Середовище розробки: бібліотека базових класів

Основною перевагою об'єктно-зорієнтованих мов програмування, таких, як C++ і Smalltalk, є високий ступінь повторного використання в добре спроектованих систем. Це означає, що для розробки кожної наступної програми потрібно набагато менше нового коду; отже, й меншу кількість коду потрібно супроводжувати і підтримувати.

Повторне використання приймає різні форми: запозичення окремих рядків коду, окремих класів або логічно пов'язаних між собою груп класів. Повтор рядків найбільш простий (хто з програмістів хоч одного разу не використав редактор для копіювання реалізації того чи іншого алгоритму з однієї програми в іншу?), Але найменш вигідний (один і той же фрагмент коду просто дублюється в різних додатках). Ми поступимо набагато краще, використовуючи об'єктно-зорієнтовані мови програмування і звертаючись до існуючих класів, модифікуючи їх або наслідуючи від них. Але ще більших успіхів можна досягти, використовуючи набори класів, організовані в інструментальні бібліотеки, - середовища розробки. Як вже зазначалося в розділі 4, під середовищем розробки розуміється сукупність класів, що надають набір послуг у певній галузі; таким чином, середовище розробки експортує ряд окремих класів і механізмів, які клієнти можуть використовувати безпосередньо або адаптувати.

Середовища розробки можуть бути досить універсальні і застосовні до широкого кола додатків. До цієї категорії належать загальні фундаментальні класи, математичні бібліотеки та бібліотеки графічного інтерфейсу користувача. Середовища розробки можуть зустрічатися і в досить вузьких предметних областях, таких, наприклад, як облік пацієнтів лікарні, торгівля текстилем, менеджмент, телефонні системи. Там, де існує сімейство програм, що вирішують подібні завдання, з'являється привід створити прикладне середовище розробки.

У цій главі ми застосуємо об'єктно-зорієнтований підхід до створення бібліотеки фундаментальних класів. У попередньому розділі нашими основними завданнями були реалізація управління системою в реальному часі і оптимальний розподіл функціональних властивостей між декількома відносно автономними і статичними об'єктами. Тут же будуть домінувати два різних аспекти: прагнення до гнучкості архітектури з широким вибором альтернатив оптимізації за часом і пам'яті, і необхідність використання загальних механізмів управління пам'яттю і синхронізацією.

Визначення меж проблемної області

У вирізці представлені детально сформульовані вимоги до бібліотеки базових класів. На жаль, ці вимоги навряд чи практично здійснимі: бібліотека, що містить абстракції, необхідні для всіх можливих програм, виявилася б занадто великою. Першим обов'язком аналітика, таким чином, є скорочення проблемної області до розумних розмірів і формулювання завдання, що піддається вирішенню. Проблема в тому вигляді, як вона представлена зараз, може потягти за собою невдачу аналізу в цілому, тому необхідно сконцентрувати увагу лише на найбільш загальних абстракціях і механізмах, придатних для широкого використання, а не намагатися зробити все для всіх (що, швидше за все, призведе до створення середовища, яке нікому нічого не дасть). Ми почнемо аналіз з огляду теорії структур даних і алгоритмів, а потім перейдемо до абстракцій, властивим стандартному програмному забезпеченню.

Перше відкриття в ході нашого аналізу - це чітке відділення структурних абстракцій (таких як черги, стеки і графи) від алгоритмічних (сортування, порівняння з зразком і пошук). Перша категорія понять добре відповідає класам. Друга категорія, на перший погляд, не піддається об'єктно-орієнтованої декомпозиції. Однак при належному підході виявляється, що вона цілком можлива: ми можемо ввести класи, екземпляри яких будуть агентами, які виконують дані функції. Як буде видно далі, об'єктивація алгоритмічних абстракцій забезпечує переваги спільності, завдяки тому, що алгоритми можна розмістити в ієрархії "узагальнення / спеціалізація".

Вимоги до бібліотеки базових класів

Бібліотека повинна містити універсальні структури даних і алгоритми, здатні задовольнити потреби більшості стандартних додатку C ++. Крім того, бібліотека повинна бути:

- Повною. Бібліотека повинна містити сімейство класів, об'єднаних узгодженим зовнішнім інтерфейсом, але з різними уявленнями, так щоб розробники могли вибрати те, семантика якого найбільш точно відповідає їх вимогам.

- Адаптовною. Всі фрагменти коду, що залежать від платформи, повинні бути виділені і ізольовані в окремі класи для забезпечення можливості локальних змін в них. Зокрема, розробник повинен мати контроль над механізмами зберігання даних і синхронізації процесів.

- Ефективною. Процедура підключення різних фрагментів бібліотеки до додатка повинна бути простою (ефективність при компіляції). Невеликі витрати оперативної пам'яті і процесорного часу на обслуговування і підключення повинні бути зведені до мінімуму (ефективність при виконанні). Бібліотека повинна забезпечувати більш надійну роботу, ніж механізми, розроблені програмістом вручну (ефективність при розробці).

- Безпечною. Кожна абстракція повинна бути безпечною з погляду типів, так щоб статичні припущення про поведінку класу могли бути забезпечені компілятором. Для виявлення порушень динамічної семантики класів повинен бути використаний механізм винятків. Виклик виключення не повинен зіпсувати стан об'єкта, що його викликав.

- Простою. Бібліотека повинна мати прозору структуру, яка дає можливість користувачеві легко знаходити і підключати до додатка її фрагменти.

- Розширюваною. Для користувача повинна бути реалізована можливість включення в бібліотеку нових класів. При цьому архітектурна цілісність середовища розробки не повинна порушуватися.

Бібліотека повинна бути відносно невеликих розмірів; треба завжди пам'ятати, що користувач з більшим бажанням займеться розробкою власного коду, ніж вивченням чужого малозрозумілого класу.

Передбачається наявність трансляторів мови C ++, що підтримують параметризрвані класи та обробку винятків. З метою забезпечення переносимості бібліотеки вона не повинна залежати від служб операційної системи.

Таким чином, першим результатом нашого аналізу буде поділ усіх абстракцій на дві категорії:

* Структури містять всі структурні абстракції.
* Інструменти містять всі алгоритмічні абстракції.

Виділимо наступні типи структур:

* Набір. Безліч різних елементів (в тому числі дублікатів).
* Безліч. Набір неповторяющихся елементів.
* Колекція. індексованих безліч елементів.
* Список. Послідовність елементів, що має початок; структурний поділ допускається.
* Стек. Послідовність елементів; елементи можуть видалятися і додаватися тільки з одного кінця.
* Черга. Послідовність елементів, до якої можна додавати елементи з одного кінця, а видаляти - з іншого.
* Дека. Послідовність елементів, до якої можна додавати і з якої можна видаляти елементи з обох кінців.
* Кільце. Послідовність елементів, до якої можна додавати і з якої можна видаляти елементи, що знаходяться на вершині кругової структури.
* Рядок. Индексируемая послідовність елементів, в якій можливі операції з підрядками.
* Асоціативний масив. Словник пар "елемент / значення".
* Дерево. Набір (що має початок - корінь дерева) вершин і ребер, які не можуть утворювати цикли і перетинатися; структурний поділ допускається.
* Граф. Безліч вершин і ребер (без виділеного початкового елемента), яке може містити цикли і перетину; структурний поділ допускається.
* Виділимо наступні типи інструментів:
* Дата / Час. Операції з датою і часом.
* Фільтри. Введення, обробка і виведення.
* Пошук за зразком. Операції пошуку послідовностей всередині інших послідовностей.
* Пошук. Операції пошуку елементів усередині структур.
* Сортування. Операції упорядкування структур.
* Утиліти. Складові операції, що базуються на базових структурних операціях.

Отже, ми визначили основні функціональні елементи нашої бібліотеки; проте ізольовані абстракції самі по собі - ще не середовище розробки. Як зазначив Вірфс-Брок: "Середовище розробки надає користувачеві модель взаємодій між об'єктами, що входять до її класів ... Щоб освоїти середовище розробки, перш за все слід вивчити методи взаємодії та відповідальності її класів". Це і є той критерій, за яким можна відрізнити середовище розробки від простого набору класів: середовище - це сукупність класів і механізмів взаємодії примірників цих класів.

Аналіз показує, що існує певний набір основних механізмів, необхідний для бібліотеки базових класів:

* семантика часу і пам'яті;
* управління зберіганням даних;
* обробка виключень;
* ідіоми ітерації;
* синхронізація при багатопотоковості.

При проектуванні системи базових класів необхідно зберігати баланс між перерахованими технічними вимогами. Якщо ми будемо намагатися вирішити кожну задачу окремо, то, швидше за все, отримаємо ряд ізольованих рішень, не пов'язаних між собою ні загальними протоколами, ні загальною концепцією, ні реалізацією.

Розглянемо опис абстракцій нашої бібліотеки з двох точок зору: користувача, який тільки оголошує об'єкти вже існуючих класів, і клієнта, який конструює власні підкласи на базі бібліотечних. При проектуванні з розрахунком на першому користувача бажано якомога сильніше обмежити доступ до реалізацій абстракцій і сконцентруватися на їх відповідальності; проектування з урахуванням запитів другого користувача передбачає відкритість деяких внутрішніх деталей реалізації, однак, не настільки, щоб стало можливим порушити фундаментальну семантику абстракції. Таким чином, доводиться відзначити деяку суперечливість основних вимог до системи.

Однією з головних проблем при роботі з великою бібліотекою є труднощі в розумінні того, які, власне, механізми вона включає в себе. Перераховані вище моделі являють собою як би душу архітектури бібліотеки: чим більше розробник знає про ці механізмих, тим легше йому буде використовувати існуючі в бібліотеці компоненти, а не складати з нуля власні. На практиці виходить так, що користувач спочатку знайомиться зі змістом і роботою найбільш простих класів, і тільки потім, перевіривши надійність їх роботи, поступово починає використовувати все більш складні класи. У процесі розробки, у міру того як починають вимальовуватися нові, притаманні предметної області користувача, абстракції, вони теж можуть додаватися до бібліотеки. Розвиток об'єктно-зорієнтованої бібліотеки - це тривалий процес, що проходить через ряд проміжних етапів.

Як вже зазначалося в попередніх розділах, класи є необхідним, але не достатнім засобом декомпозиції системи. Це зауваження повною мірою стосується і бібліотеки класів. Невпорядкований набір класів, в якому розробники шукають що-небудь корисне, - чи не найгірше з можливих рішень. Краще розбити класи на окремі категорії (рис. 1). Таке рішення дозволяє задовольнити вимогу простоти бібліотеки.

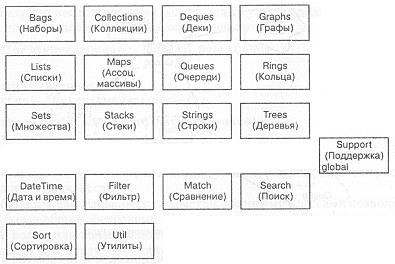


Рис. 1. Категорії класів у бібліотеці.

Сімейства класів

Третій основний принцип проектування бібліотеки полягає в побудові сімейств класів, пов'язаних ставленням успадкування. Для кожного типу структур ми створимо кілька різних класів, об'єднаних єдиним інтерфейсом, але з різними конкретними підкласами, що мають кілька різні уявлення і тому відрізняються своїм устроєм і характеристиками "час / пам'ять". Таким чином ми забезпечимо бібліотечну вимогу повноти. Розробник зможе вибрати той конкретний клас, який більшою мірою підходить для вирішення його завдання. У той же час цей клас володіє тим самим інтерфейсом, що й інші класи сімейства. Свідомо чіткий поділ абстрактного базового класу і його конкретних підкласів дозволяє користувачеві системи вибрати, скажімо, на першому етапі проектування один з класів в якості робітника, а потім, в процесі, замінити його на інший витративши на це мінімум часу і зусиль. При цьому розробник буде впевнений у нормальному функціонуванні програми, так як всі класи, що належать одному сімейству, мають ідентичний зовнішній інтерфейс і схожу поведінку. Сенс у такій організації класів полягає ще й у можливості копіювання, присвоювання і порівняння об'єктів одного сімейства навіть у тому випадку, якщо їх подання абсолютно різняться. Можна сказати, що базовий абстрактний клас містить у собі всі важливі риси абстракції.

Елементи сімейства класів являють собою різні форми абстракції. Досвід показує, що існують дві основні форми абстракцій, якими слід користуватися розробнику при створенні серйозних додатків. По-перше, це форма конкретного уявлення абстракції в оперативній пам'яті машини. Існує два варіанти такого подання: виділення пам'яті для структури з стека або виділення оперативної пам'яті з купи. Їм відповідають дві форми абстракцій: обмежена і необмежена:

® Обмежена Структура зберігається в стеку і, таким чином, має статичний розмір (відомий у момент створення об'єкта).

® Необмежена Структура зберігається в купі і її розміри можуть динамічно змінюватися.

Так як обмежена і необмежена форми абстракції мають спільні інтерфейс і поведінку, їх обидві можна представити у вигляді прямих підкласів абстрактного базового класу для кожної структури. Ми обговоримо ці та інші особливості організації даних в наступних розділах.

Як зазначалося в розділі 3, в даному випадку при проектуванні існують лише три можливих альтернативи, кожна з яких вимагає забезпечення різного рівня взаємодії між агентами, що оперують із загальними об'єктами:

послідовний;

захищений;

синхронізований.

Ми розглянемо кожен з цих варіантів більш детально в наступному розділі. Забезпечення взаємодії між абстрактним базовим класом, формами його подання і формами синхронізації породжує для кожної структури сімейство класів, подібне тому, яке наведене на рис. 9-3. Тепер можна зрозуміти, чому ми свого часу вирішили організувати бібліотеку саме у вигляді сімейств класів, а не у вигляді єдиного дерева. Це було зроблено через те, що така архітектура:

Відображає спільність різних форм.

Дозволяє здійснювати більш простий доступ до елементів бібліотеки.

Дозволяє уникнути нескінченних метафізичних суперечок про "чистому об'єктно-орієнтованому підході".

Спрощує інтеграцію системи з іншими бібліотеками.

Мікроорганізація

Семантика часу і пам'яті

З п'яти основних принципів будови бібліотеки базових класів, можливо, найбільш важливий механізм, що забезпечує клієнта альтернативної просторово-часової семантикою всередині кожного сімейства класів.

Розглянемо той спектр вимог, який повинен враховуватися при розробці бібліотеки загального призначення. На робочій станції, що володіє великим віртуальним адресним простором, користувач швидше за все буде марнувати пам'ять заради більш високої швидкодії. З іншого боку, в деяких вбудованих системах, таких, як супутник або автомобільний мотор, ресурси пам'яті часто обмежені, і розробник змушений вибирати в якості робочих ті абстракції, які використовують менше пам'яті (наприклад, виділяючи місце під дані в стеку, а не в " купі "). Раніше ми розрізнили ці дві можливості як обмежену і необмежену форми відповідно.

Необмежені форми застосовні в тих випадках, коли розмір структури не може бути передбачений, а виділення та утилізація пам'яті з купи не приводить ні до втрати часу, ні до зниження надійності (як це буває в деяких додатках, критичних за часом) [Деякі вимоги до системи можуть заборонити використання динамічно розподіляє пам'яті. Розглянемо серцевий імпульсний регулятор і можливі фатальні результати, які може викликати збирач сміття, "прокинувся" в невідповідний момент. Є системи з тривалим робочим циклом: у них навіть мінімальна витік пам'яті може дати серйозний кумулятивний ефект; вимушена перезавантаження системи через нестачу пам'яті може призвести до неприйнятної втрати функціональності]. Обмежені форми краще підходять для роботи з невеликими структурами, розмір яких досить добре передбачуваний. Врахуємо також, що динамічне виділення пам'яті менш терпимо до помилок програміста.

Таким чином, всі структури даної бібліотеки повинні бути присутніми в альтернативних варіантах; тому нам доведеться створити два низькорівневих класу підтримки, Unbounded (необмежений) і Bounded (обмежений). Завданням класу unbounded є підтримка швидко працюючого зв'язного списку, елементи якого розміщуються в пам'яті, виділеної з "купи". Це подання ефективно по швидкості, але не по пам'яті, так як кожен елемент списку повинен, крім свого значення, додатково містити покажчик на наступний елемент того ж списку. Завдання класу Bounded полягає в організації структур на базі масиву, що ефективно з точки зору пам'яті, але добитися великої продуктивності важко, так як, наприклад, при додаванні елемента в середину списку доводиться послідовно копіювати всі наступні (або попередні) елементи масиву.

Всі наші структури, однак, містять в якості елементів не посилання, а значення, що виключає виникнення посилань на тимчасові об'єкти в стеку при роботі програми. З тієї ж причини ми відмовилися від зберігання покажчиків на елементи структур, так як це викликає вкрай нестабільна поведінка системи при інстанцірованія шаблону вбудованими типами даних. Подібні питання надзвичайно істотні при проектуванні середовищ розробки, що включають в себе параметризрвані класи, так як користувач може інстанціювати шаблон довільним типом даних. При використанні посилань існують, взагалі кажучи, три випадки, і нам доведеться при створенні бібліотеки постаратися знайти певний баланс між ними.

По-перше, вбудовані типи даних можна без зусиль передавати по посиланню і копіювати. Оголосивши типи аргументу постійними засланнями, можна уникнути неприємностей, пов'язаних з появою тимчасових структур, що виникають при приведенні типів [12].

По-друге, типи даних, визначені користувачем, також можна передавати по посиланню і копіювати, але тільки в тому випадку, коли для них визначені конструктор копіювання і оператор привласнення. Посилання можна використовувати в поліморфних операціях (передаючи об'єкт похідного класу замість оголошеного при інстанцірованія), але копіювання це не буде поліморфним. В результаті об'єкт буде "зрізаний" до розмірів свого базового класу.

По-третє, при полиморфном використанні бібліотеки зустрінеться інстанцірованія шаблонів покажчиками на базовий клас. Хоча передача покажчиків за посиланням може і не поліпшити продуктивність, але копіювання покажчиків в уявлення зберігає поліморфізм похідних об'єктів.

Управління пам'яттю

Завдання управління пам'яттю виникає для необмежених форм реалізації. У цьому випадку розробник бібліотеки повинен визначити політику виділення і звільнення пам'яті з купи при здійсненні операцій над вузлами. Наївний підхід просто використовує глобальні функції new і delete, що не може забезпечити достатньої продуктивності системи. Крім того, на деяких комп'ютерних платформах управління пам'яттю вкрай ускладнено (наприклад, при наявності сегментированного адресного простору в деяких операційних системах персональних комп'ютерів) і вимагає розробки спеціальної стратегії, жорстко прив'язаною до певної операційному середовищі. Для нашої бібліотеки треба чітко виділити підсистему управління пам'яттю.

**Ітерація**

**Ітерація - це ще один архітектурний шаблон нашої бібліотеки. У розділі 3 ми вже відзначали, що итератор являє собою операцію, що забезпечує послідовний доступ до всіх частин об'єкта. Виявляється, такий механізм потрібен не тільки користувачам, він необхідний і при реалізації самої бібліотеки, зокрема, її базових класів.**

**При цьому перед нами стояв вибір: можна було визначати ітерації як частина протоколу об'єктів або створювати окремі об'єкти, відповідальні за ітеративний опитування інших структур. Ми вибрали другий підхід з двох причин:**

**• Наявність виділеного ітератора класів дозволяє одночасно проводити кілька переглядів одного і того ж об'єкта.**

**• Наявність итерационного механізму в самому класі кілька порушує його інкапсуляцію; виділення ітератора в якості окремого механізму поведінки сприяє досягненню більшої ясності в описі класу.**

**Для кожної структури визначені дві форми ітерацій. Активний итератор вимагає щораз від клієнта явного звернення до себе для переходу до наступного елементу. Пасивний итератор застосовує функцію, надану клієнтом, і, таким чином, вимагає меншої участі клієнта. Щоб забезпечити безпеку типів, для кожної структури створюються свої ітератори.**

**Кожному ітератори у момент створення ставиться у відповідність певний об'єкт. Ітерація починається з "верху" структури, що б це не означало для даної абстракції.**

Синхронізація

При розробці будь-якого універсального інструментального засобу повинні враховуватися проблеми, пов'язані з організацією паралельних процесів. В операційних системах типу UNIX, OS / 2 і Windows NT додатки можуть запускати кілька "легких" процесів ["Легким" називається процес, який виконується в тому ж адресному просторі, що й інші. На противагу їм існують "важкі" процеси; їх створює, наприклад, функція fork в UNIX. Важкі процеси вимагають спеціальної підтримки операційної системи для організації зв'язку між собою. Для C ++ бібліотека AT & T пропонує "полупереносімую" абстракцію легких процесів для UNIX. Легкі процеси безпосередньо доступні в OS / 2 і Windows NT. У бібліотеку класів Smalltalk включений клас Process, який реалізує підтримку легких процесів]. У більшості випадків класи просто не зможуть працювати в такому середовищі без спеціальної доробки: коли два завдання взаємодіють з одним і тим же об'єктом, вони повинні робити це узгоджено, щоб не зруйнувати стану об'єкта. Як вже зазначалося, існують два підходи до задачі управління процесами; вони знаходять своє відображення в існуванні захищеною і синхронізованою форм класу.

Так само, як і при управлінні пам'яттю, ми поділяємо політику синхронізації процесів і її реалізацію. З цієї причини в аргументи шаблону для кожної захищеної форми включений клас Guard (страж), відповідальний за зв'язок з локальною реалізацією класу Semaphore або його еквівалента. Аргументи шаблону для кожної з синхронізованих форм містять клас Monitor, який близький за своїми функціональними властивостями до класу Semaphore, але, як буде видно надалі, забезпечує більш високий рівень паралелізму процесів.

Основна перевага захищеної схеми - її простота. У той же час для агентів, які виробляють операції над одним і тим же об'єктом, використання даної моделі обумовлює необхідність виконання певних колективних дій. Інша особливість захищених форм полягає в тому, що вона дає можливість агентам виділяти критично важливі моменти, коли кілька операцій, вироблених над об'єктом, будуть гарантовано інтерпретуватися як одна атомарна транзакція.

Подібно механізму управління пам'яттю, сигнатура шаблону захищеної форми імпортує варта, а не перетворює його в неизменяемую характеристику. Це дозволяє користувачам ввести нову політику синхронізації. При використанні як стража зумовленого класу Semaphore, стандартна політика синхронізації передбачає, що кожному об'єкту ставиться у відповідність свій семафор. Дане рішення прийнятно тільки до тих пір, поки кількість паралельних процесів не досягне деякого критичного значення.

Альтернативний підхід передбачає можливість обслуговування одним семафором відразу декількох захищених об'єктів. Розробнику при цьому потрібно тільки створити новий клас-страж, що має той же протокол, що і semaphore (але не обов'язково є його підкласом). Цей клас може містити семафор в якості статичного члена; тоді семафор буде спільно використовуватися всіма екземплярами класу. Інстанціруя захищену форму з цим новим вартовим, розробник бібліотеки вводить нову політику, оскільки всі об'єкти інстанціювати класу користуються спільним стражем, замість виділення окремого стража кожному об'єкту. Перевага даної схеми найбільш ясно проявляється, коли новий клас-страж використовується для інстанцірованія інших структур: всі отримані об'єкти будуть працювати з одним і тим же вартовим. Таким чином, на перший погляд незначна зміна політики призводить не тільки до зменшення кількості паралельних процесів, але також дозволяє клієнту блокувати цілу групу об'єктів, незв'язаних безпосередньо. Захоплення одного об'єкта автоматично блокує доступ і до всіх інших структурам, які мають того ж варта, навіть якщо це структури різного типу.

Синхронізований клас, будучи прямим подклассом якого-небудь конкретного обмеженого або необмеженого класу, містить в собі об'єкт-монітор

За допомогою моніторів можна реалізувати два типи синхронізації:

® Одиночна Гарантує семантику структури в присутності кількох потоків управління, але з одним читаючою або одним записуючим.

® Множинна Гарантує семантику структури в присутності кількох потоків управління, з кількома читають або одним записуючим.

Агент записи змінює стан об'єкта; агенти записи викликають функції-модифікатори. Агент читання зберігає стан об'єкта; він викликає тільки функції-селектори. Як видно, множинна форма синхронізації забезпечує найвищу ступінь паралелізму процесів. Ми можемо реалізувати обидві політики у вигляді підкласів абстрактного базового класу Monitor. Обидві форми можна побудувати на основі класу Semaphore.

На відміну від захищених форм, синхронізувати класи не містять додаткових функцій-членів у порівнянні зі своїм суперкласом: вони просто перекривають однойменні всі віртуальні функції суперкласу. Семантика, що вноситься синхронізованим класом, змушує трактувати кожну таку функцію як атомарну транзакцію. У той час, як клієнти захищеного об'єкта повинні для отримання ексклюзивного доступу щоразу явно захоплювати і звільняти доступ, синхронізовані форми забезпечують ексклюзивність доступу, не вимагаючи спеціальних дій з боку своїх клієнтів.

Це досягається за допомогою механізму блокування, схема роботи якого наведена на рис. 9-11. Взаємодія моніторів з примірниками визначених класів ReadLock і WriteLock забезпечує ексклюзивність виклику кожної функції-члена. У цьому механізмі блокування використовує або семафор, або монітор в якості агента, відповідального за процес синхронізації, а сама блокування відповідає за захоплення цього агента при створенні і звільнення при видаленні.

Наша архітектура забезпечує синхронізованим формам відсутність ситуацій типу "смертельне обійми". Наприклад, операції привласнення об'єкта самому собі або порівняння його з самим собою потенційно небезпечні, оскільки вимагають блокування і лівого і правого елементів вирази, які в даному випадку є одним і тим же об'єктом. Будучи створений, об'єкт не може змінити свою ідентичність, тому тести на самоідентичність виконаються до блокування будь-якого об'єкта. Саме тому описаний раніше оператор присвоювання operator = включав таку перевірку, як показує наступна скорочений запис

Будь-які функції-члени, серед аргументів яких є екземпляри класу, до якого вони належать, повинні проектуватися так, щоб забезпечувалася коректна схема блокування цих аргументів. Наше рішення базується на полиморфизме двох службових функцій, lock і unlock, визначених у кожному абстрактному базовому класі. Кожен абстрактний базовий клас за замовчуванням містить заглушку для цих двох функцій; синхронізовані форми забезпечують захоплення і звільнення аргументу.

Проектування інтерфейсу класів

Після того, як вироблені основні принципи побудови архітектури системи, що залишається робота проста, але найчастіше досить нудна і втомлює. Наступний етап полягатиме у реалізації трьох або чотирьох сімейств класів (таких, як чергу, безліч і дерево) відповідно до обраної архітектурою, і в подальшому їх тестуванні в декількох додатках (Вірфс-Брок вважає, що необхідно тестувати середовище розробки принаймні на трьох додатках, щоб перевірити правильність стратегічних і тактичних рішень ).

Одне з найбільш цікавих властивостей середовищ розробки полягає в тому, що, в разі вдалої реалізації, вони прагнуть набрати певну критичну масу функціональності і адаптованості. Іншими словами, якщо ми правильно вибрали основні абстракції і наділили бібліотеку поруч добре взаємодіють між собою механізмів, то незабаром виявимо, що клієнти використовують наш продукт для вирішення тих завдань, про які розробники середовища і не підозрювали. Після того, як визначилися основні схеми використання середовища, має сенс зробити їх формальною частиною самої бібліотеки. Ознакою правильності конструкції середовища розробки є можливість впровадження нових моделей поведінки за допомогою повторного використання вже існуючих властивостей продукту і без порушення принципів його архітектури.

Однією з таких завдань є проблема часу життя об'єктів. Може зустрітися клієнт, який не хоче або не потребує використання повно масштабної об'єктно-орієнтованої бази даних, а планує лише час від часу зберігати стан таких структур, як черги і множини, щоб мати можливість отримати їх стан при наступному виклику з тієї ж програми або з іншої програми. Беручи до уваги те, що подібні вимоги можуть виникати досить часто, має сенс доповнити нашу бібліотеку простим механізмом збереження об'єктів.

Зробимо два допущення, що стосуються цього механізму. По-перше, клієнт повинен забезпечити потоки, в які об'єкти будуть записуватися і зчитуватися. По-друге, клієнт зобов'язаний забезпечити об'єктам поведінку, необхідне для направлення в потік.

Для створення такого механізму є два альтернативних підходи. Можна побудувати клас-домішка, що забезпечує семантику "долгожітіе"; саме такий підхід реалізований в багатьох об'єктно-орієнтованих базах даних. Як альтернативу можна створити клас, екземпляри якого виступають в якості агентів, відповідальних за перенаправлення різних структур в потік. Для того, щоб обгрунтувати наш вибір, спробуємо оцінити переваги та недоліки того і іншого підходу.

Як виявилося, для вибраного дуже простого механізму зберігання домішка не зовсім підходить (зате вона дуже добре вписується в архітектуру справжньою об'єктно-орієнтованої бази даних). При використанні домішки користувач повинен сам додати її до свого класу, часто перевизначивши при цьому деякі службові функції класу-домішки. У нашому випадку, для такого простого механізму це виявиться неефективним, тому що користувачеві буде легше розробити свої кошти, ніж доопрацьовувати бібліотечні. Таким чином, ми схиляємося до другого рішенням, яке вимагатиме від користувача лише створення екземпляра вже існуючого класу.

Завдання побудови середовища розробки є досить складною. При конструюванні основних ієрархій класів необхідно враховувати різні, часто суперечливі вимоги до системи. Намагайтеся зробити вашу бібліотеку як можна більш гнучкою: ніколи не можна передбачити, як саме спробує її використовувати розробник. Також дуже важливо зробити її якомога більш незалежною від програмного середовища - так легше буде використовувати її спільно з іншими бібліотеками. Пропоновані абстракції повинні бути якомога простішими, ефективними і зрозумілими розробнику. Найелегантніші рішення ніколи не будуть використані, якщо терміни їх освоєння перевищать час, необхідний програмісту для вирішення проблеми своїми силами. Сказати, що ефект досягнутий, можна буде тільки коли стане видно, що ваші абстракції використовуються повторно багато разів. Тобто, коли розробник відчув переваги їх використання і не винаходить велосипед, а зосереджує увагу на тих особливостях завдання, які ще ніким не були вирішені.