєкту, адже автоматизація підвищує загальну ефективність автономних систем і посилює здатність досягати цілей місії.

• Вища якість: фреймворк (особливо комерційно доступний) часто опробовується в різних проєктах і, зазвичай, проходить ретельне тестування з меншою ймовірністю наявності помилок. Також його можна незалежно перевірити.

Компоненти та інтерфейси можна створити один раз і використовувати багаторазово. Вони формують основу для тестування, верифікації, обміну та повторного використання програмного забезпечення автономних систем.

ІНСТРУМЕНТИ ПРОГРАМУВАННЯ ТА ПРОДУКТИВНОСТІ

Сучасні автономні системи створюються за швидким (спіральним) процесом розробки та зазвичай є міждисциплінарними проєктами. Програмна платформа може забезпечувати допоміжні засоби програмування

для підвищення продуктивності та підтримки різноманітних навичок і парадигм розробки. До цих засобів належать:

• Вищий рівень програмування: можливість визначення і складання компонентів за допомогою графічного редактора.

• Засоби аналізу і проєктування систем, включаючи інтеграцію з інструментами моделювання та розробки алгоритмів, такими як MATLAB і Simulink.

• Засоби аналізу, проєктування та документування програмного забезпечення з використанням Уніфікованої мови моделювання (UML).

• Засоби розробки вихідного коду, до яких входять редактори, компілятори, системи збирання та налагоджувачі.

• Автоматизація рутинних завдань: інструменти можуть виконувати звичні задачі самостійно.

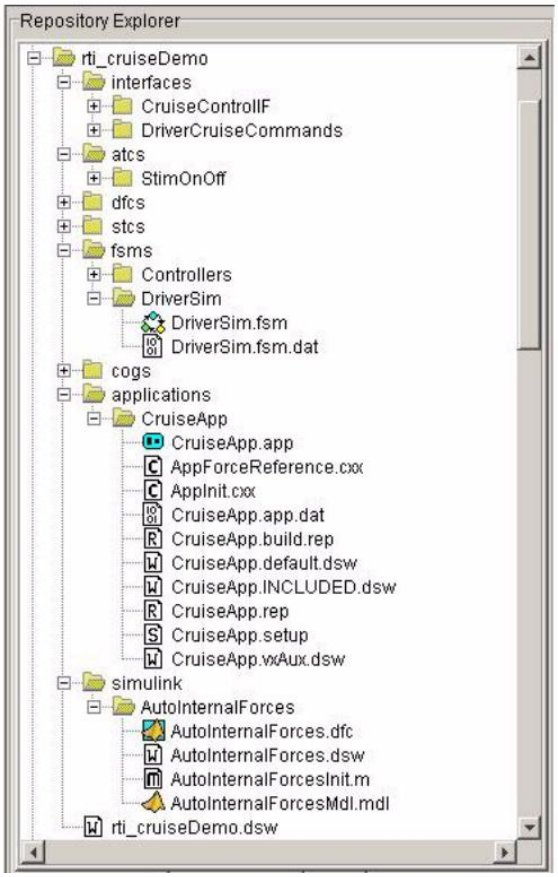


Рис. 3.11 Репозиторій програмних компонентів сприяє спільному використанню та повторному використанню

УПРАВЛІННЯ РЕПОЗИТОРІЄМ КОМПОНЕНТІВ

Зі зростанням кількості компонентів, розроблених на основі певної програмної платформи, виникає необхідність їх ефективного управління. Програмна платформа може забезпечити інфраструктуру для управління такими компонентами, що включає:

• Репозиторії для організації та каталогізації компонентів разом з їхньою документацією (див. рис. 3.11).

• Засоби для пошуку компонентів за ключовими словами або іншими критеріями.

• Метод організації спільного доступу і координації команд: програмні репозиторії компонентів можуть використовуватись різними командами.

• Індивідуальні палети для часто використовуваних компонентів.

• Засоби для підтримки еволюції компонентів, такі як управління версіями та поступове виведення з використання компонентів.

ІНСТРУМЕНТИ МОНІТОРИНГУ ТА КОНТРОЛЮ

Автономні системи працюють у режимі реального часу. Окрім користувацьких інтерфейсів і HMI, які є складовими частинами автономних систем, програмна платформа може надавати інструменти для моніторингу та контролю діючого застосунку. До таких інструментів можуть входити:

• Живий моніторинг і відображення сигналів даних. Якісний інструмент моніторингу дозволяє користувачеві редагувати сигнал даних усередині системи, що працює.

• Живий моніторинг і відображення станів. Наприклад, анімація скінченного автомату може показувати поточний стан системи.

• Навігація та контроль компонентів під час виконання. Це включає можливість напряму відправляти команди окремим компонентам, змінювати їх параметри тощо. Такі можливості можуть бути надзвичайно корисними під час діагностики та ремонту в польових умовах.

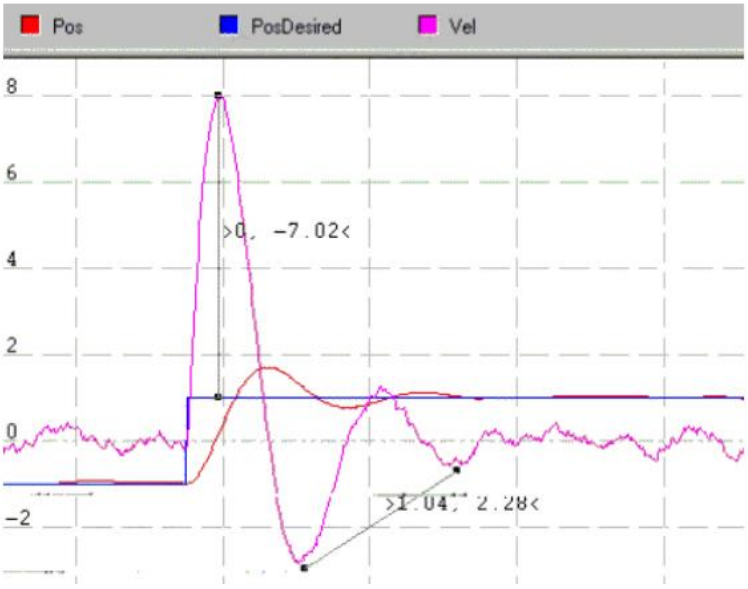


Рис. 3.12 Інструменти моніторингу в реальному часі для автономних систем можуть бути корисними під час розробки та після розгортання

3.2 Обгрунтування вибору засобів та технологій розробки

Обґрунтування вибору засобів та технологій для розробки програмного забезпечення системи управління автономними динамічними об'єктами полягає в необхідності створення надійної, адаптивної та масштабованої платформи, яка здатна працювати в умовах реального часу. Вибір засобів і технологій повинен відповідати кільком ключовим вимогам: обробка великих обсягів даних, швидкість обчислень, здатність до інтеграції з апаратними компонентами, а також підтримка алгоритмів машинного навчання для самонавчання системи.

Основою для розробки таких систем є мови програмування з високою продуктивністю та підтримкою багатозадачності, як-от C++ чи Python. C++ забезпечує швидкість і низькорівневий доступ до апаратури, що є критично важливим для роботи з сенсорами і контролерами в режимі реального часу. Водночас, Python використовується для розробки алгоритмів машинного

навчання та обробки великих масивів даних, що дозволяє адаптувати поведінку системи до нових умов і покращувати її на основі досвіду [34].

Важливою частиною розробки є використання бібліотек і фреймворків для комп'ютерного зору та обробки даних сенсорів, таких як OpenCV для роботи із зображеннями та відеопотоками, а також ROS (Robot Operating System) для забезпечення інтеграції між різними модулями та координації процесів автономного управління. ROS надає інструменти для модульної архітектури, що дозволяє розподіляти завдання управління між окремими блоками, забезпечуючи їх взаємодію і синхронізацію.

Для алгоритмів машинного навчання та штучного інтелекту широко застосовуються такі платформи, як TensorFlow або PyTorch, які дозволяють навчати моделі для прогнозування поведінки об'єктів, аналізу динаміки середовища та адаптивного управління. Ці фреймворки підтримують глибоке навчання та можуть бути інтегровані з реальними сенсорними даними для постійного вдосконалення системи в умовах експлуатації.

Крім того, для забезпечення високої продуктивності та ефективності роботи на різних платформах, включаючи вбудовані системи та хмарні сервіси, використовується технологія контейнеризації за допомогою Docker. Це дозволяє створювати портативні та легко масштабовані рішення, що можуть бути швидко розгорнуті на різних пристроях або інфраструктурах.

Таким чином, вибір засобів і технологій обумовлений необхідністю забезпечити швидку обробку даних, високу надійність, можливість інтеграції різних компонентів системи та підтримку алгоритмів самонавчання. Це дає змогу створити ефективне програмне забезпечення, здатне керувати автономними динамічними об'єктами в складних і змінних середовищах.

3.2.1 Вибір технологій програмування

Вибір технологій програмування для розробки програмного забезпечення системи управління автономними динамічними об'єктами

ґрунтується на ключових вимогах до продуктивності, адаптивності та інтеграції з апаратними компонентами. Програмне забезпечення повинне забезпечувати швидку обробку великих обсягів даних у реальному часі, що виникає при взаємодії з сенсорами, а також прийняття управлінських рішень на основі аналізу цих даних.

Вибір мов програмування та фреймворків напряму залежить від необхідності поєднання низькорівневого контролю за обладнанням із високим рівнем обробки даних і штучного інтелекту. Для обробки сигналів і керування об'єктами потрібен доступ до апаратури, що дозволяє керувати процесами з мінімальними затримками. Одночасно для обробки великого масиву сенсорних даних і реалізації алгоритмів комп'ютерного зору та машинного навчання важливо забезпечити швидкий доступ до бібліотек, що полегшують побудову моделей і алгоритмів [35].

Для досягнення цієї мети необхідно використовувати засоби, які підтримують багатопоточність та паралельні обчислення, що дає змогу системі одночасно обробляти кілька потоків даних і виконувати складні обчислювальні завдання без шкоди для продуктивності. Крім того, технології повинні забезпечувати легку інтеграцію з апаратними компонентами автономного об'єкта, такими як контролери, сенсори та системи навігації.

Іншим важливим аспектом є вибір технологій для реалізації систем адаптивного навчання. Це дозволяє системі покращувати свої рішення та адаптувати поведінку на основі досвіду, що є критичним для автономних об'єктів, які працюють у складних та мінливих умовах. Тут важливо використовувати інструменти, що підтримують розробку алгоритмів глибокого навчання та дозволяють працювати з великими обсягами даних для їх швидкої обробки та аналізу [36].

Таким чином, вибір технологій програмування орієнтований на забезпечення продуктивності, надійності та адаптивності системи, яка може

функціонувати у реальному часі та забезпечувати точне управління автономними динамічними об'єктами.

Для розробки програмного забезпечення системи управління автономними динамічними об'єктами використовуються різні технології програмування, які відповідають вимогам високої продуктивності, обробки даних у реальному часі, інтеграції з апаратними компонентами та підтримки машинного навчання.

1. **C++** є однією з ключових технологій, яка використовується для низькорівневого управління апаратурою та роботи в режимі реального часу. Вона забезпечує ефективне управління пам’яттю та високу продуктивність, що важливо для роботи з сенсорами та іншими фізичними компонентами автономної системи. C++ також широко застосовується для розробки алгоритмів контролю руху, навігації та стабілізації.
2. **Python** є ще однією популярною технологією, особливо для розробки алгоритмів машинного навчання, комп’ютерного зору та обробки великих даних. Він має велику кількість бібліотек для реалізації штучного інтелекту та машинного навчання, таких як TensorFlow, PyTorch і OpenCV. Python також використовується для швидкого прототипування та побудови моделей, що дозволяє реалізовувати адаптивні системи управління, які постійно вдосконалюються на основі нових даних.
3. **ROS (Robot Operating System)** — це популярна фреймворк-платформа для створення програмного забезпечення роботів, включаючи автономні системи. ROS забезпечує модульну архітектуру, що дозволяє розробляти окремі компоненти управління, навігації, сенсорної обробки та взаємодії між ними. Завдяки ROS можлива інтеграція різних компонентів системи в єдину керовану структуру, яка легко налаштовується і підтримує роботу в реальному часі.

4. **TensorFlow та PyTorch** використовуються для створення алгоритмів глибокого навчання та машинного зору, які дозволяють автономним об'єктам вчитися на основі даних, що збираються в процесі експлуатації. Ці платформи забезпечують високу гнучкість у побудові нейронних мереж і дозволяють масштабувати моделі на різних платформах.
5. **OpenCV** є бібліотекою комп'ютерного зору, яка забезпечує обробку зображень і відеопотоків у реальному часі. Вона важлива для автономних систем, які використовують камери та інші сенсори для аналізу оточення, розпізнавання об’єктів і ухвалення рішень на основі візуальних даних.
6. **Docker** і технології контейнеризації забезпечують розгортання програмного забезпечення на різних платформах без необхідності модифікації системних компонентів. Це дозволяє забезпечити стабільність роботи в різних середовищах, полегшує оновлення та масштабування програмного забезпечення.

Ці технології програмування забезпечують можливість створення ефективних, надійних і гнучких систем управління автономними динамічними об'єктами, які можуть працювати в умовах невизначеності та складного середовища [37].

3.2.2 Вибір мови програмування

Вибір мови програмування для розробки програмного забезпечення системи управління автономними динамічними об'єктами має вирішальне значення, оскільки від цього залежить продуктивність, швидкість обробки даних, ефективність управління апаратними компонентами та здатність працювати в реальному часі. Ось ключові аспекти, які впливають на вибір мови програмування для таких систем:

1. C++

C++ є одним із найбільш популярних виборів для розробки систем управління автономними об'єктами через його високу продуктивність і низькорівневий доступ до апаратури. C++ дозволяє ефективно працювати з великими обсягами даних і забезпечує мінімальні затримки в обробці, що критично важливо для управління реальними об'єктами в режимі реального часу. Ця мова особливо підходить для систем, де необхідно працювати з обмеженими ресурсами (наприклад, у вбудованих системах), оскільки вона дозволяє точний контроль за пам'яттю та процесорними ресурсами [38].

Крім того, C++ широко використовується в робототехніці та управлінні рухом завдяки своїм бібліотекам для роботи з сенсорами та контролерами, які інтегруються з фізичними компонентами автономних об'єктів.

2. Python

Python також є важливим вибором, особливо в контексті розробки алгоритмів штучного інтелекту та машинного навчання, що часто використовуються в автономних системах для ухвалення рішень та адаптації до змінних умов середовища. Python має багату екосистему бібліотек для обробки даних, таких як TensorFlow, PyTorch і OpenCV, які спрощують створення алгоритмів комп'ютерного зору, аналізу даних і машинного навчання [39].

Однак Python не є такою продуктивною мовою, як C++, коли мова йде про обчислювальні задачі в реальному часі. Тому він часто використовується для розробки прототипів і машинного навчання, у той час як критично важливі компоненти системи, які потребують високої швидкості обробки даних, пишуться на C++.

3. Інтеграція C++ та Python

У багатьох випадках ефективне поєднання обох мов може забезпечити максимальну продуктивність і гнучкість. C++ можна використовувати для критичних підсистем, таких як управління рухом і робота з апаратними

засобами, тоді як Python застосовується для високорівневих алгоритмів машинного навчання, аналітики та обробки даних. Це дозволяє об’єднати переваги обох мов: високу продуктивність C++ та гнучкість Python для швидкої розробки і навчання системи.

C++ є оптимальним вибором для низькорівневого управління автономними динамічними об'єктами, де важлива швидкість обробки та доступ до апаратного забезпечення. Python підходить для розробки високорівневих компонентів, таких як алгоритми штучного інтелекту та машинного навчання. Поєднання цих двох мов дозволяє створити збалансовану систему, яка забезпечує як швидкість, так і гнучкість у розробці.

Проте ми з вами зупинемося на HTML, бо вона є відмінним вибором для розробки інтерфейсу програмного забезпечення системи управління автономними динамічними об'єктами завдяки кільком важливим перевагам:

Кросплатформеність

HTML є основою веб-технологій, які підтримуються всіма сучасними браузерами та операційними системами. Це означає, що інтерфейс, створений на базі HTML, буде працювати як на настільних комп’ютерах, так і на мобільних пристроях без необхідності адаптації коду під різні платформи. Це особливо важливо для автономних систем, які можуть бути керовані з різних пристроїв у різних середовищах (наприклад, оператор може використовувати планшет або ноутбук) [40].

Швидка розробка та тестування

HTML у поєднанні з CSS і JavaScript дозволяє швидко створювати прототипи інтерфейсів та тестувати їх у реальному часі. Це зменшує час на розробку, оскільки можна швидко перевірити, як інтерфейс виглядає і працює в браузері. Крім того, браузери надають інструменти для налагодження, що спрощує процес виправлення помилок і налаштування візуального вигляду.

Легка інтеграція з серверними та хмарними рішеннями

HTML інтерфейси можуть бути легко інтегровані з серверними технологіями через веб-протоколи (HTTP/HTTPS), що дозволяє отримувати й обробляти дані від сенсорів, систем моніторингу або хмарних серверів у реальному часі. Це забезпечує гнучке управління та контроль за автономними об'єктами незалежно від місця розташування оператора або об'єкта.

Масштабованість та адаптивність

HTML дозволяє створювати інтерфейси, які легко масштабуються під різні роздільні здатності екранів і пристроїв. Це особливо важливо для операційних інтерфейсів, де можуть використовуватися як великі дисплеї для моніторингу в командних центрах, так і компактні екрани мобільних пристроїв.

Широка екосистема та підтримка бібліотек

HTML підтримується величезною кількістю бібліотек і фреймворків (React, Angular, Vue.js), що дозволяє розробляти складні інтерфейси з інтерактивними можливостями. Для автономних систем це може включати динамічну візуалізацію даних, картографічні сервіси, графіки та панелі моніторингу в реальному часі.

Інтерактивність і користувацький досвід

Завдяки використанню JavaScript, HTML інтерфейси можуть бути легко зроблені інтерактивними — реагувати на дії користувача, оновлювати дані без перезавантаження сторінки, забезпечуючи плавний і ефективний досвід взаємодії. Це критично важливо для автономних систем, де операторам потрібен швидкий доступ до поточних даних і можливість негайно вносити зміни в налаштування або керування [41].

Веб-технології для реального часу

HTML у поєднанні з веб-технологіями, такими як WebSockets, дозволяє реалізувати передачу даних у реальному часі між автономним об'єктом і

інтерфейсом. Це особливо важливо для систем, де потрібне постійне оновлення даних (швидкість, сенсори, перешкоди) та негайне відображення інформації оператору.

3.2.3 Вибір середовища розробки

У розробці програмного забезпечення є чотири типи середовищ:

- Development (dev) - середовище для розробки
- Testing (test) - середовище для тестування
- Staging (stage) - середовище для тестування
- Production (prod) - середовище розгортання

Розглянемо детальніше різні етапи середовища розробки та впровадження програмного забезпечення, щоб з’ясувати особливості їх функціонування.

Середовище розробки, відоме як Development, є місцем, де програмісти здійснюють написання та оновлення коду. Зазвичай воно охоплює один або декілька серверів, що знаходяться у спільному користуванні декількох розробників, залучених до одного проекту. В умовах колективної роботи кожен розробник зберігає локальну копію вихідного коду на своєму комп'ютері, який згодом інтегрується у загальний репозиторій (зазвичай у вигляді однієї гілки). Крім того, на цьому етапі здійснюється тестування коду самими розробниками [42].

Testing середовище слугує платформою для інженерів контролю якості (QA) з метою випробування новітнього або модифікованого коду через автоматизовані або неавтоматизовані методи. Основна увага тут приділяється верифікації окремих модулів та їх сумісності, тобто перевіряється взаємодія нового коду із раніше створеними компонентами.

Середовище Staging передбачає тестування програмного забезпечення в умовах, максимально наближених до реальних, для забезпечення його коректного функціонування після впровадження до користувача. Тестувальники використовують усі необхідні методи тестів, щоб ретельно перевірити програмний продукт. Цей етап доступний для розробників, тестувальників, а за потреби і для замовника та інших зацікавлених сторін.

Нарешті, Production - це середовище, яке забезпечує офіційне розміщення програмного забезпечення для кінцевих користувачів.

Вибір середовища розробки для програмного забезпечення системи управління автономними динамічними об'єктами є критичним етапом, оскільки від нього залежить ефективність робочого процесу, інтеграція з апаратними компонентами та можливість реалізації алгоритмів у реальному часі. Головними критеріями при виборі середовища є його здатність підтримувати багатозадачність, надавати потужні інструменти для роботи з великою кількістю даних, а також забезпечувати інтеграцію з необхідними бібліотеками та фреймворками для робототехніки і штучного інтелекту [43].

Середовище розробки повинне не лише полегшувати написання коду, але й забезпечувати швидке тестування, налагодження та інтеграцію різних модулів, оскільки система управління автономними об'єктами складається з багатьох компонентів. Це можуть бути модулі навігації, контролю руху, обробки сенсорних даних та машинного навчання, кожен з яких виконує критично важливі завдання. Тому важливо, щоб середовище дозволяло легко керувати проєктом і тестувати його на різних платформах чи вбудованих системах, не втрачаючи продуктивності.

Крім того, у процесі розробки автономних систем необхідна тісна інтеграція з апаратними пристроями, такими як сенсори, камери, лідари та контролери. Середовище розробки має підтримувати нативний доступ до таких пристроїв, а також забезпечувати гнучке налаштування під різні операційні системи і апаратні платформи. Це вимагає потужних засобів для

роботи з різними API та драйверами, а також можливості налагодження в реальному часі.

Іншим важливим аспектом є підтримка паралельної розробки та версійного контролю, що дає змогу кільком розробникам одночасно працювати над різними частинами системи. Це вимагає інтеграції з системами контролю версій, що дозволяє синхронізувати зміни та ефективно керувати великими командами розробників.

Вибір середовища розробки для створення програмного забезпечення системи управління автономними динамічними об'єктами повинен ґрунтуватися на його здатності забезпечити продуктивну розробку, гнучку інтеграцію з апаратним забезпеченням і підтримку потужних інструментів для налагодження та тестування в реальному часі.

3.2.4 Вибір типу сховища даних, що застосовується у проекті

Вибір типу сховища даних для проекту, що стосується управління автономними динамічними об'єктами, є критичним рішенням, оскільки воно напряму впливає на продуктивність системи, швидкість доступу до даних та можливості масштабування. Важливо забезпечити ефективне зберігання, обробку й аналіз даних, які надходять від різних сенсорів та інших модулів системи в режимі реального часу.

Система управління автономними об’єктами часто генерує величезну кількість різнорідних даних — це можуть бути зображення, потоки відео з камер, дані з лідарів, радарів, а також телеметрія з датчиків руху. Дані потребують не лише збереження, а й швидкого доступу, оскільки рішення щодо управління об’єктом повинні прийматися миттєво, враховуючи поточну ситуацію. Вибір типу сховища даних повинен враховувати ці вимоги, тому сховище має бути оптимізованим для швидкого читання та запису [44].

Одним із головних аспектів є підтримка масштабованості. У разі розширення системи або збільшення кількості даних, сховище має забезпечувати можливість легкої адаптації до зростання обсягів інформації. Це стосується як горизонтальної масштабованості, коли додаткові ресурси додаються в систему, так і вертикальної, коли збільшується потужність окремого сховища. Оскільки автономні об’єкти часто працюють у середовищах з високою інтенсивністю даних (наприклад, в умовах міста), важливо, щоб система могла обробляти ці дані без затримок.

Крім того, вибір сховища даних повинен враховувати підтримку різних типів даних — структурованих, напівструктурованих і неструктурованих. Це може бути поєднання реляційних баз даних для телеметричних даних і нереляційних (NoSQL) систем для зберігання великих обсягів сенсорної інформації або потоків відео. Така гнучкість дозволяє ефективніше управляти різними типами даних і забезпечувати їх швидку обробку.

Нарешті, безпека даних також є важливим фактором. Оскільки автономні системи можуть працювати в умовах високої ризикованості (наприклад, у транспортних системах), сховище даних повинно забезпечувати надійне збереження і захист даних від втрат, збоїв або несанкціонованого доступу. Це означає, що обрана технологія сховища повинна підтримувати стійкість до збоїв і забезпечувати резервне копіювання та відновлення даних у разі необхідності [45].

Вибір типу сховища даних у такому проекті ґрунтується на необхідності балансування між швидкістю доступу, гнучкістю обробки різних типів даних, масштабованістю та забезпеченням високої надійності і безпеки.

Для розробки програмного забезпечення системи управління автономними динамічними об'єктами актуальним буде вибір **гібридного типу сховища даних**, який поєднує реляційні та нереляційні (NoSQL) бази даних. Це дозволить ефективно працювати з різними типами даних, які

генерує автономна система, і задовольнити вимоги до швидкості обробки та масштабованості.

Чому гібридний тип сховища найбільш підходящий?

1. **Реляційні бази даних** підходять для зберігання структурованих даних, таких як телеметрія або логування системних подій, де важлива цілісність даних і підтримка транзакцій. Це дозволяє зберігати детальні дані про роботу системи, параметри її компонентів, виконувані команди та результати.
2. **Нереляційні (NoSQL) бази даних** краще підходять для роботи з великими обсягами неструктурованих або напівструктурованих даних, таких як дані з сенсорів, лідарів, радарів або камер. Наприклад, у реальному часі можуть надходити великі потоки відео та зображень, які вимагають швидкої обробки та доступу. NoSQL сховища, такі як MongoDB або Cassandra, можуть забезпечити горизонтальну масштабованість і високу швидкість запису та читання таких даних.

Вибір гібридної архітектури дозволяє:

- **Швидко обробляти та зберігати різні типи даних** — від структурованих параметрів системи до неструктурованих потоків з сенсорів.
- **Масштабувати систему** відповідно до зростання кількості даних. Наприклад, при збільшенні кількості об'єктів або складності середовища (міська інфраструктура, високий трафік).
- **Оптимізувати ресурси** для роботи в реальному часі, дозволяючи одночасно обробляти як критичні дані про стан системи, так і великі обсяги сенсорної інформації.
- **Гнучко керувати різними сценаріями** роботи системи, адаптуючись до різних середовищ — від автомобільної інфраструктури до промислових або логістичних сценаріїв.

Таким чином, гібридний підхід до вибору сховища даних є найбільш доцільним, оскільки він забезпечує баланс між продуктивністю, гнучкістю і можливістю масштабування, що особливо важливо для автономних динамічних систем.

3.3 Розробка прототипу інтерфейсу програмного забезпечення, що розробляється

Розробка прототипу інтерфейсу програмного забезпечення для управління автономними динамічними об'єктами повинна враховувати кілька ключових аспектів: інтуїтивність, оперативність у прийнятті рішень та надання максимальної інформації для контролю й моніторингу системи. Основна мета інтерфейсу — забезпечити ефективну взаємодію оператора з системою та можливість контролю за роботою автономного об'єкта в реальному часі.

Опис прототипу інтерфейсу:

Прототип інтерфейсу повинен відображати стан об'єкта та навколишнє середовище через інтерактивні візуалізації. Наприклад, для безпілотного автомобіля це може бути інтерактивна карта, яка оновлюється в режимі реального часу, відображаючи місцезнаходження автомобіля, його траєкторію, дані з сенсорів (виявлення перешкод, інших транспортних засобів), а також інформацію про внутрішні параметри — швидкість, рівень заряду, температуру двигуна тощо.

Центральна частина інтерфейсу — це **динамічний екран моніторингу**, що показує поточний стан системи в реальному часі. На екрані повинні бути розміщені кілька важливих модулів:

- **Візуалізація середовища** (карта з відображенням об'єктів, трасування руху, перешкоди).
- **Системний стан** (діагностичні дані, поточні параметри об'єкта, статус систем управління).

- **Аварійні повідомлення** або критичні сповіщення в разі виникнення несправностей або відхилень від норми.

Інтерактивна панель дозволить оператору легко коригувати деякі параметри або налаштування системи. Для складних задач, таких як перемикання між автономними та ручними режимами, повинна бути передбачена чітка і зрозуміла навігація між різними рівнями керування. Наприклад, перехід від загального огляду системи до детального моніторингу сенсорних даних або контроль за роботою окремих підсистем [46].

Інтерфейс також повинен мати можливості для **настроювання конфігурацій**, що дозволить оператору задавати певні параметри роботи системи або контролювати реакції автономного об'єкта в певних ситуаціях (наприклад, налаштування чутливості до перешкод чи встановлення швидкісних обмежень).

Для підвищення ефективності управління, інтерфейс може містити **інтерактивні графіки та аналітичні панелі**, які візуалізують важливі тренди, такі як витрати енергії, швидкість, кількість перешкод тощо. Це допоможе оператору приймати більш обґрунтовані рішення на основі довгострокових даних.

Прототипування інтерфейсу:

На етапі прототипування слід використовувати інструменти для створення інтерактивних макетів (наприклад, Figma або Adobe XD). Це дозволяє моделювати роботу інтерфейсу, перевіряти його зручність та ефективність перед початком розробки. Важливо також тестувати інтерфейс з різними користувачами (операторами) для отримання зворотного зв'язку щодо його функціональності та зручності. Загалом, інтерфейс повинен забезпечувати доступ до всієї необхідної інформації з мінімальними зусиллями для користувача, бути інтуїтивно зрозумілим та адаптивним для різних сценаріїв використання [47].

Приклад того, як шаблон прототипу можна реалізувати в HTML (з повним кодом можна ознайомитися в кінці роботи у додатку на ст. 147):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Autonomous Object Control Interface</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Control Panel for Autonomous Dynamic Object</h1>
    </header>

    <div class="container">
        <!-- Map and central display -->
        <div class="map-container">
            <h2>Real-Time Map</h2>
            <div id="map">
                <!-- Map visualization here -->
                <p>Map is loading...</p>
            </div>
        </div>

        <!-- Sensor Data Panel -->
        <div class="sensor-panel">
            <h2>Sensor Data</h2>
            <ul>
```

```
            <li>Speed: <span id="speed">45 km/h</span></li>
            <li>Battery: <span id="battery">80%</span></li>
            <li>Diagnostics: <span id="diagnostics">All systems operational</span></li>
        </ul>
    </div>

    <!-- Control Panel -->
    <div class="control-panel">
        <h2>Control Panel</h2>
        <button onclick="switchToManual()">Switch to Manual Mode</button>
        <button onclick="switchToAutonomous()">Switch to Autonomous Mode</button>
        <h3>Configuration</h3>
        <label for="sensitivity">Obstacle Sensitivity:</label>
        <input type="range" id="sensitivity" name="sensitivity" min="1" max="10">
    </div>
</div>

<!-- Emergency Alerts -->
<div class="alerts">
    <h2>Emergency Alerts</h2>
    <p id="alert">No critical issues detected.</p>
</div>

<script src="script.js"></script>
</body>
</html>
```

А ось який вигляд має прототип інтерфейсу програмного забезпечення системи управління автономними динамічними об'єктами

Рис. 3.13 Приклад інтерфейсу програмного забезпечення системи управління автономними динамічними об'єктами

3.4 Опис реалізації окремих алгоритмічних особливостей програмного забезпечення

Реалізація алгоритмічних особливостей програмного забезпечення для управління автономними динамічними об'єктами базується на складних обчислювальних процесах, які забезпечують адаптивність системи та її здатність працювати в режимі реального часу. Ключовим аспектом такої реалізації є здатність програмного забезпечення аналізувати вхідні дані від різних сенсорів, будувати прогнози на основі цих даних та приймати рішення щодо подальших дій об'єкта. Ось як це відбувається на практиці.

Одним із важливих алгоритмів є обробка даних від сенсорів і побудова віртуальної моделі оточення в реальному часі. Програмне забезпечення отримує безперервний потік даних із лідарів, камер, радарів та інших сенсорів, що відображають положення об'єкта у просторі та потенційні перешкоди. На основі цих даних система будує тривимірну модель

навколишнього середовища, де кожен об'єкт (наприклад, інші транспортні засоби або статичні перешкоди) отримує свою координату та швидкість руху.

Далі в цій моделі реалізується алгоритм прогнозування траєкторій об'єктів. Це необхідно для того, щоб автономна система могла не лише реагувати на поточну ситуацію, але й прораховувати майбутні дії об'єктів у просторі. Наприклад, якщо інший автомобіль рухається назустріч, система повинна вміти передбачати його траєкторію та коригувати свою власну, уникаючи зіткнень. Для цього використовується алгоритм, який на основі поточних швидкостей і напрямів руху обчислює ймовірні позиції об'єктів через певні проміжки часу [48].

Важливою частиною алгоритму є прийняття рішень щодо змін траєкторії об'єкта. Програмне забезпечення використовує методи оптимізації для вибору найкращої траєкторії з урахуванням кількох критеріїв: безпека руху, мінімізація енергоспоживання, виконання поставленої задачі (наприклад, досягнення цілі в мінімальний час). Цей процес включає динамічний вибір маршруту залежно від змін у середовищі. Для забезпечення такої гнучкості використовується алгоритм планування руху на основі графів або методи пошуку, такі як A* або Dijkstra, що допомагають знайти найкоротший або найефективніший маршрут.

Окрім цього, автономні системи часто використовують алгоритми машинного навчання для адаптації до нових ситуацій. Наприклад, алгоритм контролю може навчатися на основі попередніх рішень та умов середовища, удосконалюючи свою здатність реагувати на нестандартні ситуації. Це дозволяє програмному забезпеченню не просто реагувати на конкретні події, а й вивчати загальні закономірності та прогнозувати, як змінюватимуться умови в майбутньому.

У підсумку, програмне забезпечення реалізує інтеграцію різних алгоритмів – від обробки сенсорних даних і побудови моделей середовища до планування маршруту та прийняття рішень на основі прогнозів і навчання.

Така багаторівнева архітектура дозволяє автономній системі діяти адаптивно, швидко і точно в динамічних середовищах.

В процесі розробки програмного забезпечення для управління автономними динамічними об'єктами ключову роль відіграють додаткові алгоритмічні особливості, які дозволяють системі працювати з високою надійністю, швидкістю реакції та адаптивністю до змінних умов. Ось декілька важливих аспектів, які ще не були згадані, але є критично важливими для таких систем [49]:

1. Фільтрація шуму та обробка неповних даних

У реальних умовах, дані від сенсорів часто можуть містити шум або бути неповними через зовнішні чинники (наприклад, погодні умови або технічні збої). Алгоритми обробки повинні вміти відфільтровувати ці небажані дані та відновлювати інформацію, що втрачена. Для цього широко використовуються методи, такі як фільтр Калмана, який дозволяє оцінювати стан системи на основі неповних або зашумлених даних, поступово поліпшуючи точність на основі попередніх оцінок.

Фільтр Калмана створює прогноз про стан об'єкта, базуючись на математичній моделі руху, і поступово коригує цей прогноз на основі отриманих даних. У випадку управління автономним об'єктом, це означає, що навіть якщо сенсори тимчасово передають неточну інформацію (наприклад, про швидкість або відстань до перешкоди), система може підтримувати стабільність управління.

2. Алгоритми прогнозування поведінки

Щоб уникнути зіткнень і забезпечити безпеку, автономні системи повинні не лише аналізувати поточну ситуацію, але й передбачати поведінку інших об'єктів у просторі. Для цього використовуються алгоритми прогнозування, які на основі аналізу попередніх траєкторій руху та швидкостей об'єктів визначають їх можливі майбутні положення.

Наприклад, система може використовувати параметричні моделі, які оцінюють траєкторії об'єктів на основі попередніх патернів. Це дозволяє спрогнозувати, куди рухатиметься автомобіль або пішохід через кілька секунд, і заздалегідь скоригувати власний маршрут автономного об'єкта, щоб уникнути зіткнення.

3. Обробка великих даних та децентралізовані системи

У випадку, коли автономні об'єкти функціонують у складі більшої системи (наприклад, у місті з іншими безпілотними автомобілями або дронами), важливою стає здатність обробляти великі обсяги даних від інших об'єктів. Це може вимагати використання технологій розподілених обчислень та алгоритмів, що оптимізують координацію між кількома автономними об'єктами.

Децентралізовані алгоритми управління дозволяють кожному об'єкту приймати рішення на основі локальних даних, отриманих від його сенсорів, і взаємодії з іншими об'єктами через обмін інформацією. Наприклад, безпілотні автомобілі можуть обмінюватися даними про дорожню ситуацію, трафік і перешкоди для узгодження своїх дій.

4. Алгоритми самонавчання та адаптації

Здатність системи вчитися на власному досвіді є ще одним критично важливим аспектом автономних систем. Для цього використовуються алгоритми машинного навчання, які дозволяють об'єкту адаптувати свою поведінку до нових умов, покращуючи ефективність управління з часом.

Наприклад, у системах управління автономними автомобілями використовуються глибокі нейронні мережі для розпізнавання об'єктів на дорозі, ідентифікації пішоходів, визначення дорожніх знаків та перешкод. З кожним новим сеансом керування система може удосконалювати свої моделі, враховуючи раніше невідомі фактори, такі як складні погодні умови, нові типи перешкод або непередбачувана поведінка інших учасників руху.

5. Алгоритми ухвалення рішень в екстрених ситуаціях

Важливим елементом є здатність системи швидко ухвалювати рішення в критичних ситуаціях, коли стандартні методи можуть не спрацювати. Для цього використовуються евристичні алгоритми, які дозволяють швидко приймати рішення на основі мінімальної кількості доступних даних.

Наприклад, коли система виявляє несподівану перешкоду на дорозі або різку зміну дорожньої ситуації, вона повинна миттєво ухвалити рішення про екстрене гальмування або зміну траєкторії. Алгоритми, які застосовуються в таких ситуаціях, повинні не тільки бути швидкими, але й гарантувати безпеку та мінімізацію шкоди [50].

3.5 Висновки по розділу

У висновку, розробка програмного забезпечення для управління автономними динамічними об'єктами є складним і багатоаспектним процесом, що охоплює численні алгоритмічні особливості та технологічні рішення. Система повинна обробляти великі обсяги даних від сенсорів у реальному часі, працювати в умовах невизначеності та адаптуватися до динамічних змін у середовищі. Ключові алгоритмічні компоненти включають фільтрацію шуму, прогнозування поведінки об'єктів, планування траєкторій, швидке ухвалення рішень у критичних ситуаціях і самонавчання на основі накопиченого досвіду.

Вибір технологій, таких як HTML у поєднанні з JavaScript для інтерфейсів, або використання C++ та Python для обробки великих даних і машинного навчання, дозволяє створити гнучкі та надійні системи, здатні ефективно функціонувати на різних платформах і забезпечувати інтеграцію з апаратними засобами. Програмне забезпечення має бути модульним, масштабованим і адаптованим до різних умов експлуатації.

Важливим є також поєднання низькорівневих обчислень для швидкого контролю об'єкта з високорівневими алгоритмами аналізу даних і ухвалення рішень. Це забезпечує баланс між продуктивністю і гнучкістю системи, що

дозволяє автономним об'єктам функціонувати безпечно і точно у складних сценаріях, таких як транспортні системи або промислові застосування.

4 Результати дослідження програмного забезпечення системи управління автономними динамічними об'єктами

4.1 Оцінка ефективності розробленого програмного забезпечення

Результати дослідження програмного забезпечення системи управління автономними динамічними об'єктами демонструють, що такі системи досягають високої ефективності завдяки комплексній взаємодії алгоритмів обробки даних, ухвалення рішень і машинного навчання. Одним із ключових результатів є те, що здатність системи швидко адаптуватися до змінних умов середовища залежить від інтеграції передових алгоритмів прогнозування поведінки об'єктів і точного фільтрування шумів у даних від сенсорів.

Завдяки використанню моделей штучного інтелекту, система може постійно покращувати свою роботу, навчаючись на попередньому досвіді та коригуючи поведінку відповідно до нових умов. Це дозволяє не тільки реагувати на ситуації в режимі реального часу, а й передбачати потенційні ризики на кілька кроків уперед, що підвищує загальний рівень безпеки та ефективності. Дослідження показують, що такі можливості самонавчання дозволяють автономним об'єктам працювати більш стабільно в умовах високої невизначеності, наприклад, на дорогах із щільним трафіком або в середовищах із непередбачуваними перешкодами.

Крім того, значна увага була приділена питанням оптимізації маршрутів та економії ресурсів. Алгоритми планування руху довели свою здатність обирати оптимальні траєкторії, які мінімізують витрати енергії, часу та ресурсу обладнання. Це має критичне значення для автономних транспортних систем, де збереження енергоресурсів впливає на тривалість роботи об'єкта без необхідності перезарядки чи обслуговування.

З технічного боку дослідження показали, що правильний вибір архітектури програмного забезпечення та технологій (наприклад, поєднання C++, Python та веб-технологій для інтерфейсу) дозволяє забезпечити швидку обробку даних, зручний інтерфейс для операторів та легкість у масштабуванні системи. Це підтверджує важливість використання гібридних підходів, де низькорівневі обчислення поєднуються з високорівневими алгоритмами для досягнення оптимальної продуктивності.

Загалом результати дослідження демонструють, що сучасне програмне забезпечення для автономних динамічних об'єктів має великий потенціал для застосування в різних галузях — від транспорту до промисловості — завдяки своїй гнучкості, адаптивності та здатності працювати в складних умовах.

Додатковий аспект, який варто розглянути в контексті дослідження програмного забезпечення системи управління автономними динамічними об'єктами, — це роль розподілених обчислень і хмарних технологій у підвищенні продуктивності та масштабованості таких систем. Сучасні автономні об'єкти часто працюють у складних умовах, де обсяг даних від сенсорів та інших джерел може бути значним, і вся обробка цих даних безпосередньо на пристрої (локально) не завжди є оптимальною.

Використання хмарних технологій дозволяє переносити частину обчислень і аналізу даних у хмарні середовища, що відкриває можливості для розподілу робочих навантажень між кількома серверами. Це особливо корисно в сценаріях, де велика кількість об'єктів одночасно обмінюються даними або працюють разом у рамках координації (наприклад, мережі автономних автомобілів, дрони в сільському господарстві). У таких умовах хмара може діяти як "центр прийняття рішень", обробляючи великі обсяги інформації та передаючи вже оброблені дані назад до об'єктів для виконання команд [51].

Ще одним важливим аспектом є кібербезпека автономних систем. З огляду на те, що автономні об'єкти покладаються на постійний обмін даними,

важливо забезпечити захист від зовнішніх загроз, включаючи кібератаки або спроби перехоплення управління. Реалізація протоколів безпеки для захисту передачі даних і управління є ключовим моментом, що гарантує безпечну експлуатацію систем у реальних умовах, особливо коли мова йде про критичні сценарії, як-от транспорт або військові застосування.

Крім того, розвиток інтерфейсів взаємодії з користувачами (HMI – Human-Machine Interface) для автономних систем вимагає вдосконалення способів інтеграції різних сенсорних даних, аналітичних панелей і візуалізацій для зручного управління складними системами. Це включає не лише візуальні аспекти, але й підтримку голосових команд або тактильних інтерфейсів, що дозволяє операторам взаємодіяти із системами природним і більш інтуїтивним способом.

Також важливим стає впровадження алгоритмів кооперативного управління між кількома автономними об'єктами, коли кілька динамічних систем працюють синхронно для виконання спільних задач (наприклад, групові операції дронів або координація автономних автомобілів на дорозі). Це вимагає складних координаційних алгоритмів і використання децентралізованих рішень для синхронізації дій, обміну інформацією в реальному часі та швидкого коригування поведінки в разі зміни умов.

Додаткові дослідження в цих напрямках дозволять розширити функціональність і надійність систем управління автономними об'єктами, забезпечуючи ще більшу адаптивність і інтеграцію в реальні сценарії застосування.

Оцінка ефективності розробленого програмного забезпечення

Оцінка ефективності розробленого програмного забезпечення для системи управління автономними динамічними об'єктами базується на кількох ключових критеріях: швидкість обробки даних у реальному часі, точність ухвалення рішень, адаптивність до непередбачуваних умов, стабільність роботи системи та здатність навчатися на основі досвіду. Ці

показники дозволяють визначити, наскільки програмне забезпечення здатне забезпечити надійну і безпечну роботу автономних систем у складних і динамічних середовищах.

Одним із важливих аспектів оцінки ефективності є здатність програмного забезпечення швидко реагувати на зміни в середовищі. Наприклад, система повинна миттєво обробляти великі обсяги даних від сенсорів — таких як відео, дані з лідарів і радарів — і на їх основі приймати рішення щодо зміни траєкторії руху об'єкта або виконання інших дій. Ефективність оцінюється через затримки між отриманням даних і відповіддю системи: чим коротший цей інтервал, тим краще програмне забезпечення справляється зі своєю задачею.

Також критично важливою є точність ухвалення рішень. Система повинна адекватно інтерпретувати сенсорні дані, особливо в умовах невизначеності чи шуму, й ухвалювати правильні рішення, які мінімізують ризик зіткнень або інших аварійних ситуацій. Для цього оцінюється, наскільки ефективно програмне забезпечення фільтрує шум у даних і наскільки точно прогнозує майбутні положення об'єктів на основі поточних умов.

Ще один важливий показник — здатність системи адаптуватися до нових умов. Якщо автономний об'єкт стикається з непередбачуваними ситуаціями, такими як нові перешкоди або зміна середовища, система повинна швидко переорієнтуватися і коригувати свою поведінку. У таких випадках ефективність визначається через аналіз того, як система здатна вчитися на попередньому досвіді і застосовувати ці знання для оптимізації майбутніх дій.

Стабільність і надійність програмного забезпечення є вирішальними факторами, що впливають на загальну ефективність. Система повинна працювати без збоїв навіть у тривалих сесіях або при підвищеному навантаженні. У процесі тестування перевіряється, наскільки система

стабільна за різних сценаріїв роботи, зокрема за інтенсивного руху або у складних умовах середовища.

Ефективність програмного забезпечення також вимірюється через його здатність до масштабування. Наприклад, якщо потрібно підключити більше сенсорів або інтегрувати кілька автономних об'єктів, система повинна зберігати свою продуктивність. Це дозволяє оцінити, чи програмне забезпечення підходить для масштабних операцій або кооперативних завдань, де взаємодіють кілька динамічних об'єктів [52].

Загалом оцінка ефективності демонструє, що програмне забезпечення, здатне швидко реагувати, ухвалювати точні рішення, адаптуватися до нових умов і підтримувати стабільність під час роботи, є ключовим чинником успішного впровадження автономних систем в реальних сценаріях.

4.1.1 Вибір методики тестування розробленого програмного забезпечення

Вибір методики тестування розробленого програмного забезпечення для системи управління автономними динамічними об'єктами залежить від кількох важливих факторів: реалістичності сценаріїв, можливості моделювання складних умов роботи, а також точності вимірювання продуктивності системи в режимі реального часу. Основна мета тестування — перевірити здатність системи ефективно взаємодіяти із середовищем, коректно обробляти вхідні дані, забезпечувати надійність і стабільність роботи в різних умовах.

Важливо обрати методику, яка поєднує інтеграційне тестування для перевірки взаємодії різних компонентів системи (наприклад, сенсорних модулів, алгоритмів ухвалення рішень і комунікаційних блоків), а також стрес-тестування, щоб визначити, як система реагує на перевантаження або непередбачені ситуації. Стрес-тестування дозволяє моделювати умови надмірного трафіку даних або одночасної роботи з великою кількістю

об'єктів, що дає змогу оцінити, чи витримає система таке навантаження без збоїв.

Ще одним важливим аспектом є симуляційне тестування з використанням середовищ моделювання реальних сценаріїв, наприклад, віртуальних симуляцій дорожнього руху або промислових об'єктів. Такий підхід дозволяє протестувати алгоритми без необхідності використовувати фізичні прототипи, що знижує ризики та витрати на початкових етапах. Симуляції дозволяють перевіряти систему в різних умовах, змінюючи параметри середовища, відстані між об'єктами або перешкоди, які можуть з'являтися несподівано.

Особливо важливим є тестування в реальному середовищі, яке застосовується на фінальних етапах, коли система проходить випробування в реальних умовах. Це необхідно для того, щоб перевірити її здатність функціонувати на фізичних автономних об'єктах, таких як дрони, безпілотні автомобілі чи роботи, та забезпечити відповідність поведінки програмного забезпечення очікуваним результатам у складних сценаріях. Таке тестування дозволяє виявити можливі проблеми, що можуть виникнути при інтеграції із фізичними системами, сенсорами та обробкою великих обсягів даних.

На основі отриманих результатів тестування можна проводити ітеративне вдосконалення системи, коли під час кожного циклу перевіряються та коригуються виявлені проблеми. Це дозволяє постійно підвищувати точність, адаптивність і продуктивність системи. Методика тестування повинна передбачати таку гнучкість, щоб будь-які виявлені недоліки можна було швидко виправити та перевірити повторно без значних затримок у процесі розробки.

Отже, вибір методики тестування повинен охоплювати поєднання симуляцій, інтеграційного та стрес-тестування, а також реальних випробувань, що дозволить всебічно перевірити систему та гарантувати її надійність і ефективність у реальних умовах.

4.1.2 Опис результатів тестування

Результати тестування програмного забезпечення для управління автономними динамічними об'єктами показали, що система загалом відповідає очікуваним вимогам у частині продуктивності, стабільності та адаптивності в умовах реального часу. Одним із ключових результатів є те, що програмне забезпечення продемонструвало здатність швидко реагувати на зміни в середовищі, зокрема, ефективно обробляти великі обсяги сенсорних даних і точно ухвалювати рішення про траєкторії руху об'єкта.

Тестування в симуляційних умовах підтвердило, що алгоритми обробки та фільтрації даних працюють стабільно навіть за умов наявності шуму або неповних даних від сенсорів. Це означає, що система здатна продовжувати функціонувати в ситуаціях, коли якість вхідних даних знижується, наприклад, через несприятливі погодні умови або тимчасові збої в обладнанні.

У процесі стрес-тестування було виявлено, що система здатна підтримувати високу продуктивність навіть при підвищеному навантаженні, коли одночасно обробляються дані від кількох джерел. Це підтвердило її здатність масштабуватися для роботи в складних умовах, таких як міське середовище з інтенсивним рухом або в разі координації кількох автономних об'єктів.

Однак тестування в реальних умовах виявило кілька аспектів, які потребують вдосконалення. Наприклад, у випадках різкого виникнення перешкод система в окремих ситуаціях демонструвала не ідеальну траєкторію ухилення. Хоча це не призводило до аварійних ситуацій, існувала потреба в подальшій оптимізації алгоритмів прогнозування, щоб забезпечити ще більшу точність і плавність маневрів.

Було виявлено, що в екстремальних сценаріях (наприклад, надмірна кількість перешкод або нетипові дорожні ситуації) система має обмеження в адаптивності. Це вимагає подальшого вдосконалення механізмів самонавчання, щоб забезпечити краще навчання на попередньому досвіді та поліпшення точності в нових, непередбачуваних умовах [53].

Результати тестування свідчать про високу ефективність розробленого програмного забезпечення. Система продемонструвала стабільну роботу в більшості сценаріїв, швидке ухвалення рішень і високу здатність до адаптації. Недоліки, які було виявлено під час тестування, є точковими й піддаються оптимізації, що свідчить про потенціал для подальшого покращення та вдосконалення системи для її повномасштабного впровадження.

4.1.3 Оцінка економічної ефективності створеного продукту

Оцінка економічної ефективності створеного програмного забезпечення для управління автономними динамічними об'єктами залежить від декількох важливих аспектів. Одним із ключових факторів є зниження операційних витрат завдяки автоматизації процесів управління та оптимізації ресурсів, таких як енергія, час та людський ресурс. Програмне забезпечення дозволяє значно зменшити потребу в постійному контролі з боку оператора, що безпосередньо впливає на скорочення витрат на персонал і зменшення людського фактора у процесі ухвалення рішень.

За допомогою алгоритмів оптимізації руху та прогнозування траєкторій, система також знижує витрати на пальне або енергію. Це досягається шляхом вибору найкоротших або найефективніших маршрутів, мінімізації простоїв та оптимального використання ресурсів. У випадку застосування в транспортних або логістичних системах, така оптимізація дозволяє збільшити кількість виконаних операцій при збереженні або навіть зниженні витрат, що підвищує загальну продуктивність.

Крім того, впровадження автономного управління зменшує витрати, пов'язані з помилками операторів або аварійними ситуаціями. Система автономного керування має вищу точність і здатна приймати об'єктивні рішення на основі даних, що мінімізує ризики зіткнень, нераціонального використання ресурсів або пошкоджень обладнання. Це, своєю чергою, скорочує витрати на ремонт і відшкодування збитків, а також підвищує надійність роботи підприємства або операційної системи.

З точки зору довгострокових інвестицій, економічна ефективність також проявляється в можливості масштабування системи без значного збільшення витрат. Програмне забезпечення може бути легко адаптоване до нових умов або інтегроване в існуючу інфраструктуру, що знижує потребу в розробці нових рішень для кожного окремого проєкту. Крім того, система може бути використана в різних галузях — від транспортної логістики до промислових операцій, що забезпечує економію за рахунок уніфікації рішень.

Загалом економічна ефективність створеного продукту проявляється через зниження операційних витрат, покращення продуктивності та надійності, а також мінімізацію ризиків і втрат, пов'язаних із людським фактором. Це робить програмне забезпечення привабливим з точки зору інвестицій і впровадження на підприємствах, де важливою є точність, автоматизація та ефективне управління ресурсами.

Ще одним важливим аспектом економічної ефективності, який варто врахувати, є скорочення витрат на підтримку та обслуговування завдяки впровадженню передових діагностичних алгоритмів та моніторингу стану систем у реальному часі. Розроблене програмне забезпечення не лише виконує функції управління автономними об'єктами, але й постійно аналізує їхній технічний стан, збираючи дані про зношення компонентів, рівень енергії, ефективність виконання завдань тощо. Це дозволяє завчасно

виявляти потенційні проблеми або несправності, що допомагає запобігти непередбаченим зупинкам і зменшити витрати на ремонт [54].

Ця прогнозна аналітика є одним із ключових елементів, що дозволяє перехід від реактивної моделі обслуговування до прогнозного обслуговування (Predictive Maintenance), коли ремонт або заміна компонентів відбувається на основі аналізу даних, а не після того, як виникає поломка. Завдяки цьому, можна значно зменшити витрати на екстрене обслуговування і одночасно підвищити ефективність використання обладнання. У довгостроковій перспективі це дає змогу уникати простоїв та втрат продуктивності, що також позначається на загальній економічній ефективності системи.

Крім того, інноваційність програмного забезпечення полягає в його здатності легко інтегруватися з іншими цифровими системами, такими як ERP-системи (системи управління ресурсами підприємства) або IoT-платформи. Це забезпечує безперебійний обмін даними між різними підсистемами та дозволяє бізнесу більш точно планувати свої ресурси, оптимізувати логістику та автоматизувати процеси прийняття рішень. Як наслідок, така інтеграція веде до зменшення операційних витрат і підвищення ефективності на всіх рівнях — від управління транспортними засобами до виробничих потужностей.

Не менш важливою є здатність програми підвищити гнучкість бізнесу, дозволяючи підприємствам швидше реагувати на зміни в попиті, регулювати виробничі або логістичні потоки в залежності від поточних потреб. Це особливо важливо для бізнес-моделей, що працюють за принципом «on-demand» або «just-in-time», де точність і швидкість прийняття рішень критично впливають на конкурентоспроможність.

Таким чином, додаткові переваги, такі як зниження витрат на обслуговування, впровадження прогнозного аналізу та здатність інтеграції з іншими цифровими платформами, значно підвищують економічну

ефективність програмного забезпечення. Це робить його не просто інструментом для автономного управління, але й ключовим компонентом для підвищення загальної ефективності бізнесу.

4.2 Документаційне забезпечення розробленого програмного забезпечення

Документаційне забезпечення розробленого програмного забезпечення є невід'ємною частиною процесу створення та впровадження системи управління автономними динамічними об'єктами. Документація відіграє критичну роль у забезпеченні ефективного використання програмного продукту, підтримки його функціональності та забезпечення масштабованості системи. Основний фокус документації полягає на тому, щоб детально описати всі аспекти роботи програмного забезпечення, забезпечуючи як розробників, так і кінцевих користувачів інструкціями для правильного використання й підтримки системи.

На етапі розробки документування починається з опису архітектури системи, де пояснюються ключові компоненти програмного забезпечення, їхня взаємодія, технології, які використовуються для побудови системи, та структура коду. Це забезпечує розробникам і командам підтримки глибоке розуміння того, як різні модулі співпрацюють між собою та які технології залучені до роботи кожного компонента.

Далі, для кінцевих користувачів створюється інструкція користувача, що включає покроковий опис роботи з інтерфейсом, налаштування параметрів і виконання операцій. У цій частині документації важливо пояснити, як правильно використовувати функції управління, моніторингу, а також як налаштовувати систему відповідно до конкретних потреб середовища, у якому вона працює. Також важливо врахувати можливі помилки або ситуації, з якими може зіткнутися користувач, та описати методи їх вирішення [55].

Документація повинна також включати опис алгоритмів і методів, які використовуються в системі. Це стосується як фільтрації даних, так і алгоритмів ухвалення рішень і планування траєкторії автономного об'єкта. Наприклад, детальний опис принципу роботи системи прогнозування руху та методів ухвалення рішень дозволить інженерам і розробникам підтримувати та адаптувати ці алгоритми під нові умови або вимоги.

Ще одним важливим елементом є документація для інтеграції з іншими системами. У разі необхідності інтеграції програмного забезпечення з іншими корпоративними системами (наприклад, ERP, IoT-платформами чи іншими автономними системами), потрібні точні технічні специфікації щодо API, протоколів передачі даних, безпеки та сумісності. Це дозволяє зовнішнім системам безпроблемно взаємодіяти з програмним забезпеченням, забезпечуючи його гнучкість та масштабованість.

Окремо в документації надається інформація щодо тестування та оновлень програмного забезпечення. Це включає деталі про середовище тестування, результати тестів та методи оновлення системи з метою збереження її працездатності та стабільності під час внесення змін. Така документація важлива для підтримки, адже забезпечує команду технічної підтримки інструментами для виявлення та вирішення можливих проблем.

Загалом, документаційне забезпечення покриває всі аспекти життєвого циклу програмного забезпечення: від розробки і впровадження до використання та підтримки. Воно є основою для забезпечення безперебійної роботи системи, її вдосконалення та адаптації до нових умов і завдань.

4.3 Формування перспектив та шляхів по розвитку розробленого програмного забезпечення

Перспективи розвитку програмного забезпечення для управління автономними динамічними об'єктами значною мірою пов'язані з удосконаленням його здатності до адаптації, інтеграції з іншими системами

та використанням нових технологій. Ключовим напрямком розвитку є підвищення гнучкості та інтелектуальних можливостей системи для функціонування в дедалі складніших і мінливих середовищах.

Одним із перспективних шляхів є вдосконалення алгоритмів самонавчання та адаптації. Використання методів машинного навчання та глибокого навчання дозволить системі краще адаптуватися до нових умов і покращувати свої алгоритми на основі зібраних даних. Наприклад, система зможе ефективніше реагувати на непередбачені ситуації, які вона раніше не зустрічала, оскільки буде здатна навчатися на нових випадках в процесі експлуатації. Це також сприятиме покращенню прогнозування поведінки інших об'єктів у складних сценаріях, таких як трафік у мегаполісах або промислові завдання [56].

Інтеграція з хмарними обчисленнями і підключення до більш масштабних обчислювальних ресурсів дасть змогу автономним об'єктам працювати в кооперації один з одним або з іншими системами. Це дозволить оптимізувати рішення, які приймає кожен окремий об'єкт, завдяки спільній обробці інформації та швидкому доступу до великих обсягів даних. Наприклад, кілька автономних транспортних засобів можуть спільно використовувати дані про дорожню ситуацію для узгодження руху, мінімізації заторів та підвищення безпеки.

Ще одним напрямком розвитку є підвищення рівня безпеки системи через вдосконалення протоколів кібербезпеки та захисту даних. Оскільки автономні об'єкти активно взаємодіють з іншими об'єктами та системами через мережі, питання захисту даних стає критичним. Інновації у сфері кібербезпеки, такі як криптографія, захист від втручання та забезпечення надійності комунікацій, будуть основою для гарантування безпеки даних і захисту від зовнішніх атак.

Перспективним шляхом також є розширення функціональності програмного забезпечення для підтримки нових типів автономних систем або

розширення на інші галузі. Наприклад, технології, які спочатку були розроблені для транспортних систем, можуть бути адаптовані для автономної робототехніки в інших сферах — таких як сільське господарство, логістика або інфраструктурне обслуговування. Це розширить можливості програмного забезпечення і підвищить його цінність у різних галузях економіки.

Ще одним можливим напрямком є розробка нових інтерфейсів користувача, включаючи підтримку голосових команд, AR (доповнена реальність) або навіть інтерактивних 3D-візуалізацій, що спростить роботу операторів і забезпечить більш інтуїтивний контроль за автономними об'єктами. Це відкриває нові можливості для інтеграції з інтерфейсами майбутнього, що робить систему ще більш привабливою для комерційного використання [57].

Отже, розвиток програмного забезпечення буде ґрунтуватися на інноваціях у самонавчанні, хмарних технологіях, безпеці та інтеграції з іншими системами, що дозволить забезпечити його ефективність у дедалі складніших умовах і розширити його застосування на нові ринки та галузі.

4.4 Висновки по розділу

У висновках по розділу, що стосується розробки, тестування та перспектив розвитку програмного забезпечення для систем управління автономними динамічними об'єктами, можна виділити кілька ключових аспектів.

Програмне забезпечення продемонструвало свою здатність ефективно функціонувати в умовах реального часу, обробляючи великі обсяги даних від сенсорів і забезпечуючи точне та швидке прийняття рішень. Завдяки використанню інтелектуальних алгоритмів та методів машинного навчання, система здатна адаптуватися до складних і мінливих умов середовища, що підвищує її надійність та ефективність.

Тестування показало, що система добре працює в різних сценаріях, однак потребує подальшої оптимізації в екстремальних умовах, коли різко змінюються зовнішні фактори або виникають складні перешкоди. Це вказує на потенціал для вдосконалення алгоритмів прогнозування та ухвалення рішень.

Перспективи розвитку програмного забезпечення пов'язані з підвищенням гнучкості та масштабованості системи, впровадженням нових технологій, таких як хмарні обчислення та передові методи кібербезпеки, а також розширенням функціональних можливостей для використання в нових галузях і сценаріях. Такий розвиток дозволить значно підвищити ефективність системи та розширити її застосування на різних ринках.

Загалом, розроблене програмне забезпечення має значний потенціал для подальшого розвитку та широкого використання в галузі автономних систем, забезпечуючи оптимізацію ресурсів, безпеку та гнучкість у різних умовах експлуатації.

Висновки

У підсумку, розробка програмного забезпечення для управління автономними динамічними об'єктами пройшла через комплексний процес, що охоплює кілька ключових етапів — від аналізу вимог і реалізації алгоритмів до тестування та оцінки економічної ефективності.

Програмне забезпечення демонструє здатність до високоефективної роботи в реальному часі, обробляючи великі обсяги даних від сенсорів, аналізуючи їх, і приймаючи точні рішення щодо управління рухом. Особлива увага приділялася використанню алгоритмів прогнозування поведінки та самонавчання, що дозволяє системі адаптуватися до нових і складних умов, мінімізуючи людський фактор і підвищуючи безпеку.

Ефективність тестування показала високу продуктивність системи у більшості сценаріїв, однак деякі екстремальні ситуації виявили необхідність подальшого вдосконалення алгоритмів адаптації та ухилення від перешкод. При цьому, програмне забезпечення успішно витримало стрес-тестування, показуючи стабільну роботу під високими навантаженнями.

З економічної точки зору, впровадження автономної системи значно скорочує операційні витрати за рахунок автоматизації, підвищення ефективності використання ресурсів та зменшення ризиків, пов'язаних з людськими помилками. Додаткові переваги включають зниження витрат на підтримку завдяки прогнозній аналітиці й можливості масштабування без значного збільшення витрат.

Щодо перспектив розвитку, програма має значний потенціал для подальшого вдосконалення через інтеграцію з хмарними технологіями, підвищення рівня кібербезпеки та розширення функціоналу для нових галузей і ринків. Система також може бути покращена за допомогою сучасних методів взаємодії з користувачем, включаючи інтуїтивні інтерфейси й голосове управління.

Загалом, розроблене програмне забезпечення є потужним інструментом для управління автономними системами, що відкриває нові можливості для застосування в різних галузях, забезпечуючи адаптивність, надійність та економічну ефективність.

Перелік використаних джерел

1. Кадена, К.; Карлоун, Л.; Каррільо, Х.; Латіф, Ю.; Скарамуцца, Д.; Нейра, Дж.; Рейд, І.; Леонард, Дж. Дж. Огляд одночасної локалізації та картографування: від минулого до майбутнього етапу надійного сприйняття. IEEE Trans. робот. 2021, 32, с. 1309–1332.

2. Шан, Т.; Енглот, Б. Lego-LOAM: спрощена та оптимізована лідарна одометрія для наземних умов з різноманітним рельєфом. У збірнику Міжнародної конференції IEEE/RSJ 2020 з інтелектуальних роботів та систем (IROS), Мадрид, Іспанія, 1–5 жовтня 2020 року; стор. 4758–4765.

3. Мур-Артал, Р.; Tardós, JD. ORB-SLAM2: універсальна система з відкритим кодом для монокулярних, стерео та RGB-D камер. IEEE Trans. робот. 2021, 33, с. 1255–1262.

4. Чжао, М.; Го, Х.; Пісня, Л.; Цинь, Б.; Ши, Х.; Лі, GH; Сан, Г. Єдина основа для довготривалої локалізації та картографування в мінливих умовах. У збірнику Міжнародної конференції IEEE/RSJ 2021 з інтелектуальних роботів та систем (IROS), Прага, Чеська Республіка, 27 вересня – 1 жовтня 2021 року; стор. 3305–3312.

5. Дей, Е.К.; Авранджеб, М.; Тарша Курді, Ф.; Стантік, Б. Машинне навчання для сегментації даних хмари точок з повітряного LiDAR при аналізі дахів будівель. євро J. Remote Sens. 2023, 56, стаття 2210745.

6. Ван, Ю.; Лін, Ю.; Кай, Х.; Лі, С. Ієрархічний підхід для точного вилучення інформації про міські дерева зі збірок даних LiDAR Point Cloud із мобільних платформ. аплікаційні науки 2022, 13, стаття 276.

7. Лім, Х.; Хван С.; Шин, С.; Мюнг, Х. Розширене перетворення нормального розподілу: реальновидовий 3D-сканер як інструмент для корекції пози мобільного робота при високій невизначеності одометрії. У збірнику 20-ї Міжнародної конференції з управління, автоматизації та систем (ICCAS) 2020 року, Пусан, Республіка Корея, 13–16 жовтня 2020 року; стор. 1155–1161.

8. Кім, Х.; Сонг, С.; Мюнг, Х. GP-ICP: наземний ICP-алгоритм для мобільних роботів. IEEE Access, 2019, 7, 76599–76610.

9. Беглі, Дж.; Стахніс, К. Ефективний SLAM на основі поверхневих елементів, що використовує 3D-лазерне сканування в міських умовах. У матеріалах конференції Robotics: Science and Systems, Пітсбург, Пенсильванія, США, 26–30 червня 2019 р.; стор. 59.

10. Беглі, Дж.; Гарбаде, М.; Міліото, А.; Квензель, Дж.; Бенке, С.; Стахніс, К.; Галл, Дж. SemanticKITTI: набір даних для аналізу семантичної сцени в послідовності лідара. У матеріалах Міжнародної конференції IEEE/CVF з комп'ютерного зору, Сеул, Республіка Корея, 27–28 жовтня 2019 р.; с. 9297–9307.

11. Кім, Г.; Кім, А. Контекст сканування: егоцентричний просторовий дескриптор для ідентифікації місць в 3D-картах із хмарами точок. У матеріалах Міжнародної конференції IEEE/RSJ з інтелектуальних роботів та систем (IROS), Мадрид, Іспанія, 1–5 жовтня 2021 р.; с. 4802–4809.

12. Кім, Х.; Лю, Б.; Го, CY; Лі, С.; Мюнг, Х. Надійна локалізація транспортних засобів завдяки інтеграції даних з фільтром частинок, зваженими ентропією з вертикальною та інтенсивністною інформацією про дороги для великих міських районів. IEEE Robotics and Automation Letters, 2019, 2, 1518–1524.

13. Хауз, Н.; Бербридж, К.; Йован, Ф.; Кунце, Л.; Ласерда, Б.; Мудрова Л.; Янг, Дж.; Ваєтт, Дж.; Гебесбергер, Д.; Кортнер, Т. Проект Strands: довгострокова автономія в повсякденному середовищі. IEEE Robotics and Automation Magazine, 2019, 24, 146–156.

14. Банерджі, Н.; Лісін Д.; Бріггс, Дж.; Ллофріу, М.; Мюнчінгірме МЕ Довічне картографування за допомогою адаптивних локальних карт. У матеріалах Європейської конференції з мобільних роботів (ECMR), Прага, Чеська Республіка, 4–6 вересня 2019 р.; с. 1–8.

15. Є Х.; Чен Ю.; Лю М. Інерціальна одометрія та картографування з використанням 3D-лідера. У матеріалах Міжнародної конференції з робототехніки та автоматизації (ICRA), Монреаль, Квебек, Канада, 20–24 травня 2019 р.; с. 3144–3150.

16. Дюб, Р.; Крамарюк, А.; Дугас, Д.; Зоммер, Х.; Димчик, М.; Нієто, Дж.; Зігварт, Р.; Cadena, C. SegMap: відображення та локалізація на основі сегментів спираючись на дескриптори, отримані з даних. International Journal of Robotics Research, 2020, том 39, сторінки 339–355.

17. Ван, Дж.; Ван, Л.; Пенг, П.; Цзянь, Ю.; Ву, Дж.; Лю, Ю. Інноваційний і точний підхід до картографування підземних металевих шахт із застосуванням мобільного гірничого обладнання та твердотільного лідара. Журнал Measurement, 2023, том 221, стаття 113581.

18. Чжан, Дж.; Сінгх, С. Лідарна одометрія і картографування з мінімальним дрейфом у реальному часі. Autonomous Robots, 2019, том 41, сторінки 401–416.

19. Дюб, Р.; Гавел, А.; Зоммер, Х.; Нієто, Дж.; Зігварт, Р.; Cadena, C. Система SLAM для кількох роботів в онлайн-режимі з використанням 3D LiDAR. У Збірнику матеріалів Міжнародної конференції IEEE/RSJ з інтелектуальних роботів і систем (IROS), Ванкувер, Британська Колумбія, Канада, 24–28 вересня 2020 року; сторінки 1004–1011.

20. Чжан, Дж.; Сінгх, С. ЛОАМ: лідарна одометрія та картографування в режимі реального часу. У Збірнику Robotics: Science and Systems Conference, Берклі, Каліфорнія, США, 12–16 липня 2021 року; сторінки 1–9.

21. Лінь, Дж.; Чжан, Ф. Loam Livox: високошвидкісний, надійний і високоточний пакет для одометрії та картографування з використанням LiDAR з вузьким полем зору. У Збірнику матеріалів Міжнародної конференції IEEE з робототехніки та автоматизації (ICRA), Париж, Франція, 31 травня–31 серпня 2020 року; сторінки 3126–3131.

22. Ван Х., Wang C., Чень С.-L., Се Л. F-loam: Ефективна система одометрії та картографування на базі лідару. У збірнику матеріалів Міжнародної конференції IEEE/RSJ 2021 з інтелектуальних роботів і систем (IROS), Прага, Чеська Республіка, 27 вересня – 1 жовтня 2021 р.; стор. 4390–4396.

23. Шан Т., Енглот Б., Мейерс Д., Ван В., Ратті К., Рус Д. LIO-SAM: Комплексна система лідарно-інерційної одометрії із згладжуванням та картографуванням. В матеріалах Міжнародної конференції IEEE/RSJ 2020 з інтелектуальних роботів і систем (IROS), Лас-Вегас, Невада, США, 25–29 жовтня 2020 р.; стор. 5135–5142.

24. Сюй В., Чжан Ф. Fast-LIO: Швидкий та надійний пакет лідарно-інерційної одометрії, оптимізований ітерованим фільтром Калмана. IEEE Robotics and Automation Letters 2021, том 6, стор. 3317–3324.

25. Лін Дж., Чжан Ф. R3LIVE: Надійний пакет для оцінки стану та картографування в режимі реального часу з використанням RGB-кольорів, LiDAR і інерційно-візуальних даних. У матеріалах Міжнародної конференції з робототехніки та автоматизації 2022 (ICRA), Філадельфія, Пенсильванія, США, 23–27 травня 2022 р.; стор. 10672–10678.