Інститут фізико-технічних та комп’ютерних наук

Відділ комп’ютерних технологій

Кафедра програмного забезпечення комп’ютерних систем

КУРСОВИЙ РЕФЕРАТ

на тему:

**ОБ′ЄКТНО-ЗОРІЄНТОВАНЕ ПРОГРАМУВАННЯ**

Студента 2 курсу 243 групи

напряму підготовки

«Програмна інженерія»

Джурана А.С.

Керівник

Доцент кафедри МПУіК,

Лазорик В.В.

Національна шкала \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_Оцінка: ECTS \_\_\_\_\_

**Члени комісії** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

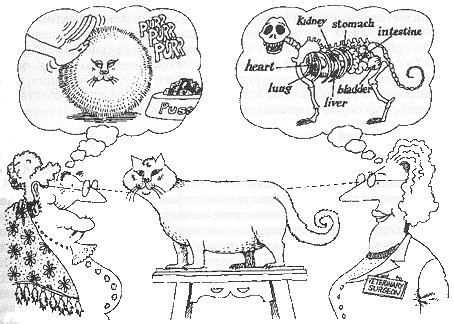
(підпис) (прізвище та ініціали)

Чернівці – 2015

****ОСНОВНІ ПРИНЦИПИ ОзП****

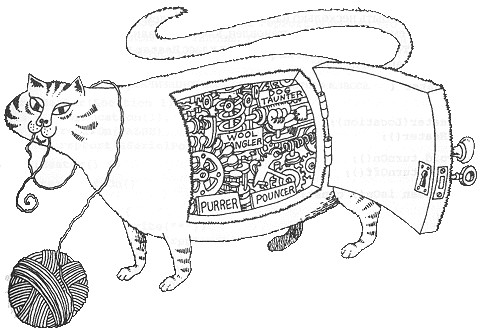
* **абстрагування**
* **інкапсуляція**
* **модульність**
* **ієрархія**
* **Додаткові принципи:**
* **типізація**
* **паралелізм**
* **стійкість**

Абстрагування **-** це процес виділення найбільш істотних характеристик деякого об'єкта, що відрізняють його від усіх інших видів об'єктів, важливих з точки зору подальшого розгляду та аналізу, та ігнорування менш важливих або незначних деталей.

**Об'єкти і класи - основні абстракції предметної області.**

Абстракція фокусується на суттєвих з точки зору спостерігача характеристиках об'єкта.

Інкапсуляція **-** це процес відділення один від одного елементів об'єкта, що визначають його пристрій і поведінку; інкапсуляція служить для того, щоб ізолювати контрактні зобов'язання абстракції від їх реалізації.



Інкапсуляція приховує деталі реалізації об'єкта.

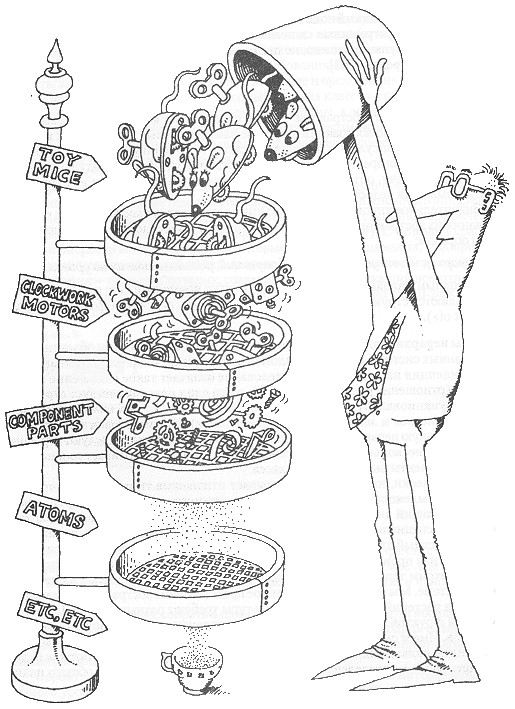
Модульність - це властивість системи, пов'язана з можливістю її декомпозиції на ряд внутрішньо сильно зчеплених, але слабо пов'язаних між собою підсистем (частин).

Модульність знижує складність системи, дозволяючи виконувати незалежну розробку її окремих частин.



Модульность позволяет хранить абстракции раздельно.

Ієрархія- це впорядкування абстракцій, розташування їх за рівнями.

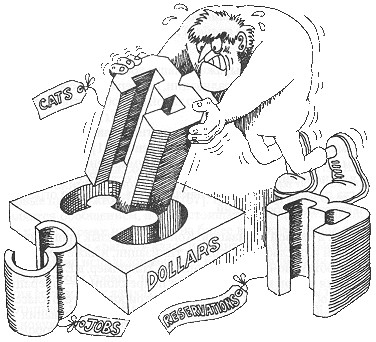


Абстракції утворюють ієрархію.

Типізація - спосіб захиститися від використання об'єктів одного класу замість іншого, або, принаймні, керувати таким використанням**.**

Тип **-** точна характеристика деякої сукупності однорідних об'єктів, що включає структуру і поведінку.

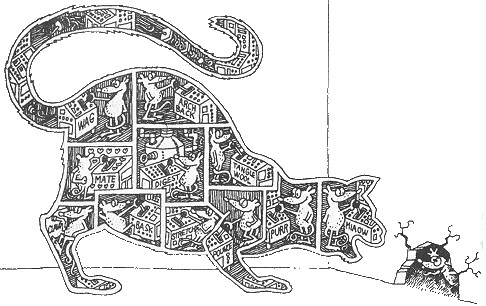
При строгій типізації (наприклад, у мові Оберон) забороняється використання об'єктів невірного типу, потрібно явне перетворення до потрібного типу. В мовах з менш суворою типізацією такого роду заборони ослаблені. Зокрема, допускається поліморфізм - багатозначність імен. Один з проявів поліморфізму, використання об'єкт підтипу (спадкоємця) в ролі об'єкта супертипа (предка).



Строга типізація запобігає змішуванню абстракцій.

Паралелізм **-** це властивість, що відрізняє активні об'єкти від пасивних.

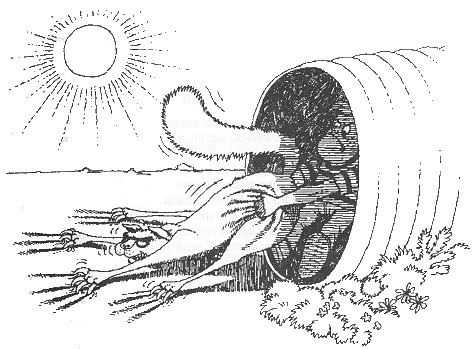
Паралелізм **-** наявність в системі декількох потоків керування одночасно. Об'єкт може бути активний, тобто може породжувати окремий п отік управління. Різні об'єкти можуть бути активні одночасно.



Паралелізм дозволяє різним об'єктам діяти одночасно.

**Збереженість (стійкість) -** здатність об'єкта існувати в часі, переживаючи породивший його процес, і (або) в просторі, переміщаючись зі свого первісного адресного простору**.**

**Стійкість -** здатність об'єкта зберігати своє існування в часі та / або просторі (адресному, зокрема при переміщенні між вузлами обчислювальної системи). Зокрема, стійкість об'єктів може бути забезпечена за рахунок їх зберігання в базі даних.



Збереженість підтримує стан і клас об'єкта в просторі і часі.

****Основні поняттия об′ектно-зорієнтованного підходу або Елементи об′ектної моделі****

“ Об'єкт в ОЗП - це сутність, здатна зберігати свій стан (інформацію) і забезпечує набір операцій (поведінку) для перевірки і зміни цього стану.”

**Об'єкт - відчутна сутність (tangible entity) - предмет або явище (процес), що мають чітко виражені межі, індивідуальність і поведінку.**

Будь-який об'єкт володія властивостями, поведінкою й індивідуальністю.

Стан об'єкта визначається значеннями його властивостей (атрибутів) і зв'язками з іншими об'єктами, він може змінюватися з часом.

Поведінкавизначає дії об'єкта та його реакцію на запити від інших об'єктів. Поведінка представляється за допомогою набору повідомлень, які сприймаються об'єктом (операцій, які може виконувати об'єкт).

Індивідуальність - це властивості об'єкта, що відрізняють його від усіх інших об'єктів.

Структура і поведінку схожих об'єктів визначають загальний для них клас.

Об'єкт в JavaScript створюється за допомогою функції Object.create. Ця функція з батьківського класу та опціонального набору властивостей створює нову сутність

Прототип **-** це об'єкт-зразок, за образом і подобою якого створюються інші об'єкти. Об'єкти-копії можуть зберігати зв'язок з батьківським об'єктом, автоматично наслідуючи зміни в прототипі; ця особливість визначається в рамках конкретної мови.

Клас - це безліч об'єктів, пов'язаних спільністю властивостей, поведінки, зв'язків і семантики. Будь-який об'єкт є екземпляром класу. Визначення класів і об'єктів - одна з найскладніших завдань об'єктно-зорієнтованого проектування.

Клас (class) **-** це група даних і методів (функцій) для роботи з цими даними. Це шаблон. Об'єкти з однаковими властивостями, тобто з однаковими наборами змінних стану і методів, утворюють клас.

Конструктор класу - спеціальний блок інструкцій, що викликається при створенні об'єкта.

var s = new String ();

Деструктор класу **-** спеціальний метод класу, який слугує для деініцілізації об'єкта (наприклад, звільнення пам'яті).

Атрибут- йменована властивість класу, що визначає інтервал допустимих значень, які можуть приймати екземпляри даної властивості. Атрибути можуть бути приховані від інших класів, це визначає видимість атрибута: рublic (загальний, відкритий); private (закритий, секретний); protected (захищений).

Необхідну поведінку системи реалізується через взаємодію об'єктів. Взаємодія об'єктів забезпечується механізмом пересилання повідомлень. Певний вплив одного об'єкта на інший з метою викликати відповідну реакцію називається операцією або посилкою повідомлення. Повідомлення може бути надіслано тільки протягом з'єднання між об'єктами. У термінах програмування з'єднання між об'єктами існує, якщо один об'єкт має посилання на інший.

**Дескриптор -** це атрибут об'єкта зі зв'язаною поведінкою (англ. Binding behavior), тобто такий, чия поведінка при доступі перевизначається методами протоколу дескриптора.

**Операція -** це послуга, яку можна запросити у будь-якого об'єкта даного класу. Операції реалізують поведінку екземплярів класу. Опис операції включає чотири частини: ім'я; список параметрів; тип значення; видимість.

Реалізація операції називається методом.

**Метод** - це функція або процедура, що належить певному класу або об'єкту.

Розрізняють прості методи і статичні методи (методи класу):

прості методи мають доступ до даних об'єкта (конкретного екземпляра даного класу), статичні методи не мають доступу до даних об'єкта і для їх використання не потрібно створювати екземпляри (даного класу).

Методи предаставляють інтерфейс, за допомогою якого здійснюється доступ до даних об'єкта деякого класу, тим самим, забезпечуючи інкапсуляцію даних.

Залежно від того, який рівень доступу надає той чи інший метод, виділяють:

**відкритий (public) інтерфейс** - загальний інтерфейс для всіх користувачів даного класу;

**захищений (protected) інтерфейс -** внутрішній інтерфейс для всіх спадкоємців даного класу;

**закритий (private) інтерфейс -** інтерфейс, доступний тільки зсередини даного класу.

Такий поділ інтерфейсів дозволяє зберігати незмінним відкритий інтерфейс, але змінювати внутрішню реалізацію.

Поліморфізм - здатність приховувати безліч різних реалізацій під єдиним загальним ім'ям або інтерфейсом.

Поняття поліморфізму може бути інтерпретовано, як здатність об'єкта належати більш ніж одному типу.

Інтерфейс **-** це сукупність операцій, що визначають набір послуг класу або компонента. Інтерфейс не визначає внутрішню структуру, всі його операції відкриті.

Компонент -це відносно незалежна і частина системи, яку можна замінити, що виконує чітко визначену функцію в контексті заданої архітектури.

Компонент являє собою фізичну реалізацію проектної абстракції і може бути: компонентом вихідного коду (cpp-шник); компонентом часу виконання (dll, ActiveX і т. п.); виконуваний компонентом (exe-шник). Компонент забезпечує фізичну реалізацію набору інтерфейсів. Компонентна розробка (component-based development) являє собою створення програмних систем, що складаються з компонентів (не плутати з об'єктно-зорієнтованим програмуванням (ОЗП).

Компонентна розробка **-** технологія, що дозволяє об'єднувати об'єктні компоненти в систему.

Пакет **-** це загальний механізм для організації елементів у групи. Це елемент моделі, який може включати інші елементи. Кожен елемент моделі може входити тільки в один пакет.

Пакет являється:

**-**Засобом організації моделі в процесі розробки, підвищення її керованості і читаності;

-одиницею управління конфігурацією.

Підсистема- це комбінація пакетів (може включати інші елементи моделі) і класу (володіє поведінкою). Підсистема реалізує один або більше інтерфейсів, що визначають її поведінку. Вона використовується для представлення компонента в процесі проектування.

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).
* Множина. Набір елементів, які не повторюються.
* 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. Індексуюча 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:

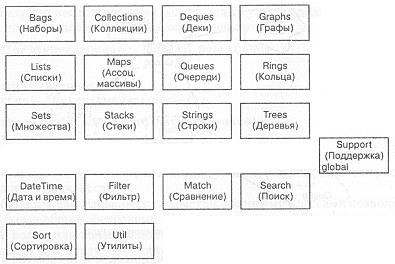
* семантика часу і пам'яті;
* управління зберіганням даних;
* обробка виключень;
* ідіоми ітерації;
* синхронізація при багатопотоковості.

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 механізми, 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, 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 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.

Проектування інтерфейсу класів

Після того, як визначені основні принципи побудови архітектури системи, 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, є 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.

Одним з таких завдань є проблема часу життя об'єктів. Може зустрітися клієнт, який не хоче або не потребує використання повно масштабної об'єктно-зорієнтованої 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, необхідну для направлення в потік.

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

Як виявилося, для вибраного дуже простого механізму зберігання додаток не зовсім підходить (зате він дуже добре вписується в архітектуру справжньої об'єктно-зорієнтованої бази даних). При використанні додатку користувач повинен сам додати її до свого класу, часто перевизначивши при цьому деякі службові функції класу-додатку. 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 складним. 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.