15. Об’єктно-орієнтоване проектування

Більшість сучасних програм пишуть у об’єктно-орієнтованому стилі за допомогою класів і об’єктів, то ж і ми не можемо оминути увагою цю тему. Клас – тип даних, оголошений програмістом, що поєднує дані та методи. Дані зберігають стан об’єкта, екземпляра класу, а методи описують його поведінку. Завдяки цьому класами легко моделювати сутності реального світу. В попередньому розділі ми вже проектували класи: *Tour* для моделювання одного маршруту комівояжера і *Solver* для реалізації генетичного алгоритму. Читачі могли відчути відмінність у трактуванні частин програми, порівняно з попередніми розділами посібника. За процедурного підходу ми акцентуємо увагу на алгоритмі перетворення даних, а самі дані розглядаємо дещо відсторонено. Вони відіграють роль тільки параметрів функцій: вхідних параметрів або результатів. Уся програма – це сукупність функцій, що викликають одна одну, серед яких одна є головною, кожна функція описує свою частинку алгоритму або «конвеєра опрацювання даних». За об’єктного підходу програміст мислить іншими категоріями – сутностями предметної області, втіленими в програмі екземплярами класів. Як і в реальному житті, об’єкти програми взаємодіють один з одним, надсилаючи повідомлення. Пригадайте попередню програму: екземпляр класу *Tour* на вимогу *Solver* може продукувати потомство, екземпляр упорядкованого контейнера вміє додавати нові елементи, об’єкти типу *Tour*, запитуючи у них їхню довжину, екземпляр *Solver* на вимогу вікна діалогу повідомляє найкращий тур тощо. На етапі проектування ми запитували, які дані має містити тур, які дії з цими даними він може виконати. Ми міркували про тур як про окрему самодостатню сутність. Так само окремим гравцем був «розв’язувач», об’єкт типу *Solver*. На нашу думку, об’єктно-орієнтований підхід суттєво полегшує написання складних програм.

Процедурний та об’єктно-орієнтований підходи можна порівняти також на прикладі програм розділу 9: за процедурного підходу головна програма викликала відповідні функції для опрацювання «великого цілого», а оголошення класу *LargePositiveInteger* дало змогу написати програму з використанням операторів так, ніби «велике ціле» є вбудованим типом даних, схожим на *int* чи *long long*.

Під час створення об’єктно-орієнтованої програми доводиться знаходити відповіді на кілька важливих запитань. Передусім з якими сутностями предметної області маємо справу? Які дані описують стан цих сутностей, та яке коло їхніх обов’язків – що вони «вміють»? Які дані та поведінкові можливості є обов’язковими, а якими можна знехтувати? На цьому етапі доводиться абстрагуватися від неважливих особливостей предметної області, оскільки неможливо охопити в програмі геть усе. У підсумку виділяють перелік важливих сутностей, їхніх даних і методів, і проектують класи, що їх моделюватимуть.

Було помічено, що складним системам притаманна ієрархічна будова. Наприклад, персональний комп’ютер складається з системного блока та периферійних пристроїв, системний блок так само містить материнську плату, відео- та мережеву плати, жорсткий диск, материнська плата – процесор, оперативну пам’ять, порти тощо. Це приклад ієрархії за структурою. Але є ще й ієрархія за типом: процесори мають різну тактову частоту, різну кількість ядер, різних виробників; жорсткі диски бувають різної конструкції, обсягу та швидкодії. Процесор – загальне поняття, процесор певної моделі певного виробника – дуже конкретне. Усі процесори мають спільні властивості, проте процесор конкретної моделі уточнює їх і може додавати якісь свої.

Після виділення сутностей предметної області програміст з’ясовує, чи не можна їх класифікувати за схожістю. Чи не описують вони споріднені поняття? Чи не є сутності конкретизаціями якогось загального поняття? Відповіді на ці запитання важливі, бо класи можна оголошувати на базі вже оголошених. Новий клас наслідує можливості свого базового класу. Якщо один раз описати програмно загальне поняття як клас, то оголошення

нових конкретизацій підкласами суттєво пришвидшиться. Програміст мав би класифікувати виділені раніше поняття від загального до конкретного. Потрібно також зрозуміти, чи не є об'єкт частиною іншого об'єкта, щоб правильно змоделювати ієрархію за структурою.

Далі з'ясовують, як об'єкти взаємодіють. Коли створюють екземпляри виділених класів? Які повідомлення та кому надсилає об'єкт упродовж свого існування, які параметри він передає? Які повідомлення і від кого він має опрацьовувати? Як об'єкти «бачитимуть» один одного? Структурно об'єктно-орієнтована програма схожа на мережу сутностей, пов'язаних повідомленнями.

Наступний етап – втілення проєкту в коді програми. Зазвичай він вносить доповнення і виправлення до проєкту, адже деякі його аспекти стають зрозумілими вже під час написання програми. Не зайвим на цьому етапі буде модульне тестування коду, щоб гарантувати правильну роботу окремих частин програми. Виправлення виявлених помилок також може вплинути на початковий проєкт. Програма є матеріальною, тому доведеться вирішити, якою має бути її фізична структура, в яких файлах зберігати класи програми, чи потрібно створювати бібліотеки класів тощо.

Усе сказане проілюструємо на прикладі написання невеликої ігрової програми. Ми не претендуватимемо на створення досконалої іграшки, адже для цього довелося б вивчити та застосувати спеціалізоване програмне забезпечення. У нас інша мета: показати, як проектують ієрархії класів і налагоджують взаємодію їхніх екземплярів для розв'язання задачі достатньої складності.

Задача 52. *Створити комп'ютерну гру – "стрілялку" в двовимірному просторі. Фабула гри проста: верх екрана – небо, по якому летять літаки, низ екрана – земля, якою рухається зенітна гармата. Літаки з'являються автоматично з одного краю екрана, на різній висоті, з різною початковою швидкістю. Літак рухається горизонтально з постійною швидкістю на постійній висоті, доки не досягне протилежного краю екрана, або не буде збитий. Гарматою керує гравець за допомогою клавіатури. Його завдання – збити якомога більше літаків. Снаряди рухаються завжди вертикально догори з постійною швидкістю, доки не вилетять за верхню межу екрана, або до влучання в літак.*

Для виконання такої програми потрібне інтерактивне графічне середовище. Ми, як і в попередньому розділі, використаємо бібліотеку MFC та створимо віконний застосунок. Цього буде достатньо для демонстрації використання спроектованих класів.

15.1. Виділення сутностей предметної області

Почнемо з сутностей, згаданих у формулюванні задачі. Здається, у програмі потрібно моделювати *літаки*, *гармату* й *снаряди*. Зрозуміло, що для гармати вистачить однієї моделі. Її можна описати, наприклад, класом *TGun*. А скільки моделей снарядів оголошувати? Адже на арені гри може перебувати одночасно багато снарядів у різних місцях екрана. Проте усі вони мають схожі властивості – координати і швидкість – та однакову поведінку: завжди рухаються вертикально догори. Снаряди є різними екземплярами однакового типу, тому для їхнього моделювання вистачить одного класу. Назвемо його *TBullet*. Схожі міркування правильні і щодо літаків. Клас, модель літака, назвемо *TAircraft*.

Якщо програма використовує кілька екземплярів одного типу, то закономірно виникає запитання, як їх зберігати. Для одного значення вистачить простої змінної, а для групи значень потрібен контейнер, у найпростішому випадку – масив. Наприклад, у попередньому розділі тур містив масив точок, генетичний алгоритм – масив турів. Чи підійде масив для зберігання літаків, снарядів? Під час гри доведеться постійно додавати до контейнера нові екземпляри та вилучати ті, що покинули арену, чи зникли після влучання. По-перше, контейнер має легко змінювати свій розмір; по-друге, має бути простий спосіб, щоб

вилучати елементи з середини контейнера. Масив не пристосовано для таких завдань, натомість добре підходить зв'язна структура, наприклад, список. Використаємо *TAircraftList*, список літаків, і *TBulletList*, список снарядів.

Продовжимо міркування і виділимо «приховані» сутності. Гравець керує гарматою за допомогою *клавіатури*, отже, потрібен об'єкт, який розпізнає подію натискання клавіатури, повідомляє про неї гармату, повідомляє також код натиснутої клавіші. Клавіатуру моделюватиме *TKeyboard*. В умові сказано, що літаки та снаряди рухаються екраном автоматично. Але що саме змушує їх змінювати координати? Якщо не дії гравця, то плин часу! Нам потрібен *таймер*, який би сигналізував об'єктам програми про зміни значення системного годинника. Відповідний тип назвемо *TTimer*. А звідки беруться літаки? Хтось мав би їх створювати і надсилати на арену гри. Отже, існує *аеродром*, звідки вони прилітають. Відповідний тип назвемо *TAirfield*. Арена гри, чи графічний екран, також є сутністю, що взаємодіє з іншими об'єктами. Вона мала б щонайменше повідомляти межі екрана. Змоделюємо її екземпляром типу *TScreen*. Зазвичай в ігрових програмах ведеться підрахунок очок, здобутих гравцем: щось він витрачає на власні дії (постріли), а щось заробляє (за вражені мішені). Було б зручно, щоб очки рахувала не головна програма, а окрема сутність, *рефері*. Можемо для цього створити тип *TReferee*. Гравця також можна моделювати значенням типу *TPlayer*, щоб ідентифікувати його за іменем і вести облік сеансів гри.

У процесі виділення сутностей головне не перестаратися. Кожен новий клас бере на себе певне коло обов'язків і робить інші класи незалежними від деталей реалізації. Наприклад, клас *TScreen* може інкапсулювати деталі графічного середовища і надавати зручний інтерфейс. Проте проектування такого класу доцільне тоді, коли ви плануєте переносити свою програму до різних графічних середовищ: перенесення означатиме зміну лише одного класу. Ми плануємо використовувати тільки MFC, тому доцільність *TScreen* поки що під сумнівом. У ході об'єктно-орієнтованого проектування дуже часто повертаються до вже виконаних кроків, якщо з'ясовується, що варто було б дещо в них змінити чи доповнити. Ми також перейдемо до наступного етапу, до з'ясування кола обов'язків кожної сутності, а за потреби повернемося до уточнення їхнього переліку.

15.2. Обов'язки виділених сутностей

Дамо відповіді на два запитання: які дані *містить* сутність? Що *робить* сутність?

Гармата є видимим об'єктом, має екранні координати, вигляд, розмір, можливо, боєзапас. Вона уміє відображати себе, переміщатися ліворуч і праворуч, виконувати постріл. Дані об'єкта моделюють полями відповідного типу, а поведінку – методами. Метод відображення взаємодіє з графічним екраном і суттєво залежить від його можливостей. Методи переміщення змінюють абсцису гармати та виконують перемальовування, постріл створює новий об'єкт – снаряд – з тими початковими координатами, в яких розташовано цівку гармати. Створений снаряд потрібно додати до списку снарядів, тому гармата має «знати», де розташовано список. Переміщення і постріл ініціюють повідомлення від клавіатури, наприклад, стрілки переміщають гармату праворуч-ліворуч, пропуск стріляє.

Снаряд – видимий об'єкт, що має екранні координати, вигляд, розмір, швидкість. Він уміє відображати себе, переміщатися догори, перевіряти виліт за межі екрана, перевіряти влучання в літак. Як програмно змоделювати швидкість? Можна запропонувати два способи. Якщо переміщувати об'єкт через однакові інтервали часу, тоді швидкість моделюють величиною кроку переміщення. Якщо ж вважати крок переміщення сталим, то швидкість визначатиметься інтервалом часу, після якого потрібно перемістити об'єкт. Перший спосіб виглядає природнішим, то ж його й оберемо, будемо задавати крок переміщення. Нагадаємо, що для всіх снарядів він однаковий. Метод відображення, як і у гармати, взаємодіє з графічним екраном. Метод переміщення змінює ординату об'єкта на заданий крок. Щоб перевірити виліт об'єкта за межі екрана, відповідний метод має довідатися в екрана його

висоту, або використати з цією метою деяку загальнодоступну константу. Про метод перевірки влучання поговоримо після розгляду можливостей літака.

Літак – видимий об'єкт, який має екранні координати, вигляд, розмір, швидкість. Він уміє відображати себе, переміщатися вперед, перевіряти виліт за межі екрана, перевіряти влучання снаряду, можливо, вибухати. Сказане наштовхує на думку, що з погляду програми літак дуже схожий на снаряд. Перелік даних і методів майже збігається. Як і у снаряда, швидкість моделюватимемо кроком переміщення. У різних екземплярів літака ця величина відрізнятиметься, адже за умовою літаки рухаються з різною швидкістю. Метод переміщення змінює абсцису об'єкта, метод перевірки вильоту порівнює її з шириною екрана. Для перевірки влучання потрібна взаємодія двох об'єктів: літака і снаряда. Відповідний метод достатньо оголосити в одного з них, а параметром методу зробити другий. Легше міркувати так, що великий літак перевіряє влучання в себе маленького снаряда. Реалізувати такий спосіб програмно також легше: літак знає своє розміри, а снаряд може мати ширину один піксель, тому для перевірки достатньо буде лише його координат. Остаточно вирішимо оголосити метод перевірки влучання в класі літака, а клас снаряда доповнити методами отримання координат об'єкта.

Аеродром – уявний об'єкт, який створює літаки та додає їх до списку літаків. Отже, йому потрібне посилання на список і генератор випадкових чисел, щоб обирати висоту та швидкість чергового літака. Аеродром мав би генерувати літаки у випадкові моменти часу. Як цього досягти? Ми вже вирішили «оживити» літаки та снаряди за допомогою таймера. Те саме можна зробити з аеродромом: наділимо його полем даних, що зберігатиме випадкове додатне значення, інтервал до генерування наступного літака, й оголосимо метод опрацювання повідомлень від таймера. Цей метод мав би зменшувати значення інтервалу, як тільки той вичерпається, настане момент створення літака та генерування значення наступного інтервалу.

Список снарядів – контейнер, який використовує зв'язну пам'ять, уміє додавати новий снаряд, вилучати снаряд (що вилетів за межі екрана чи влучив у літак), повідомляти снаряди, що вони мають рухатися, мають зобразити себе, знаходити снаряди, що вилетіли за межі екрана. *Список літаків* – контейнер, що використовує зв'язну пам'ять, уміє додавати новий літак, вилучати літак (що вилетів за межі екрана чи вибухнув внаслідок влучання снаряда), повідомляти літаки, що вони мають рухатися, мають зобразити себе, знаходити літаки, що вилетіли за межі екрана. Бачимо, що описані списки також дуже схожі між собою. Вони відрізняються хіба що типом збережених елементів, а не функціональністю. Щоб зобразити елементи списку, треба їх перебрати і кожному надіслати повідомлення «покажись». Літак і снаряд зроблять це по-різному, але то ж їхня відповідальність, не списку. Так само і з пошуком об'єктів, що покинули арену: треба перебрати елементи списку і запитати у кожного, чи не перебуває він за межами екрана. Літак і снаряд виконають перевірку, що відповідає їхньому типові, та повідомлять результат, а спискові залишиться лише вилучити чи продовжити зберігати об'єкт. Відмінність між списками виявляється у способі пошуку влучань. Для цього потрібно порівняти кожен снаряд з кожним літаком – перебрати всі літаки, перебрати всі снаряди. Ми вирішили, що метод літака для перевірки влучання отримує снаряд як параметр, тому й методи списків треба спроектувати відповідно. Метод списку снарядів перебиратиме елементи списку та передаватиме кожного з них методові списку літаків, який перебиратиме літаки та кожному з них передасть для перевірки отриманий снаряд.

Клавіатура розпізнає натискання клавіш і надсилає повідомлення про подію, в описі якої зазначає код натиснутої клавіші. Нам не доведеться затрачати додаткових зусиль на оголошення відповідного класу, адже такі функціональні можливості вже реалізовано на рівні операційної системи. Кожен віконний застосунок може отримувати від неї повідомлення *WM_KEYDOWN* та опрацьовувати його. Схожа ситуація і з *таймером*: він є сутністю ядра операційної системи. В момент створення задають інтервал спрацювання. Після запуску

таймер регулярно надсилає застосункові повідомлення *WM_TIMER*. З попереднього розділу ми знаємо, що під час створення діалогу MFC автоматично оголошується клас, який описує вікно діалогу. Він цілком зможе відіграти роль графічного екрана. Заради пришвидшення виконання завдання використаємо також і його. Так ми зможемо обійтися без оголошення класів *TKeyboard*, *TTimer*, *TScreen*.

Рефері – уявний об'єкт, що містить лічильники пострілів, збитих і пропущених літаків, уміє обчислювати та повідомляти рахунок гри. Кожен з літаків міг би, крім інших даних, пам'ятати власну цінність, виражену балами, та повідомляти її рефері після збиття чи після втечі. Поповнення боєзапасу гармати також могло б впливати на рахунок гри, щоб заохочувати гравця економити набої. Реалізацію рефері залишимо читачам як вправу.

15.3. Класифікація сутностей

Вирішальною для класифікації об'єктів є спільність їхньої поведінки. Перегляньмо ще раз описані нами сутності. Ми вже відзначали схожість деяких з них. Наприклад, дуже близькими є список літаків і список снарядів. Вони відрізняються лише способом перевірки влучання. Спільну частину поведінки, як от додавання елементів, відображення, переміщення тощо, ми можемо описати в єдиному класі «список рухомих об'єктів». Список літаків і список снарядів стануть його підкласами, додавши до його можливостей свій власний метод перевірки влучання.

Видимі об'єкти, літак, снаряд і гармата, мають однаковий набір даних і схожу поведінку: уміють відображати себе. Зрозуміло, що схожі між собою літак і снаряд суттєво відрізняються від гармати. Тут ієрархія типів буде складнішою, ніж у списків. Поняття «літак» і «снаряд» можна узагальнити поняттям «рухомий об'єкт». Кожен рухомий об'єкт вміє виводити своє зображення у графічне вікно. Цю поведінку можна запрограмувати в базовому класі. Переміщення екраном рухомі об'єкти виконують по-різному залежно від типу, тому відповідний метод оголошують у базовому класі абстрактним і перевизначають у підкласах. Здається, гармата має не достатньо спільних рис поведінки, щоб належати до тієї ж ієрархії, що й літак та снаряд, проте у видимих об'єктів однаковий спосіб побудови свого зображення. За допомогою засобів MFC ми могли б виводити у вікно діалогу готові растрові зображення, а в такому випадку спільний базовий клас може суттєво спростити необхідні початкові налаштування, наприклад, завантаження зображень з ресурсів застосунку.

Клас *TAirfield* ніяк не пов'язаний з іншими сутностями програми відношенням наслідування, а клас вікна діалогу є членом власної ієрархії класів: він наслідує стандартний клас *CDialog*.

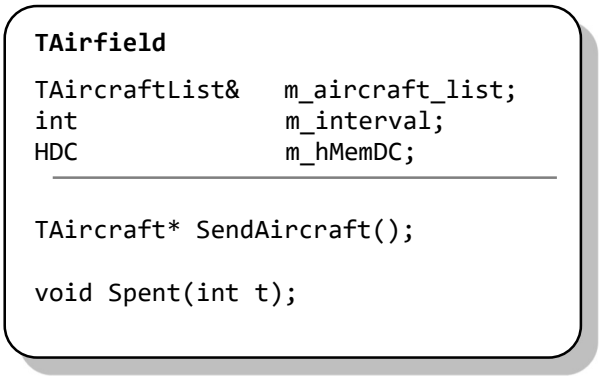

Рис. 17. Проєкт класу *TAirfield*

Настав час формалізувати все сказане та зобразити проєкт майбутніх класів за допомогою діаграми. Почнемо з найпростішого. На рис. 17 зображено клас *TAirfield*. Це

простий клас, що отримує повідомлення від таймера, генерує нові літаки і додає їх до списку літаків. У момент створення екземпляра класу потрібно надати йому посилання на цей список.

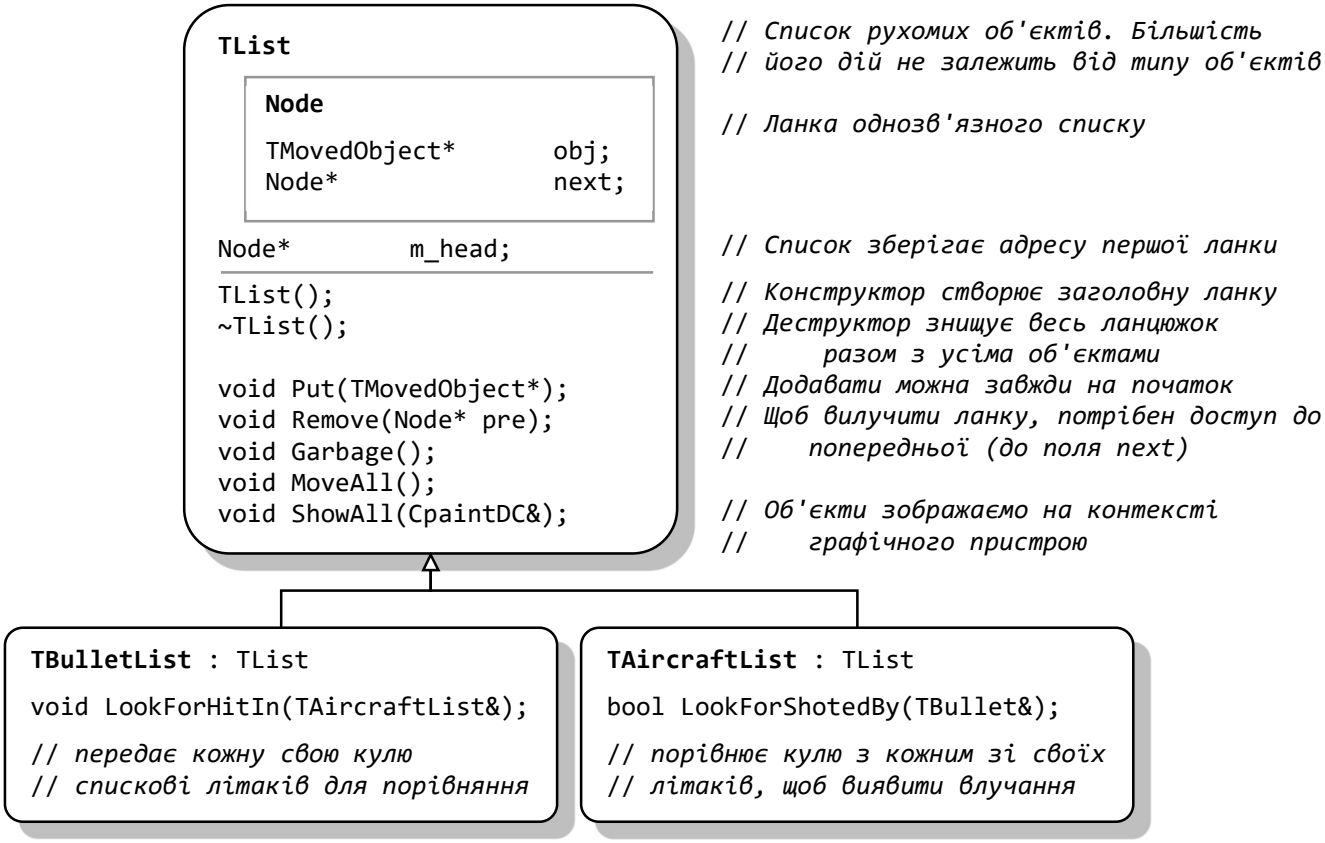

Рис. 18. Ієрархія списків рухомих об’єктів

На рис. 18 зображено ієрархію контейнерів. Усе обслуговування зв’язної структури, призначеної для зберігання рухомих об’єктів, винесено в спільний базовий клас. Підкласи спеціалізуються на своїх особливих способах перевірки влучання: список куль передає свої кулі спискові літаків, список літаків передає кулю кожному літакові для порівняння координат. Базовий клас – лінійний однозв’язний список з заголовною ланкою. Ланку списку описує вкладений тип *Node*. Заголовна ланка потрібна для того, щоб перша ланка списку не відрізнялася від усіх інших: її адресу, як і всіх інших, міститиме поле *next* попередньої (заголовної) ланки. Елементом ланки є поліморфний вказівник на базовий клас рухомого об’єкта. Додавати ланку можна в довільному місці списку, адже вилучення об’єктів зі списку непередбачуване, тому немає сенсу підтримувати якийсь порядок. Найлегше додавати на початок, одразу після заголовної ланки. Рухомі об’єкти зображатимемо в графічному середовищі ОС Windows за допомогою класів бібліотеки MFC, тому список відображатиме свої елементи на контекст графічного пристрою.

Ієрархію видимих об’єктів зображено на рис. 19. У статичних полях базового класу зберігаються усі необхідні растрові зображення об’єктів, статичний метод завантажує їх з ресурсів застосунку, а метод *Show* виводить відповідне зображення на контекст графічного пристрою. Ця частина ієрархії класів залежить від середовища виконання застосунку. При перенесенні на іншу платформу її доведеться змінити. Проміжний абстрактний клас *TMovedObject* об’єднує типи об’єктів, які рухаються самі, та є основою для створення колекцій рухомих об’єктів.

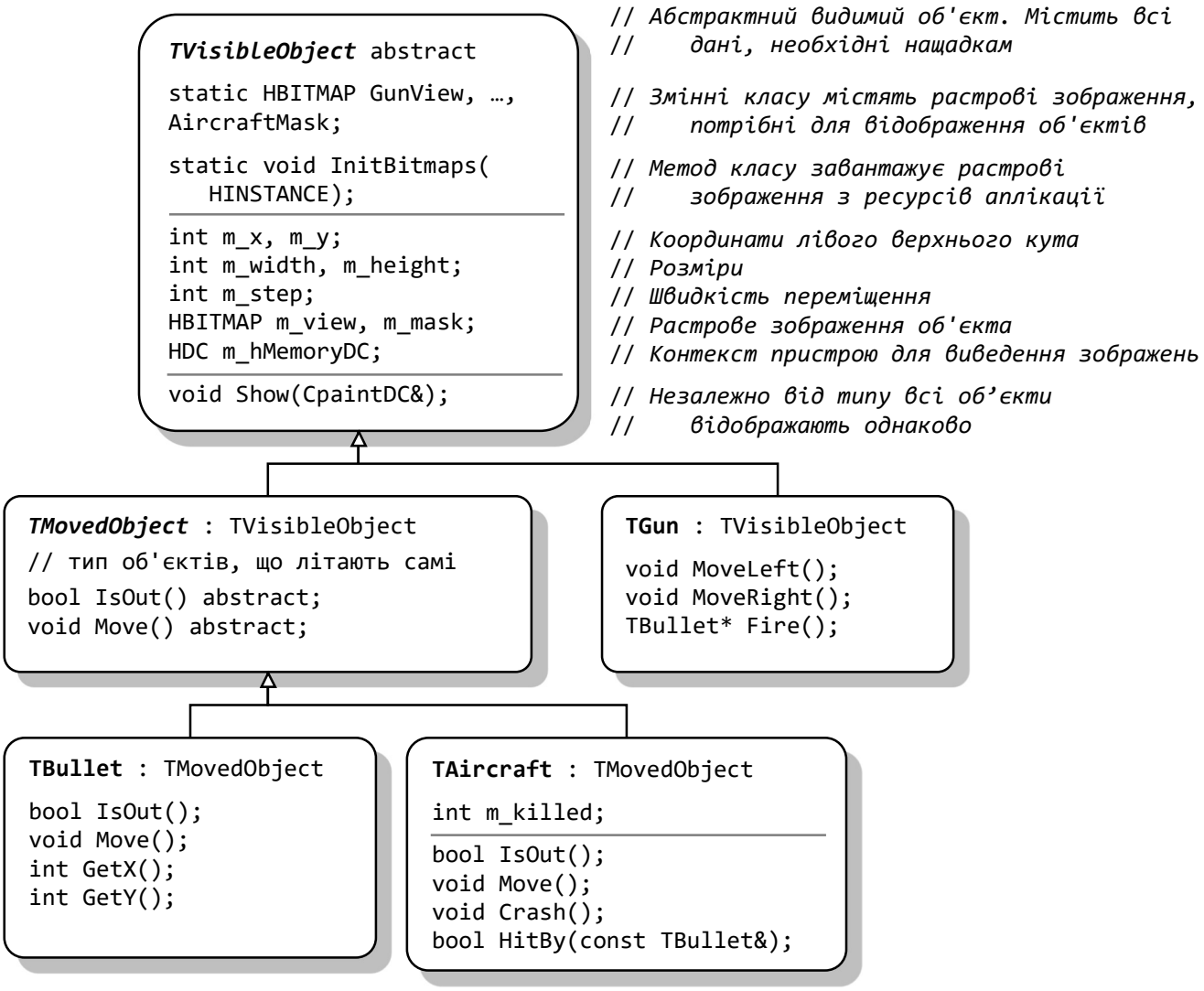

Рис. 19. Ієрархія видимих об’єктів

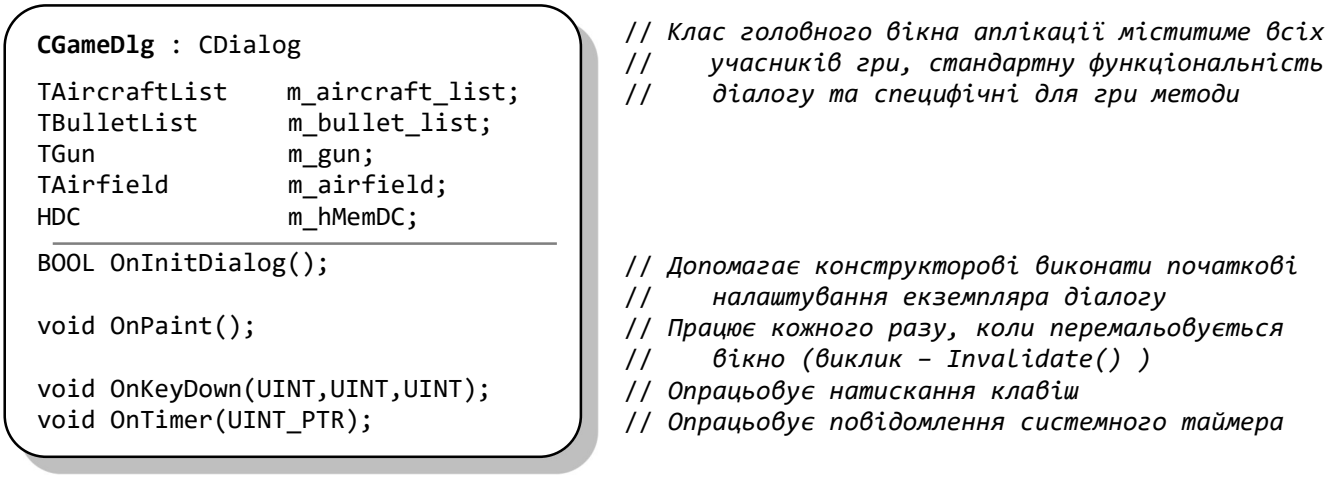

Рис. 20. Проєкт класу вікна застосунку

У літака з’явився метод *Crash*, що моделює вибух, і поле *m_killed*, про яке ми не згадували раніше. Було б цікаво, якби підбитий літак не одразу зникав з екрана. Літак може перебувати у двох станах: звичайний літак рухається горизонтально, підбитий літак змінює

зовнішній вигляд і рухається інакше, втрачаючи висоту. Через деякий час підбитий літак зникає. Поле *m_killed* – поле стану об'єкта. У звичайного літака *m_killed* = 0. Метод *Crash* змінює стан об'єкта: *m_killed* набуває невід'ємного значення, міняється зовнішній вигляд (поля *m_view*, *m_mask*), інакше починають працювати методи *Move*(), *IsOut*(). Такі відмінності навіть можна вважати підставою для оголошення підкласу «підбитий літак», але ми обмежимося відповідними налаштуваннями методів.

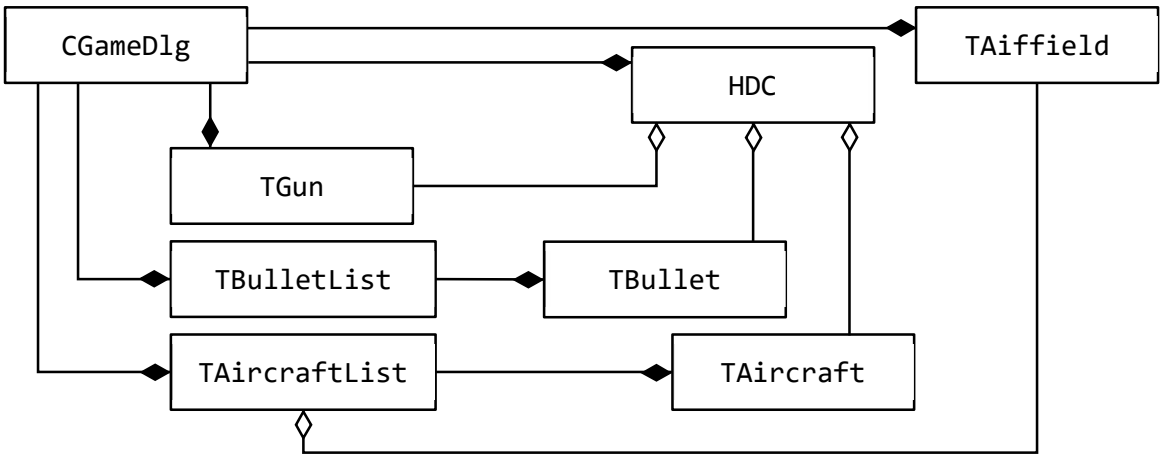

Рис. 21. Діаграма класів ігрового застосунку

Завдання екземпляра діалогу (проєкт його типу зображено на рис. 20) – створення гармати, списків, таймера, опрацювання повідомлень від таймера і клавіатури та перемальовування вікна. Діаграма класів (рис. 21) зображає відношення власності між сутностями програми: замальована стрілка веде від власника до об'єкта, який він створює та знищує, а прозора – до об'єкта, на який він має посилання та використовує впродовж існування.

15.4. Організація взаємодії

Для проектування взаємодії об'єктів використовують діаграми послідовностей. На них зображають, хто, кому, які повідомлення надсилає. Ми побудуємо кілька діаграм, щоб зобразити окремі аспекти взаємодії сутностей ігрового застосунку.

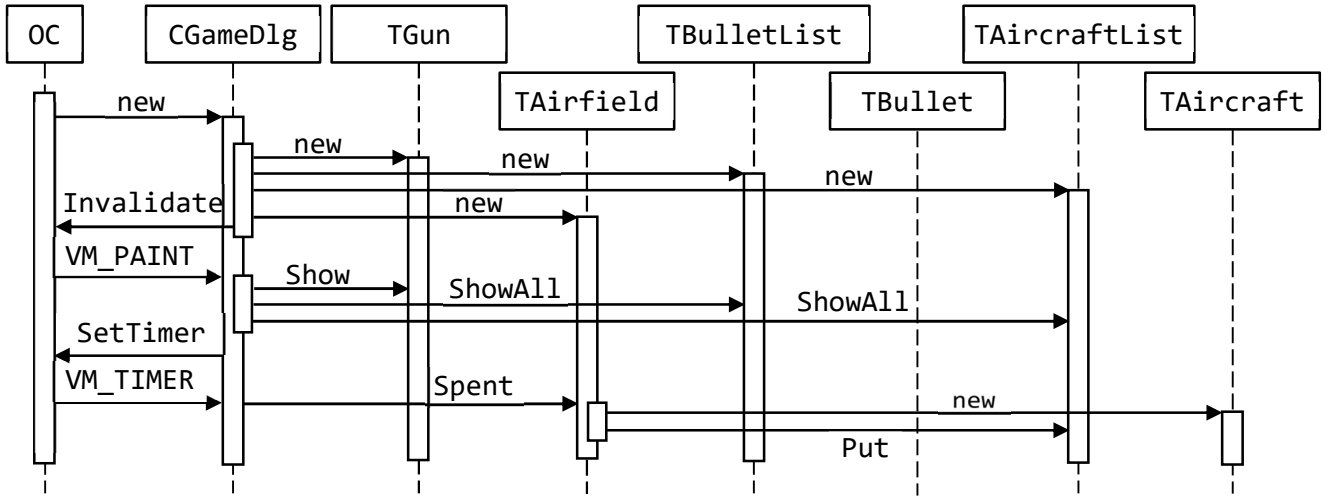

Рис. 22. Створення екземпляра застосунку

Екземпляр застосунку створює конструктор діалогу, а метод *OnInitDialog* завершує налаштування. На рис. 22 зображено послідовність повідомлень, які надсилають конструктор і цей метод: створено гармату, аеродром, списки літаків і снарядів, системний таймер. Одразу після завершення ініціалізації відбувається перемальовування вікна застосунку. Спочатку на ньому з'явиться лише гармата, адже списки літаків і снарядів – порожні. Таймер починає надсилати повідомлення одразу після створення, що може спричинити створення нового літака. Взаємодію з ним покажемо на наступних діаграмах.

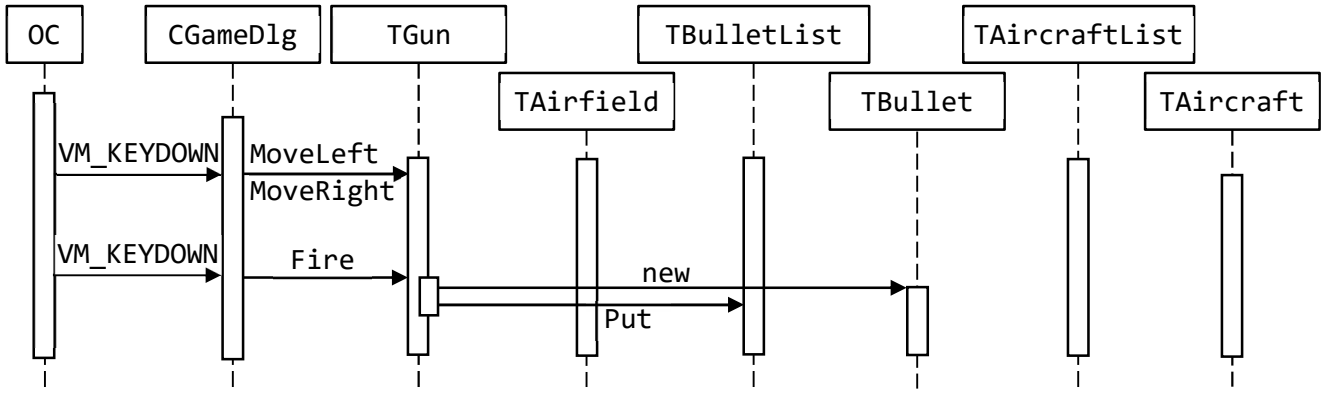

Рис. 23. Взаємодія з гарматою

Гравець керує гарматою за допомогою клавіатури. Постріл створює новий снаряд і додає його до списку.

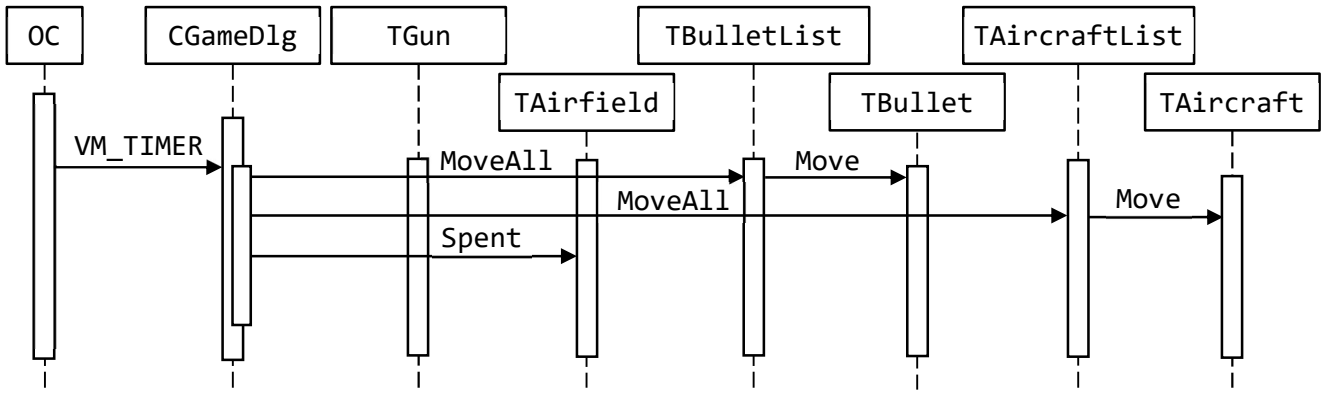

Рис. 24. Переміщення рухомих об'єктів

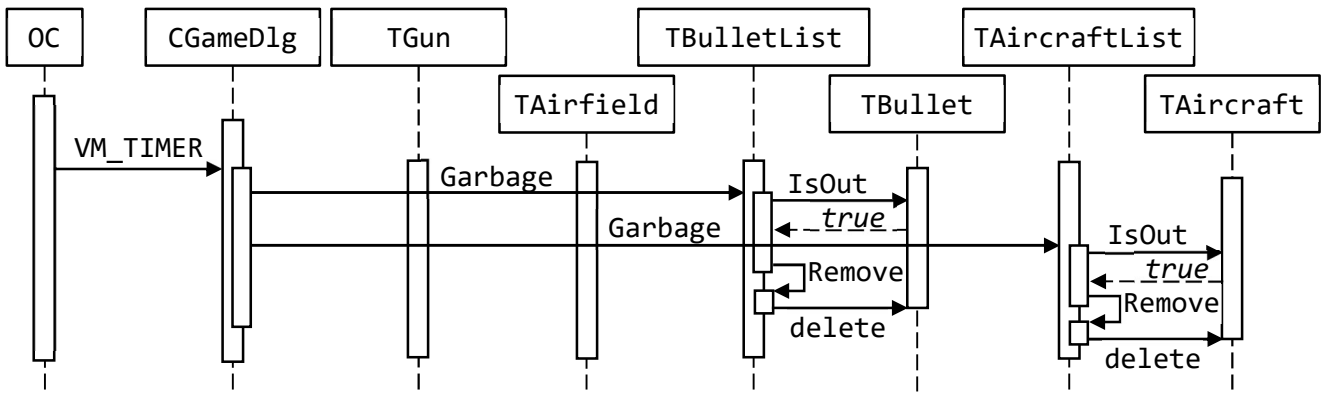

Рис. 25. Перевірка вильоту за межі екрана

Таймер керує автоматичними процесами застосунку. Метод опрацювання повідомлень від таймера має запустити переміщення рухомих об'єктів і перевірку вильоту за межі

арени та влучання, повідомити аеродром про плин часу. Демонстрація всіх цих дій на одній діаграмі була б занадто складною, тому на рис. 24 зображено лише частину, відповідальну за переміщення об'єктів, та за можливе створення нових літаків.

Список є власником рухомих об'єктів, тому саме він запитує у кожного зі своїх елементів, чи не покинув він арену (рис. 25). У випадку ствердної відповіді список викликає деструктор рухомого об'єкта і вилучає відповідну ланку.

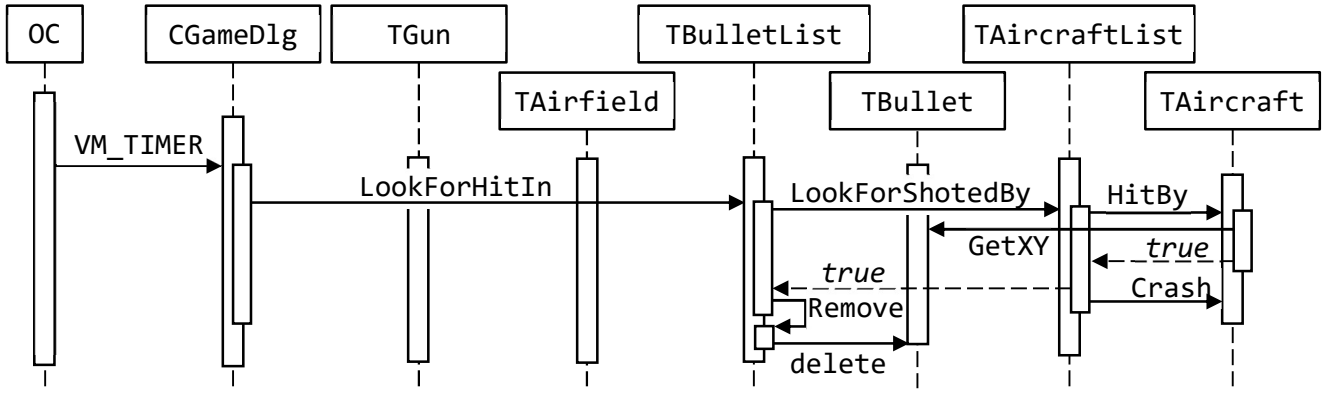

Рис. 26. Перевірка влучання

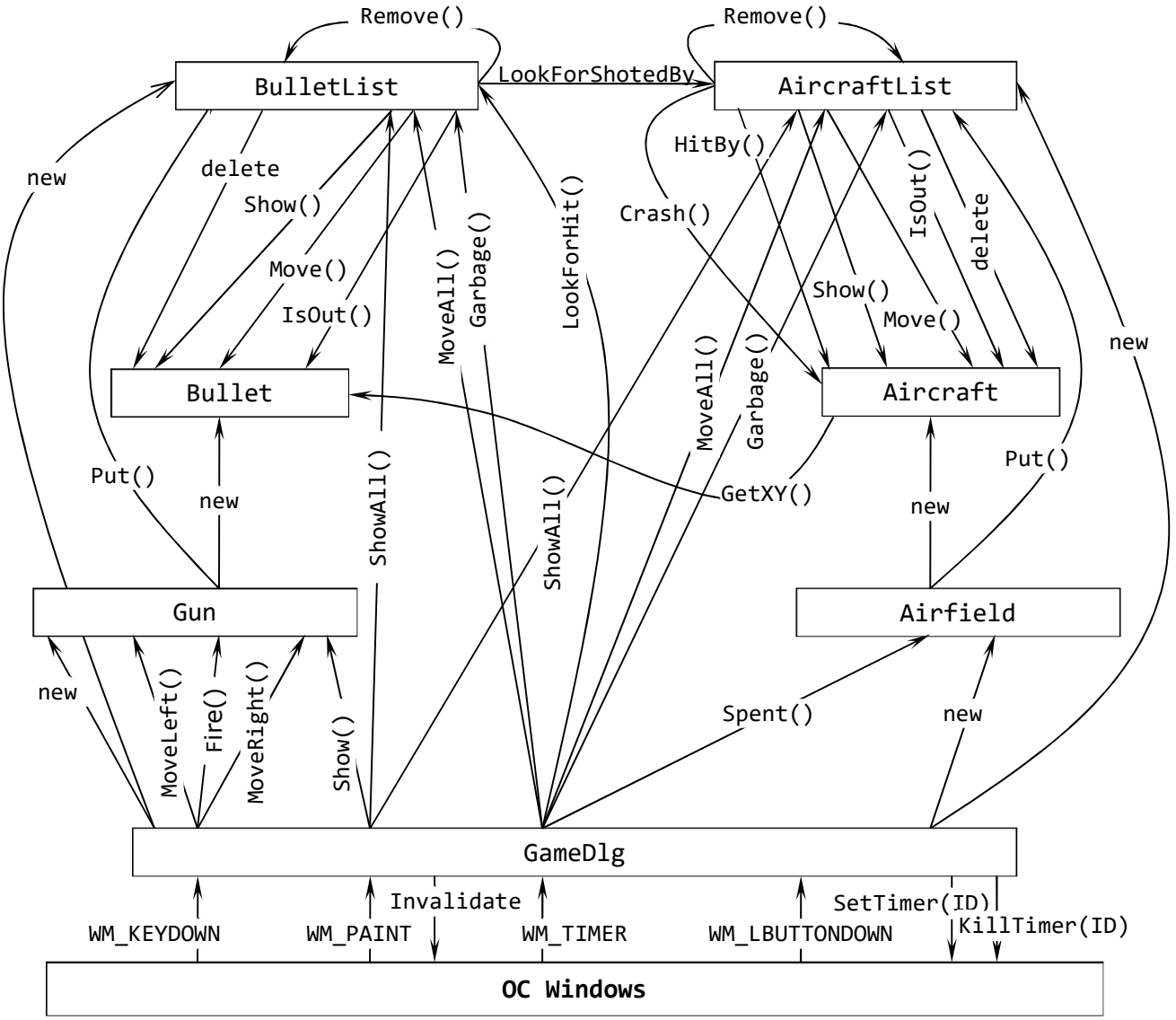

Рис. 27. Загальна схема обміну повідомленнями

Щоб перевірити влучання, діалог «просить» список снарядів (рис. 26) виконати роботу. У відповідь той кожен свій елемент передає спискові літаків із запитом про влучання. Список літаків передає отриманий снаряд кожному своєму літаку з вказівкою перевірити, чи не збили його, тому літак запитує у снаряда координати та порівнює зі своїми. У випадку збігання кожен зі списків вилучає свої елементи.

На рис. 27 зображено повну мережу сутностей програми, що взаємодіють через надсилання повідомлень. Ми розробили досить детальний проєкт застосунку. Тепер можемо перейти до написання тексту програми. Складений проєкт дає змогу розробляти ієрархії класів та окремі типи незалежно один від одного. Головне – дотримуватися описаної структури та протоколів взаємодії.

15.5. Програмна реалізація

Абстрактний базовий клас *TVisibleObject* описує спільні властивості всіх видимих об'єктів: координати, розміри, крок переміщення, посилання на растрове зображення зовнішнього вигляду, на маску і на контекст пристрою, потрібні для виведення графіки. Статичні члени класу містять ресурси, потрібні всім екземплярам класу: розміри графічного екрана, готові зображення у форматі bitmap. Єдиний метод *show* виводить бітове зображення з прозорими пікселями на контекст графічного пристрою. Клас залежить від можливостей виведення графіки бібліотеки MFC та ОС Windows.

```
class TVisibleObject abstract
{
      // "секретна" константа для виведення зображення з прозорими цятками
      enum {ROP_DstCopy = 0x00AA0029};

protected:
      int m_x, m_y;            // координати лівого верхнього кута зображення
      int m_width, m_height;   // розміри зображення
      int m_step;              // швидкість переміщення (крок за одиницю часу)
      HBITMAP m_view, m_mask;  // власне зображення (вигляд і монохромна маска)
      HDC m_hMemoryDC;         // контекст для зображення

      static int ScreenWidth;  // розміри графічного екрана -
      static int ScreenHeight; // вікна застосунку
      // графічні зображення об'єктів потрібно завантажити з ресурсів аплікації
      // і використовувати для створення екземплярів
      static HBITMAP FastView;   static HBITMAP FastMask;
      static HBITMAP MidView;    static HBITMAP MidMask;
      static HBITMAP SlowView;   static HBITMAP SlowMask;
      static HBITMAP CrashView;  static HBITMAP CrashMask;
      static HBITMAP BulletView; static HBITMAP BulletMask;
      static HBITMAP GunView;    static HBITMAP GunMask;

public:
      static void InitBitmaps(HINSTANCE);  // завантажує зображення
      // запам'ятовує розміри екрана
      static void InitScreenSize(int width, int height);

      TVisibleObject(): m_x(0),m_y(0), m_width(0),m_height(0), m_step(0),
            m_view(0),m_mask(0), m_hMemoryDC(0) {}
      TVisibleObject(int x, int y, int w, int h, int s, HBITMAP b, HBITMAP m,
            HDC dc) : m_x(x), m_y(y), m_width(w), m_height(h), m_step(s),
            m_view(b), m_mask(m), m_hMemoryDC(dc) {}
```

```
	virtual ~TVisibleObject() {}
	// усі видимі об'єкти вміють себе зображати на контексті пристрою
	void Show(CDC&) const;
};
```

Тип *TGun* – досить простий клас, оскільки всю роботу з графікою він наслідує з базового класу. Йому залишилося визначити методи переміщення та постpілу.

```
class TGun :public TVisibleObject
{
public:
	TGun() : TVisibleObject() {}
	// майже всі параметри можна одразу конкретизувати
	TGun(HDC dc) : TVisibleObject(ScreenWidth / 2 - 10, ScreenHeight - 40,
		20, 30, 5, GunView, GunMask, dc) {}
	// постріл створює новий об'єкт - снаряд
	TBullet* Fire() const
	{
		return new TBullet(m_x + 9, m_y - 4, m_hMemoryDC);
	}
	// переміщення обмежені розмірами вікна
	void MoveLeft() { m_x = (m_x >= m_step + 5) ? m_x - m_step : 5; }
	void MoveRight()
	{
		int limit = ScreenWidth - m_width - 5;
		m_x = (m_x <= limit - m_step) ? m_x + m_step : limit;
	}
};
```

У міні ієрархії рухомих об'єктів найцікавішим є тип *TAircraft*. Конструктор може наділяти екземпляри різним зовнішнім виглядом, а підбитий літак змінює не лише зовнішній вигляд, а й поведінку – починає падати, доки не «згорить». Розмаїття виглядів і поведінки можна моделювати за допомогою окремих підкласів літаків. Ми поки що не будемо цього робити, щоб не ускладнювати програму понад міру.

```
// основа для списків рухомих об'єктів, визначає поведінку рухомих об'єктів
class TMovedObject : public TVisibleObject
{
public:
	TMovedObject() :TVisibleObject() {}
	TMovedObject(int x, int y, int w, int h, int s, HBITMAP b, HBITMAP m, HDC dc)
		:TVisibleObject(x, y, w, h, s, b, m, dc) {}
	virtual void Move() abstract;
	virtual bool IsOut() const abstract;
};

// снаряд рухається догори, вміє повідомляти свої координати
class TBullet :public TMovedObject
{
public:
	TBullet() :TMovedObject() {}
	// частину параметрів конструктора можна одразу конкретизувати:
	// відомі розміри та "бітмапки" зображення/маски
	TBullet(int x, int y, HDC dc)
		: TMovedObject(x, y, 2, 4, 6, BulletView, BulletMask, dc) {}
```

```
	virtual void Move() { m_y -= m_step; }
	virtual bool IsOut() const { return m_y < 5; }
	int GetX() const { return m_x; }
	int GetY() const { return m_y; }
};

class TAircraft :public TMovedObject
{
private:
	// змінна стану: підбитий літак має інший зовнішній вигляд і рухається інакше
	int m_killed;
public:
	TAircraft() :TMovedObject(), m_killed(0) {}
	TAircraft(int y, int s, HDC dc);
	virtual void Move()
	{
		m_x += m_step;
		if (m_killed)			// підбитий літак падає
		{
			m_y += m_killed;
			m_killed *= 2;
		}
	}
	virtual bool IsOut() const
	{
		// справний літак утік, підбитий – "згорів"
		return m_x > ScreenWidth - m_width || m_killed > 32;
	}
	// перевірка влучання - порівняння координат об'єктів на збігання
	bool hitBy(const TBullet& b) const
	{
		return b.GetX() > m_x && b.GetX() + 1 < m_x + m_width &&
			b.GetY() > m_y && b.GetY() < m_y + m_height;
	}
	void Crash()	// метод зміни стану літака
	{
		m_view = CrashView;
		m_mask = CrashMask;
		m_killed = 1;
	}
};

TAircraft::TAircraft(int y, int s, HDC dc)
		:TMovedObject(1, y, 0, 0, s, 0, 0, dc), m_killed(0)
{
	// літаки бувають трьох видів:
	if (s < 3)		// повільні
	{
		m_width = 30; m_height = 16;
		m_view = SlowView; m_mask = SlowMask;
	}
	else if (s < 6)	// звичайні
	{
		m_width = 31; m_height = 18;
		m_view = MidView; m_mask = MidMask;
	}
```

```
        else              // швидкі
        {
              m_width = 29; m_height = 14;
              m_view = FastView; m_mask = FastMask;
        }
}
```

Список *TList* рухомих об’єктів – досить спеціалізований контейнер: він «уміє» зображати та рухати екраном всі свої елементи, перевіряти, чи не полишили вони його межі. Його елементами є вказівники на екземпляри *TMovedObject*. У мові програмування C++ вказівник на базовий клас поліморфний: він може містити адресу екземпляра будь-якого підкласу. Завдяки цьому до списку можна заносити і снаряди, і літаки. Як всякий зв’язний список, *TList* має методи додавання та вилучення ланок.

```
class TList
{
protected:
        struct Node  // вкладений тип для моделювання ланки списку
        {
              TMovedObject* obj;
              Node* next;
              Node(TMovedObject* ob = 0, Node* n = 0) : obj(ob), next(n) {}
        };
        Node* m_head;
public:
        // однозв'язний список з заголовною ланкою
        TList() :m_head(new Node()) {}

        virtual ~TList()
        {     // деструктор виконує глибоке очищення: вилучає і ланки, і об'єкти
              Node* victim = m_head;
              while (victim != nullptr)
              {     m_head = victim -> next;
                    delete victim -> obj; delete victim;
                    victim = m_head;
              }
        }

        // додає нові об'єкти на початок списку (одразу після заголовної ланки)
        void Put(TMovedObject* obj)
        {
              m_head->next = new Node(obj, m_head->next);
        }

        // вилучає ланку, на яку вказує pre->next та вкладений об'єкт
        void Remove(Node* pre)
        {
              Node* victim = pre->next;
              pre->next = victim->next;
              delete victim -> obj; delete victim;
        }

        // промальовує на контексті всі свої елементи
        void ShowAll(CDC& dc)
        {     // заголовну ланку пропускаємо
              Node* current = m_head->next;
```

```
            while (current != nullptr)
            {
                  current->obj->Show(dc);
                  current = current->next;
            }
      }

      // велить рухатися кожному елементу
      void MoveAll()
      {     // заголовну ланку пропускаємо
            Node* current = m_head->next;
            while (current != nullptr)
            {
                  current->obj->Move();
                  current = current->next;
            }
      }

      // вилучає зі списку ті об'єкти, які вилетіли за межі вікна
      void Garbage()
      {
            // починаємо з заголовної ланки, щоб мати адресу наступної
            Node* current = m_head;
            while (current->next != nullptr)
                  if (current->next->obj->IsOut()) this->Remove(current);
                  else current = current->next;
      }
};
```

Підкласам списків залишилося зовсім небагато: забезпечити перевірку влучання. Обидва методи працюють з об'єктами конкретного типу, *TBullet* і *TAircraft*, тому доводиться приводити тип поліморфного вказівника на рухомий об'єкт.

```
class TAircraftList :public TList
{
public:
    // всю ініціалізацію виконує батьківський конструктор
    TAircraftList() :TList() {}

    // чи не потрапив снаряд у котрийсь з літаків?
    bool LookForShotedBy(TBullet& bullet)
    {
        // заголовна ланка не містить літака – пропускаємо її
        Node* current = m_head->next;
        while (current != nullptr)      // пошук до першого влучання
            // методи HitBy та Crash є тільки в літака, потрібне приведення типу
            if (static_cast<TAircraft*>(current->obj)->HitBy(bullet))
            {
                static_cast<TAircraft*>(current->obj)->Crash();
                return true;
            }
            else current = current->next;
        return false;
    }
};
```

```
class TBulletList :public TList
{
public:
    TBulletList() :TList() {}
    // кожен снаряд може влучити в літак зі списку

    void LookForHitIn(TAircraftList& list)
    {   // розпочинаємо з заголовної ланки, щоб була змога вилучити наступну
        Node* current = m_head;
        while (current->next != nullptr)
            // спискові літаків потрібно передати саме снаряд
            if (list.LookForShotedBy(*static_cast<TBullet*>(current->next->obj)))
                this->Remove(current);
            else current = current->next;
    }
};
```

На завершення наведемо ті фрагменти класу *CGameDlg*, якими ми доповнили автоматично згенерований код.

```
class CGameDlg :public CDialog
{
      // ... автоматично згенерований код пропущено ...
protected:
      enum { TimerInterval = 15 };
      HDC m_hMemDC;              // контекст пристрою для зображення всіх об'єктів
      CRect m_client_rectangle; // придатна для малювання частина вікна

// Тут описано всі складові гри: гармату, список куль, список літаків, аеродром
      TGun gun;
      TBulletList bullet_list;
      TAircraftList aircraft_list;
      TAirfield* airfield;

public:
      // рухом куль і літаків, роботою аеродрому керує таймер
      afx_msg void OnTimer(UINT_PTR nIDEvent);
      // гарматою керує гравець за допомогою клавіатури
      afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
      //особливості MFC такі, що натискання клавіш зі стрілками треба "виловлювати"
      BOOL PreTranslateMessage(MSG* pMsg);
};
```

Метод ініціалізації завершує розпочате конструктором створення сутностей програми.

```
BOOL CGameDlg::OnInitDialog()
{
      CDialog::OnInitDialog();
      // ... автоматично згенерований код пропущено ...

      //Тут виконуємо всю додаткову ініціалізацію: налаштовуємо клас TVisibleObject
      GetClientRect(&m_client_rectangle);
      TVisibleObject::InitScreenSize(
            m_client_rectangle.Width(), m_client_rectangle.Height());
      TVisibleObject::InitBitmaps(::AfxGetInstanceHandle());
```

```
      // створюємо контекст, запускаємо таймер, створюємо гармату і аеродром.
      // Списки снарядів і літаків створив конструктор
      m_hMemDC = ::CreateCompatibleDC(NULL);
      gun = TGun(m_hMemDC);
      airfield = new TAirfield(aircraft_list, m_hMemDC);
      SetTimer(ID_TIMER, TimerInterval, NULL);
      return TRUE;
}
```

Опрацювання події малювання вікна зображає усі видимі об’єкти програми

```
void CGameDlg::OnPaint()
{
      if (IsIconic())
      {   // ... автоматично згенерований код пропущено ...    }
      else  // *** Тут зображаємо всі видимі об'єкти
      {     // спеціальний контекст, що підтримує подвійну буферизацію зображень
            CakBufferedDC dc(this);
            // зафарбовуємо фон
            CBrush bk_brush(GetSysColor(COLOR_BTNFACE));
            dc.FillRect(&m_client_rectangle, &bk_brush);
            // зображаємо рухомі об'єкти
            gun.Show(dc);
            bullet_list.ShowAll(dc);
            aircraft_list.ShowAll(dc);
      }
}
```

Усе керування програмою виконують таймер і клавіатура.

```
// *** Повідомлення від таймера потрібні багатьом:
void CGameDlg::OnTimer(UINT_PTR nIDEvent)
{
      airfield->Spent(TimerInterval);  // аеродром надсилає літаки
      aircraft_list.MoveAll();         // літаки летять
      aircraft_list.Garbage();         //   і втікають
      bullet_list.MoveAll();           // снаряди летять
      bullet_list.Garbage();           //   і самоліквідуються
      bullet_list.LookForHitIn(aircraft_list); // або влучають
      // після зміни координат об'єктів вікно треба перемалювати!!!
      Invalidate();
      CDialog::OnTimer(nIDEvent);
}

// *** Розпізнаємо, яка клавіша натиснута
void CGameDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
      switch (nChar)
      {
      case VK_SPACE: bullet_list.Put(this->gun.Fire()); break;
      case K_LEFT: gun.MoveLeft(); break;       // [ < | , ]
      case K_RIGHT: gun.MoveRight(); break;     // [ > | . ]
      case 'Q': ::PostQuitMessage(0);           // завершити виконання
      }
      CDialog::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

```
// Попередній метод не бачить клавіш зі стрілками, тому допомагаємо
BOOL CGameDlg::PreTranslateMessage(MSG* pMsg)
{
	if(pMsg->message == WM_KEYDOWN)
	{	if (pMsg->wParam == VK_LEFT) pMsg->wParam = K_LEFT; /*188*/
		else if (pMsg->wParam == VK_RIGHT) pMsg->wParam = K_RIGHT; /*190*/
	}
	return CDialog::PreTranslateMessage(pMsg);
}
```

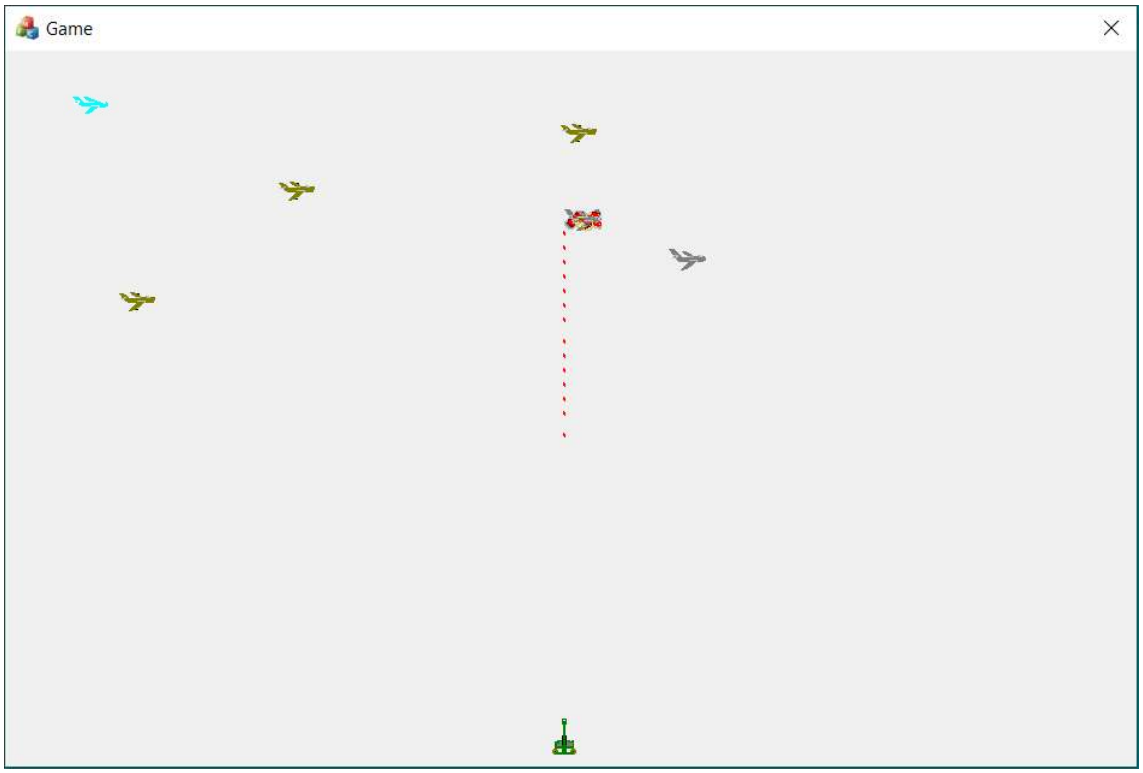

Рис. 28. Один з ігрових моментів

Настав хвилюючий момент: коли всі частини програми написано, і можна зібрати їх разом, побачити, як вони функціонують. На рис. 28 зображено один з моментів виконання готової програми.

15.6. Перебудова класу літаків

Одразу після завершення виготовлення програми розпочинається її супровід. На цьому етапі розробник усуває виявлені під час використання дефекти програми, підвищує її ефективність, поліпшує зовнішній вигляд, додає нові функціональні можливості, переносить у нове операційне середовище, застосовує нові технології тощо. Розглянемо дві можливі зміни написаної програми. Ми вже згадували про можливість розбудови ієрархії типів літаків, а тепер реалізуємо її. Згодом покажемо, як використати стандартні контейнери замість розроблених спеціалізованих списків.

Конструктор класу *TAircraft* створює літаки трьох різновидів: повільні, звичайні та швидкі. Вони відрізняються зовнішнім виглядом, але поводяться однаково, тому можуть бути екземплярами одного класу. Складнощі виникнуть, якщо спробуємо наділити їх різними можливостями, наприклад, захочемо, щоб повільні літаки могли змінювати напрям польоту, а швидкі – висоту. Тоді метод переміщення суттєво ускладниться, адже йому доведеться щоразу перевіряти різновид літака і по-різному змінювати його координати. У

літака, що змінює напрям, змінюється умова перевірки вильоту за межі екрана. Вже зараз методи *Move* та *IsOut* класу *TAircraft* враховують значення поля стану *m_killed*, щоб моделювати особливу поведінку підбитого літака. До речі, метод *HitBy* також мав би зважати на стан літака. В описаній реалізації він цього не робить, і літак можна підбити кілька разів, що трохи дивно. Написані нами методи працюють для того рівня деталізації, який ми проектували, проте їх важко доповнювати новими можливостями. Очевидно, варто задуматися над новим архітектурним рішенням.

Якщо об'єкт може перебувати в різних станах, і декілька його методів у різних станах діють по-різному, то варто використати патерн проектування, що так і називається – «Стан». *Патерном* (або шаблоном) називають сталий підхід до вирішення типової проблеми під час проектування комп'ютерних програм. Шаблон описує структуру класів, способи наслідування, оголошення методів, які варто використати, щоб збудувати гнучку надійну програмну систему. Він покликаний ізолювати місця можливих змін від решти коду. За призначенням шаблони поділяють на твірні, структурні та поведінкові. Патерни – «перлини» програмістської майстерності, які заслуговують на глибоке вивчення. Ґрунтовний їхній опис виходить далеко за межі цього посібника. Ми використаємо один з них, поведінковий патерн *Стан*, доречний у нашій програмі.

Шаблон Стан реалізують за допомогою двох об'єктів: сталого інтерфейсного об'єкта та вкладеного поліморфного об'єкта-реалізації. Різні стани описують за допомогою ієрархії класів. Базовий клас такої ієрархії зазвичай абстрактний, описує інтерфейс реалізації, а кожен підклас визначає реалізацію відповідно то того стану, який моделює. Інтерфейсний об'єкт містить об'єкт-реалізацію, всі повідомлення перенаправляє до нього: делегує йому свої обов'язки, зі зміною стану знищує попередній екземпляр реалізації та створює замість нього новий іншого типу. Зовні об'єкта-реалізації не видно, тому скидається на те, що інтерфейсний об'єкт змінив тип. Зміна поведінки довільного стану та додавання нових станів ізольовані від інших частин класу.

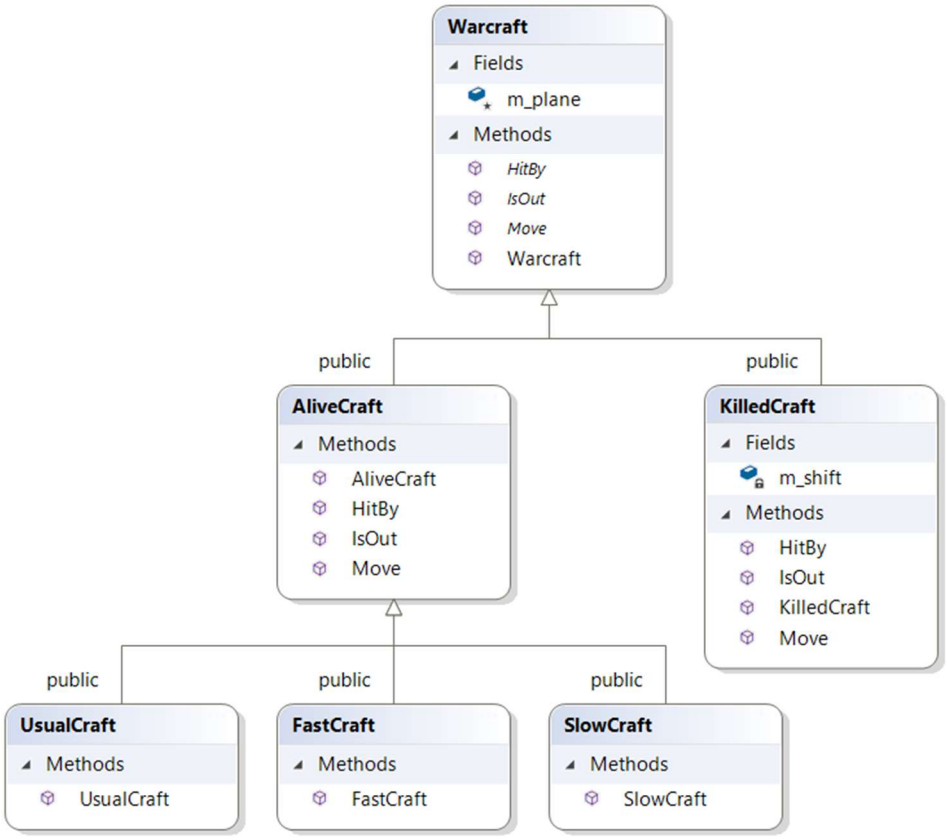

Рис. 29. Ієрархія класів стану літака

Застосуємо такий підхід для нового оголошення класу *TAircraft*. Не будемо змінювати його інтерфейс, щоб не перебудовувати програму, натомість удосконалимо реалізацію: замість поля стану *int m_killed* використаємо поліморфний вказівник на об'єкт реалізацію *Warcraft* m_plane*. Тут *Warcraft* – базовий клас ієрархії для реалізації станів. Давайте складемо її проект.

Усі дані літака – координати, вигляд, швидкість тощо – містить інтерфейсний об'єкт, тому немає потреби копіювати їх у об'єкт-реалізацію, проте реалізація потребує доступу до них. Як надати такий доступ? Клас *Warcraft* не наслідує *TMovedObject* чи *TVisibleObject*, тому надамо екземплярові реалізації посилання на свого власника. Об'єкт-інтерфейс і об'єкт-реалізація знатимуть один про одного та зможуть взаємодіяти. Проте залишається ще одна перепона: приватність даних видимого об'єкта. Щоб її подолати, зазначимо в оголошенні *TAircraft*, що класи реалізації є дружніми і можуть доступатися до приватних даних літака.

У новій архітектурі за реалізацію поведінки літака відповідає клас *Warcraft*, тому в ньому потрібно оголосити методи переміщення, перевірки вильоту та влучання. Ми могли б назвати їх довільно, проте використаємо для ясності такі самі імена, як у класі *TAircraft*. У базовому класі ці методи – абстрактні, перевизначимо їх у підкласах *AliveCraft* і *KilledCraft*. Перший узагальнює поведінку всіх діючих літаків, а другий – моделює підбитий літак. Повільні, швидкі й звичайні літаки відрізняються лише зовнішнім виглядом, тому всі вони використовують методи базового класу *AliveCraft*, а зовнішній вигляд налаштовують їхні конструктори. Клас *KilledCraft* містить додаткове поле *m_shift* для зберігання значення висоти, на яку опустився літак.

На рис. 29 зображено схему ієрархії класів реалізації станів літака, згенеровану за допомогою середовища програмування. Залишилося вирішити, де саме оголошувати ці класи. Очевидно, не варто робити їх такими ж доступними, як класи видимих об'єктів чи списків. Оголошення станів літака стосується лише реалізації *TAircraft*, тому оголошення *Warcraft* та всіх його підкласів варто розташувати в закритій частині класу *TAircraft*. На щастя, мова C++ дає змогу оголошувати вкладені класи. Окрім приховування реалізації, зможемо використовувати контекст імен класу *TAircraft* всередині ієрархії *Warcraft*, що спростить доступ до його членів, зокрема, до статичних полів. Вкладене оголошення розв'язує також проблему рекурсивного використання імен класів: для визначення *Warcraft* потрібен *TAircraft*, для визначення якого потрібен *Warcraft* і його підкласи.

Нове оголошення *TAircraft* може мати такий вигляд:

```
// використано патерн проектування "Стан"
class TAircraft :public TMovedObject
{
private:
      // *** пропущене оголошення вкладених типів Warcraft, AliveCraft,
      // *** KilledCraft та інших буде наведено згодом
      class Warcraft { ... };
      class AliveCraft { ... };
      class SlowCraft { ... };
      class UsualCraft { ... };
      class FastCraft { ... };
      class KilledCraft { ... };
      // об'єкт реалізації
      Warcraft* m_craft;

public:
      friend class AliveCraft;
      friend class KilledCraft;
      TAircraft() :TMovedObject(), m_craft(nullptr) {}
```

```
      TAircraft(int y, int s, HDC dc) :TMovedObject(1, y, 0, 0, s, 0, 0, dc)
      {
            // конструювання літака завершують конструктори об'єктів реалізації
            if (s < 3) m_craft = new SlowCraft(*this);
            else if (s < 6) m_craft = new UsualCraft(*this);
            else m_craft = new FastCraft(*this);
      }
      ~TAircraft() { delete m_craft; }
      // усю роботу виконує вкладений об'єкт реалізації
      virtual void Move() { m_craft->Move(); }
      virtual bool IsOut() const { return m_craft->IsOut(); }
      bool HitBy(const TBullet& b) const { return m_craft->HitBy(b); }
      // метод зміни стану літака змінює реалізацію
      void Crash();
      {
            delete m_craft;
            m_craft = new KilledCraft(*this);
      }
};
```

Ми свідомо пропустили текст оголошення класів реалізації, щоб наочно продемонструвати, яким стислим і виразним стало оголошення класу *TAircraft*. Якщо б ми тепер захотіли змінити, наприклад, спосіб переміщення повільного чи швидкого літака, то зміни *ніяк не вплинули б* на *TAircraft*. Додавання нового різновиду літака вплине, ймовірно, лише на конструктор.

Продемонструємо тепер пропущений фрагмент коду.

```
class Warcraft
{
protected:
      TAircraft& m_plane;     // інтерфейсний об'єкт - власник об'єкта-реалізації
public:
      Warcraft(TAircraft& p) :m_plane(p) {}
      virtual void Move() abstract;
      virtual bool IsOut() const abstract;
      virtual bool HitBy(const TBullet& b) const abstract;
};

// усі "живі" літаки поводять себе однаково
class AliveCraft :public Warcraft
{
public:
      AliveCraft(TAircraft& p) :Warcraft(p) {}
      virtual void Move() { m_plane.m_x += m_plane.m_step; }
      virtual bool IsOut() const
      {
            return m_plane.m_x > ScreenWidth - m_plane.m_width;
      }
      virtual bool HitBy(const TBullet& b) const
      {
            return b.GetX() > m_plane.m_x &&
                  b.GetX() + 1 < m_plane.m_x + m_plane.m_width &&
                  b.GetY() > m_plane.m_y &&
                  b.GetY() < m_plane.m_y + m_plane.m_height;
      }
};
```

```
// конструктори об'єктів реалізації завершують налаштування вигляду літака
class SlowCraft :public AliveCraft
{
public:
      SlowCraft(TAircraft& p) : AliveCraft(p)
      {
            m_plane.m_width = 30; m_plane.m_height = 16;
            m_plane.m_view = SlowView; m_plane.m_mask = SlowMask;
      }
};
class UsualCraft : public AliveCraft
{
public:
      UsualCraft(TAircraft& p) : AliveCraft(p)
      {
            m_plane.m_width = 31; m_plane.m_height = 18;
            m_plane.m_view = MidView; m_plane.m_mask = MidMask;
      }
};
class FastCraft : public AliveCraft
{
public:
      FastCraft(TAircraft& p) : AliveCraft(p)
      {
            m_plane.m_width = 29; m_plane.m_height = 14;
            m_plane.m_view = FastView; m_plane.m_mask = FastMask;
      }
};
// підбитий літак відрізняється виглядом і поведінкою
class KilledCraft : public Warcraft
{
private:
      int m_shift;                     // величина падіння
public:
      KilledCraft(TAircraft& p) : Warcraft(p), shift(1)
      {
            m_plane.m_view = CrashView;  // особливий вигляд
            m_plane.m_mask = CrashMask;
      }
      virtual void Move()
      {
            // за інерцією рухається вперед і падає щораз швидше
            m_plane.m_x += m_plane.m_step;
            m_plane.m_y += m_shift; m_shift *= 2;
      }
      virtual bool IsOut() const
      {
            // вилетів за край екрана чи впав (згорів)
            return m_plane.m_x > ScreenWidth – m_plane.m_width || m_shift > 64;
      }
      virtual bool HitBy(const TBullet& b) const
      {
            // повторно підбити неможливо
            return false;
      }
};
```

Якщо ви захочете тепер навчити швидкий літак змінювати висоту під час руху, то перевизначте метод *FastCraft::Move()* і, можливо, *FastCraft::IsOut()*. Такі зміни ми залишимо читачам як вправу.

15.7. Використання стандартних контейнерів

Відомо, що стандартна бібліотека STL надає контейнери, які реалізують ефективні методи вставки та вилучення значень, наприклад, *list<T>*. Ми могли б використати його для зберігання рухомих об'єктів і зекономити на створенні власних списків. Зауважимо, що час на проектування списку снарядів і списку літаків не було потрачено даремно. Ми зрозуміли й описали їхні функціональні обов'язки, протокол взаємодії. Тепер спробуємо реалізувати їх за допомогою можливостей стандартного контейнера й узагальнених алгоритмів бібліотеки. Порівняємо нові оголошення зі зробленими раніше.

Створення контейнерів. Як і раніше, списки міститимуть динамічні об'єкти, адже літаки та снаряди постійно з'являються та зникають під час виконання програми. Тому замість узагальненого типу *T* потрібно зазначити відповідний тип вказівника. У конструкторі діалогу замість

```
TBulletList bullet_list;     // спеціалізовані списки, ланки яких містять
TAircraftList aircraft_list; // вказівник на рухомий об'єкт
```

використаємо

```
list<TBullet*> list_of_bullet;         // шаблон списку конкретизовано
list<TAircraft*> list_of_aircraft;     // типом вказівника
```

Додавання та вилучення об'єктів. Для додавання об'єктів до списку ми визначили метод *Put()*, а для вилучення – *Remove()*. Стандартний контейнер має для цього власні методи: *push_back()* (або *push_front()*) та *erase()*, відповідно. Снаряди до списку додавала гармата, а літаки – аеродром. Інструкції

```
bullet_list.put(this->gun.Fire());
aircraft_list.put(this->SendAircraft())
```

замінимо на

```
list_of_bullet.push_back(this->gun.Fire());
list_of_aircraft.push_back(this->SendAircraft());
```

Вилучати об'єкти власноруч у новій версії програми нам не доведеться, бо взаємодіятимемо зі списком на вищому рівні абстракції, і відповідні методи зроблять це за нас.

Для *відображення об'єктів* список має надіслати повідомлення *Show(dc)* кожному своєму елементу. У класі *TList* ми визначали для цього метод *ShowAll*. Щоб застосувати дію до кожного елемента контейнера *list<T>*, зазвичай використовують узагальнений алгоритм *for_each*, який приймає пару ітераторів на діапазон контейнера і функціональний об'єкт, що задає цю дію. Розглянемо різні способи виклику *for_each* для відображення рухомих об'єктів на контексті графічного пристрою.

Найкоротший спосіб визначити функціональний об'єкт – записати його безпосередньо в місці виклику за допомогою лямбда-виразу. Кожному снаряду чи літаку потрібно передати графічний контекст *dc* для малювання, тому лямбда-вираз має захопити його у свій контекст імен.

```
std::for_each(list_of_bullet.begin(), list_of_bullet.end(),
      [&dc](TMovedObject* obj) { obj->Show(dc); });
```

Такий виклик *for_each* демонструє всі технічні деталі: задані межі контейнера та влаштування функціонального об'єкта, проте його важко назвати читабельним. Лямбда-вирази у C++ з'явилися порівняно недавно. Раніше всі використовували об'єкт-функції, які залишаються актуальними і нині. Об'єкт-функція може містити довільні дані, тому легко помістимо в неї графічний контекст і використаємо для виклику методу *show*.

```
class ShowObj
{
      CDC& m_dc;
public:
      showObj(CDC& dc) :m_dc(dc) {}
      void operator()(TMovedObject* a) { a->Show(m_dc); }
};

template<class T>            // шаблон функції для обох списків
void ShowAll(T& L, CDC& dc)
{
      std::for_each(L.begin(), L.end(), ShowObj(dc));
}
```

Тепер перемальовування списків об'єктів матиме цілком зрозумілий вигляд.

```
      ShowAll(list_of_bullet, dc);
      ShowAll(list_of_aircraft, dc);
```

Порівняйте з попереднім

```
      bullet_list.ShowAll(dc);
      aircraft_list.ShowAll(dc);
```

Переміщення об'єктів, як і зображення об'єктів, можна виконати за допомогою алгоритму *for_each*, а *вилiт* за межі екрана – за допомогою методу *remove_if* класу *list<T>*. Він ефективно вилучає ланки списку, що задовольняють певну умову. Деяке ускладнення зумовлено тим, що ланка містить вказівник на динамічний об'єкт, і перед вилученням ланки потрібно спершу вилучити його. Допоможе нам знову об'єкт-функція. Назвемо її *EraseObj*. Переміщення та перевірку вильоту об'єктів об'єднаємо в одну узагальнену функцію *MoveAndGarbage*.

```
class EraseObj
{
public:
      bool operator()(TMovedObject* a)
      {
            bool res = a->IsOut();
            if (res) delete a;
            return res;
      }
};

template<class T>                  // шаблон функції для обох списків
void MoveAndGarbage(T& L)
```

```
{
      std::for_each(L.begin(), L.end(),
            [](typename T::value_type x){ x->Move(); });
      L.remove_if(EraseObj());
}
```

Тепер замість

```
aircraftList.moveAll();             // літаки летять
aircraftList.garbage();             //   і втікають
bulletList.moveAll();               // снаряди летять
bulletList.garbage();               //   і самоліквідуються
```

запишемо

```
MoveAndGarbage(list_of_aircraft);   // літаки летять і втікають
MoveAndGarbage(list_of_bullet);     // снаряди летять і самоліквідуються
```

Найбільше потрудитися доведеться над *перевіркою влучання*. Знову стане в пригоді метод *remove_if* списку снарядів. Він вилучить всі ланки, які містять снаряди, що влучили, а відповідна об'єкт-функція вилучить самі снаряди, проте цього разу їй доведеться виконати набагато більше роботи: перебрати список літаків і змінити стан підбитих. Допоможе в цьому узагальнений алгоритм *find_if*.

```
// об'єкт-функція для перевірки влучання працює зі списком літаків
class Hiter
{
      list<TAircraft*>& a_list;
public:
      Hiter(list<TAircraft*>& a) :a_list(a) {}
      bool operator()(TBullet* b)
      {
            list<TAircraft*>::iterator it =
                  std::find_if(a_list.begin(), a_list.end(),
                     [b](TAircraft* a) { return a->HitBy(*b); });
            if (it != a_list.end()) // знайдено влучання
            {
                  (*it)->Crash(); // змінити стан літака
                  delete b;       // вилучити снаряд
                  return true;    // треба вилучити ланку списку
            }
            return false;
      }
};

list_of_bullet.remove_if(Hiter(list_of_aircraft));
```

замість

```
bullet_list.LookForHitIn(aircraft_list);
```

Ми завершили модифікацію програми. Варто зауважити, що використання стандартних контейнерів і алгоритмів замість власних не впливає на її швидкодію лише пришвидшує написання. А збудована ієрархія станів літака відкриває широкі можливості для доповнення

та зміни його можливостей. Сподіваємося, читачі розвинуть іграшку далі: реалізують рефері й обчислення очок, перезаряджання гармати, облік гравців і їхніх успіхів тощо.

15.8. Запитання та завдання для самоперевірки

1. Перелічіть етапи об’єктно-орієнтованого проектування.
2. Які види ієрархій притаманні складним системам? Якими засобами їх моделюють у програмах?
3. З якою метою застосовують абстракцію під час виділення сутностей предметної області задачі?
4. Яким є головний критерій класифікації сутностей? Для чого застосовувати класифікацію? Які переваги вона забезпечує у разі написання програми?
5. Пригадайте, які сутності було виділено в комп’ютерній грі. Чи всі вони були змодельовані в програмі?
6. Пригадайте, які класи було виділено під час проектування комп’ютерної гри. Перелічіть можливості базових класів, особливості підкласів.
7. Що таке діаграма класів? Які у ній використовують позначення? З якою метою?
8. Що зображає діаграма послідовностей? Які діаграми послідовностей використано у цьому розділі?
9. Яке графічне середовище було використано в програмі? Чи знаєте ви якісь інші середовища?
10. Які сутності операційної системи було безпосередньо використано в програмі?
11. Який з патернів проектування було використано в оновленій програмі для удосконалення влаштування літака? Як вплинуло його застосування на якість програмного коду?
12. Які стандартні контейнери було використано в оновленій програмі? Чи вплинуло це на швидкодію програми?
13. В оновленій програмі було використано об’єкт-функції. Пригадайте, що це таке, з якою метою його використовують, як проектують.
14. Завантажте програми за наведеним посиланням, запустіть їх на виконання.
15. Запропонуйте та випробуйте власні зміни та доповнення до програм, наприклад, реалізуйте Рефері та нарахування очок, доповніть можливості літаків та гравця.

* * *

Це останній розділ посібника. Він не претендує на роль вичерпного керівництва з розробки алгоритмів, але мав би виконати головне своє призначення – розбудити інтерес читачів до такої літератури та заохотити до творення власних алгоритмів.