Лекція 1. Моделювання систем.

Невдалі проекти закінчуються крахом в силу різних причин, а ось успішні, як правило, мають багато спільного. Хоча успіх програмного проекту забезпечується безліччю різних додатків, одним із загальних є застосування моделювання.

Моделювання - це усталена і повсюдно прийнята інженерна методика. Ми будуємо архітектурні моделі будівель, щоб допомогти їхнім майбутнім мешканцям у всіх подробицях уявити собі готовий продукт. Іноді вдаються навіть до математичного моделювання будівель, щоб врахувати вплив сильного вітру або землетрусу.

Моделювання застосовується не тільки в будівництві. Навряд чи ви зумієте налагодити випуск нових літаків або автомобілів, не зазнавши свій проект на моделях: від комп'ютерних моделей і креслень до фізичних моделей в аеродинамічній трубі, а потім і повномасштабних прототипах. Електричні прилади, від мікропроцесорів до телефонних комутаторів, також вимагають моделювання для кращого розуміння системи та організації спілкування з колегами. Навіть в кінематографії успіх фільму неможливий без попередньо написаного сценарію (теж своєрідна форма моделі). У соціології, економіці або менеджменті ми також звертаємося до моделювання, яке дозволяє перевірити наші теорії і випробувати нові ідеї з мінімальним ризиком і витратами.

Отже, що ж таке модель? Попросту кажучи, вона є спрощеним уявленням реальності. Модель - це креслення системи: в неї може входити як детальний план, так і більш абстрактне уявлення системи "з висоти пташиного польоту". Хороша модель завжди включає елементи, які суттєво впливають на результат, і не включає ті, які малозначущі на даному рівні абстракції. Кожна система може бути описана з різних точок зору, для чого використовуються різні моделі, кожна з яких, отже, є семантично замкнутої абстракцією системи. Модель може бути структурною, підкреслюючи організацію системи, або поведінковою, тобто визначаючи її динаміку.

Моделювання дозволяє вирішити чотири різних завдання:

- візуалізувати систему в її поточному або бажаному для нас стані;
- визначити структуру або поведінку системи;
- отримати шаблон, що дозволяє потім сконструювати систему;
- документувати прийняті рішення, використовуючи отримані моделі.

Сприйняття людиною складних сутностей обмежена. Моделюючи, ми звужуємо проблему, загострюючи увагу в даний момент тільки на одному аспекті. По суті, цей підхід є не що інше, як принцип " розділяй і володарюй ", який Едсгер Дейкстра проголосив багато років тому: складну задачу легше вирішити, якщо розділити її на кілька менших. Крім того, моделювання підсилює можливості людського інтелекту, оскільки правильно вибрана модель дає можливість створювати проекти на більш високих рівнях абстракції.

Сказати, що моделювання має сенс, ще не означає, що воно абсолютно необхідно. І дійсно, багато досліджень показують, що в більшості компаній, що розробляють програмне забезпечення, моделювання застосовують рідко або зовсім не застосовують. Чим простіше проект, тим менш імовірно, що в ньому буде використано формальне моделювання.

Від моделювання може виграти будь-який проект. Навіть при створенні одноразових програм, коли найчастіше буває корисніше викинути невідповідний код через переваг у швидкості розробки, які дають мови візуального програмування, моделювання допоможе колективу розробників краще представити план системи, а значить, виконати проект швидше і створити саме те, що мав на увазі початковий задум. Чим складніше проект, тим більше ймовірно, що через відсутність моделювання він потерпить невдачу чи буде створено не те, що потрібно. Усі корисні та цікаві системи з плином часу зазвичай ускладнюються. Нехтуючи моделюванням на самому початку створення системи, ви, можливо, гірко пошкодуєте про це, коли буде вже занадто пізно.

Принципи моделювання

Моделювання має багату історію в усіх інженерних дисциплінах. Тривалий досвід його використання дозволив сформулювати чотири основні принципи.

По-перше, вибір моделі справляє визначальний вплив на підхід до вирішення проблеми і на те, як виглядатиме це рішення. Інакше кажучи, підходите до вибору моделі вдумливо. Правильно вибрана модель висвітить найпідступніші проблеми розробки і дозволить проникнути в саму суть завдання, що при іншому підході було б просто неможливо. Неправильна модель заведе вас у глухий кут, оскільки увага буде загострюватися на несуттєвих питаннях.

Другий принцип формулюється так: кожна модель може бути втілена з різним ступенем абстракції.

При будівництві хмарочоса може виникнути необхідність показати його з висоти пташиного польоту, наприклад, щоб з проектом могли ознайомитися інвестори. В інших випадках, навпаки, потрібно саме детальний опис - допустимо, щоб показати який-небудь складний вигин труби або незвичайний елемент конструкції.

Те ж відбувається і при моделюванні програмного забезпечення. Іноді проста та швидко створена модель користувальницького інтерфейсу - найкращий варіант. В інших випадках доводиться працювати на рівні бітів, наприклад коли ви специфіцируете міжсистемні інтерфейси або боретеся з вузькими місцями в мережі. У будь-якому випадку кращою моделлю буде та, яка дозволяє вибрати рівень деталізації в залежності від того, хто і з якою метою на неї дивиться. Для аналітика або кінцевого користувача найбільший інтерес представляє питання "що", а для розробника - питання "як". В обох випадках необхідна можливість розглядати систему на різних рівнях деталізації в різний час.

Третій принцип: кращі моделі - ті, що ближче до реальності.

Фізична модель будівлі, яка веде себе не так, як виготовлена з реальних матеріалів, має лише обмежену цінність. Математична модель літака, для якої передбачаються ідеальні умови роботи та бездоганна складання, може і не володіти деякими характеристиками, властивими справжньому виробу, що в ряді випадків призводить до фатальних наслідків. Найкраще, якщо ваші моделі будуть у всьому співвідноситися з реальністю, а там, де зв'язок слабшає, повинно бути зрозуміло, в чому полягає відмінність і що з цього випливає. Оскільки модель завжди спрощує реальність, завдання в тому, щоб це спрощення не спричинило за собою будь-які істотні втрати.

Повертаючись до програмного забезпечення, можна сказати, що "ахіллесова п'ята" структурного аналізу - невідповідність прийнятої в ньому моделі і моделі системного проекту. Якщо цей розрив не буде усунений, то поведінка створеної системи з часом почне все більше відрізнятися від задуманого. При об'єктно-орієнтованому підході можна об'єднати всі майже незалежні представлення системи в єдине семантичне ціле.

Четвертий принцип полягає в тому, що не можна обмежуватися створенням тільки однієї моделі. Найкращий підхід при розробці будь нетривіальною системи - використовувати сукупність декількох моделей, майже незалежних один від одного.

Залежно від природи системи деякі моделі можуть бути важливіше інших. Так, при створенні систем для обробки великих обсягів даних важливіші моделі, які звертаються до точки зору статичної проектування. У додатках, орієнтованих на інтерактивну роботу користувача, на перший план виходять подання з точки зору статичних і динамічних прецедентів. У системах реального часу найбільш істотними будуть вистави з точки зору динамічних процесів. Нарешті, в розподілених системах, таких як Web -додатки, основну увагу потрібно приділяти моделям реалізації та розгортання.

Об'єктне моделювання

Інженери-будівельники створюють величезну кількість моделей. Найчастіше це структурні моделі, що дозволяють візуалізувати і специфікувати частини системи і те, як вони співвідносяться один з одним. Іноді, в особливо критичних випадках, створюються також динамічні моделі - наприклад, якщо треба вивчити поведінку конструкції при

землетрусі. Ці два типи розрізняються по організації і по тому, на що в першу чергу звертається увага при проектуванні.

При розробці програмного забезпечення теж існує кілька підходів до моделювання. Найважливіші з них - алгоритмічний і об'єктно-орієнтований.

Алгоритмічний метод представляє традиційний підхід до створення програмного забезпечення. Основним будівельним блоком є процедура або функція, а увага приділяється перш за все питань передачі управління і декомпозиції великих алгоритмів на менші. Нічого поганого в цьому немає, якщо не рахувати того, що системи не дуже легко адаптуються. При зміні вимог або збільшенні розміру програми (що відбувається нерідко) супроводжувати їх стає складніше.

Найбільш сучасним підходом до розробки програмного забезпечення є об'єктно-орієнтований. Тут в якості основного будівельного блоку виступає об'єкт або клас. У самому загальному сенсі об'єкт - це сутність, зазвичай видобувають із словника предметної області або рішення, а клас є описом безлічі однотипних об'єктів. Кожен об'єкт має ідентичністю (його можна перелічити чи якось по-іншому відрізнити від інших об'єктів), станом (звичайно з об'єктом бувають пов'язані деякі дані) і поведінкою (з ним можна щось робити або він сам може щось робити з іншими об'єктами).

Як приклад давайте розглянемо просту трирівневу архітектуру білінгової системи, що складається з інтерфейсу користувача, програмного забезпечення проміжного шару та бази даних. Інтерфейс містить конкретні об'єкти - кнопки, меню і діалогові вікна. База даних також складається з конкретних об'єктів, а саме таблиць, що представляють сутності предметної області: клієнтів, продукти і замовлення. Програми проміжного шару включають такі об'єкти, як транзакції і бізнес-правила, а також більш абстрактні уявлення сутностей предметної області (клієнтів, продуктів і замовлень).

Об'єктно-орієнтований підхід до розробки програмного забезпечення є зараз переважаючим просто тому, що він продемонстрував свою корисність при побудові систем в самих різних областях будь-якого розміру та складності. Крім того, більшість сучасних мов програмування, інструментальних засобів і операційних систем є в тій чи іншій мірі об'єктно-орієнтованими, а це дає вагомі підстави судити про світ у термінах об'єктів. Об'єктно-орієнтовані методи розробки лягли в основу ідеології складання систем з окремих компонентів; як приклад можна назвати такі технології, як JavaBeans і CQM.

Основні положення об'єктної моделі

Йонесава і Токоро свідчать: "термін" об'єкт " з'явився практично незалежно в різних областях, пов'язаних з комп'ютерами, і майже одночасно на початку 70- х років для позначення того, що може мати різні прояви, залишаючись цілісним. Для того, щоб зменшити складність програмних систем, об'єктами називалися компоненти системи або фрагменти подання знань. На думку Леві, об'єктно - орієнтований підхід був пов'язаний з такими подіями:

* Прогрес в галузі архітектури ЕОМ;
* Розвиток мов програмування, таких як Simula, Smalltalk, CLU, Ada;
*Розвиток методології програмування, включаючи принципи модульності і приховування даних.

До цього ще слід додати три моменти, що вплинули на становлення об'єктного підходу:

* Розвиток теорії баз даних;
* Дослідження в галузі штучного інтелекту;
* Досягнення філософії і теорії пізнання.

Об'єктно-орієнтоване програмування. Що ж таке об'єктно-орієнтоване програмування (object-oriented programming, OOP)? Ми визначаємо його наступним чином:

Об'єктно-орієнтоване програмування - це методологія програмування, заснована на уявленні програми у вигляді сукупності об'єктів, кожний з яких є екземпляром певного класу, а класи утворюють ієрархію успадкування.

У даному визначенні можна виділити три частини: 1) OOP використовує в якості базових елементів об'єкти, 2) кожен об'єкт є екземпляром якогось певного класу, 3) класи організовані ієрархічно. Програма буде об'єктно-орієнтованої тільки при дотриманні всіх трьох зазначених вимог. Зокрема, програмування, не засноване на ієрархічних відносинах, не відноситься до OOP, а називається програмуванням на основі абстрактних типів даних.

Відповідно з цим визначенням не всі мови програмування є об'єктно-орієнтованими. Страуструпп визначив так: "якщо термін об'єктно-орієнтована мова взагалі щось означає, то він повинен означати мову, що має засоби хорошої підтримки об'єктно-орієнтованого стилю програмування... Забезпечення такого стилю в свою чергу означає, що в мові зручно користуватися цим стилем . Якщо написання програм у стилі OOP вимагає спеціальних зусиль або воно неможливе зовсім, то ця мова не відповідає вимогам OOP. Теоретично можлива імітація об'єктно-орієнтованого програмування на звичайних мовах, таких, як Pascal і навіть COBOL або асемблер, але це вкрай важко. Карделл і Вегнер кажуть, що: "мова програмування є об'єктно-орієнтованим тоді і тільки тоді, коли виконуються наступні умови:

* Підтримуються об'єкти, тобто абстракції даних, що мають інтерфейс у вигляді іменованих операцій і власні дані, з обмеженням доступу до них.
* Об'єкти відносяться до відповідних типів (класів).
* Типи (класи) можуть успадковувати атрибути супертипу (суперкласів) ".

Підтримка успадкування в таких мовах означає можливість встановлення відносини "is-a" ("є", "це є", "- це"), наприклад, червона троянда - це квітка, а квітка - це рослина. Мови, що не мають таких механізмів, не можна віднести до об'єктно-орієнтованих. Карделл і Вегнер назвали такі мови об'єктними, але не об'єктно-орієнтованими. Згідно з цим визначенням об'єктно-орієнтованими мовами є Smalltalk, Object Pascal, C і CLOS, а Ada - об'єктна мова. Але, оскільки об'єкти і класи є елементами обох груп мов, бажано використовувати і в тих, і в інших методи об'єктно-орієнтованого проектування.

Об'єктно-орієнтоване проектування. Програмування передусім передбачає правильне й ефективне використання механізмів конкретних мов програмування. Проектування, навпаки, основну увагу приділяє правильному і ефективному структурування складних систем. Ми визначаємо об'єктно-орієнтоване проектування наступним чином:

Об'єктно-орієнтоване проектування - це методологія проектування, що з'єднує в собі процес об'єктної декомпозиції і прийоми представлення логічної і фізичної, а також статичної та динамічної моделей проектованої системи.

У даному визначенні містяться дві важливі частини: 1) об'єктно-орієнтоване проектування ґрунтується на об'єктно-орієнтованій декомпозиції, 2) використовує різноманіття прийомів представлення моделей, що відображають логічну (класи та об'єкти) і фізичну (модулі і процеси) структуру системи, а також її статичні і динамічні аспекти.

Саме об'єктно-орієнтована декомпозиція відрізняє об'єктно-орієнтоване проектування від структурного; в першому випадку логічна структура системи відображається абстракціями у вигляді класів та об'єктів, у другому - алгоритмами.

Об'єктно-орієнтований аналіз. Об'єктно-орієнтований аналіз - це методологія, при якій вимоги до системи сприймаються з точки зору класів та об'єктів, виявлених в предметної області.

Як співвідносяться ООА, ООМ та ООП? На результатах ООА формуються моделі, на яких ґрунтується ООМ; ООМ в свою чергу створює фундамент для остаточної реалізації системи з використанням методології ООП.

Парадигми програмування

Кожен стиль програмування має свою концептуальну базу. Кожен стиль вимагає свого умонастрої і способу сприйняття розв'язуваної задачі. Для об'єктно - орієнтованого стилю концептуальна база - це об'єктна модель. Вона має чотири головні елементи:

* Абстрагування;
* Інкапсуляція;
* Модульність;
* Ієрархія.

Ці елементи є головними в тому сенсі, що без будь-якого з них модель не буде об'єктно - орієнтованої. Крім головних, є ще три додаткові елементи:

* Типізація;
* Паралелізм;
* Збереженість.

Абстракція виділяє істотні характеристики деякого об'єкта, що відрізняють його від всіх інших видів об'єктів і, таким чином, чітко визначає його концептуальні кордони з точки зору спостерігача.

Розглянемо теплицю. Одна з ключових абстракцій в такому завданні - датчик. Відомо кілька різновидів датчиків. Все, що впливає на врожай, повинно бути виміряна, так що ми повинні мати датчики температури води і повітря, вологості, pH, освітлення та концентрації поживних речовин. Із зовнішнього точки зору датчик температури - це об'єкт, який здатний вимірювати температуру там, де він розташований. Що таке температура? Це числовий параметр, що має обмежений діапазон значень і певну точність, що означає число градусів за Фаренгейтом, Цельсієм або Кельвіном. Що таке місце розташування датчика? Це деяке місце в теплиці, температуру в якому нам необхідно знати; таких місць, ймовірно, небагато. Для датчика температури істотно не стільки саме місце розташування, скільки той факт, що даний датчик розташований саме в цьому місці і це відрізняє його від інших датчиків. Тепер можна поставити питання про те, які обов'язки датчика температури? Ми вирішуємо, що датчик повинен знати температуру у своєму місцезнаходженні і повідомляти її за запитом. Які ж дії може виконувати по відношенню до датчика клієнт? Ми приймаємо рішення про те, що клієнт може калібрувати датчик і отримувати від нього значення поточної температури.

```
// Температура по Фаренгейту
typedef float Temperature;

// Число, однозначно определяющее положение датчика
typedef unsigned int Location;

class TemperatureSensor {

public:

            TemperatureSensor (Location);

            ~TemperatureSensor();

            void calibrate(Temperature actualTemperature);

            Temperature currentTemperature() const;
```

```
    private:
...
};
```

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

Модульність - це властивість системи, яка була розкладена на внутрішньо зв'язкові, але слабо пов'язані між собою модулі.

Таким чином, принципи абстрагування, інкапсуляції і модульності є взаємодоповнюючими. Об'єкт логічно визначає межі певної абстракції, а інкапсуляція і модульність роблять їх фізично непорушними.

У процесі поділу системи на модулі можуть бути корисними два правила. По-перше, оскільки модулі служать як елементарних і неподільних блоків програми, які можуть використовуватися в системі повторно, розподіл класів і об'єктів по модулях має враховувати це. По-друге, багато компілятори створюють окремий сегмент коду для кожного модуля. Тому можуть з'явитися обмеження на розмір модуля. Динаміка викликів підпрограм і розташування описів всередині модулів може сильно вплинути на локальність посилань і на управління сторінками віртуальної пам'яті. При поганому розбитті процедур по модулях частішають взаємні виклики між сегментами, що призводить до втрати ефективності кеш -пам'яті і частій зміні сторінок.

Ієрархія - це впорядкування абстракцій, розташування їх по рівнях.

Основними видами ієрархічних структур стосовно до складних систем є структура класів (ієрархія "is-a") і структура об'єктів (ієрархія "part of").

Приклади ієрархії: одиночне успадкування. Важливим елементом об'єктно-орієнтованих систем і основним видом ієрархії "is-a" є згадувана вище концепція наслідування. Спадкування означає таке відношення між класами (відношення батько / нащадок), коли один клас запозичує структурну або функціональну частину одного або декількох інших класів (відповідно, одиночне і множинне успадкування). Іншими словами, успадкування створює таку ієрархію абстракцій, в якій підкласи успадковують будову від одного або декількох суперкласів. Часто підклас добудовує або переписує компоненти вищого класу.

Семантично, успадкування описує ставлення типу "is-a". Наприклад, ведмідь є ссавець, будинок є нерухомість і "швидке сортування" є сортують алгоритм. Таким чином, спадкування породжує ієрархію "узагальнення-спеціалізація", в якій підклас являє собою спеціалізований приватний випадок свого суперкласу. "Лакмусовий папірець" наслідування - зворотна перевірка; так, якщо B не є A, то B не варто проводити від A.

Лекція 2. Класи в UML.

Моделювання системи передбачає ідентифікацію сутностей, важливих з тієї чи іншої точки зору. Ці сутності становлять словник модельованої системи. Наприклад, якщо ви будуєте будинок, то для вас як домовласника будуть мати значення стіни, двері, вікна, вбудовані шафи та освітлення. Кожна з названих сутностей відрізняється від інших і характеризується власним набором властивостей. Для стін є висота і ширина, вони тверді і суцільні. Для дверей також є висота і ширина, вони теж суцільні, але, крім того, забезпечені особливим механізмом, що дозволяє їм відкриватися в один бік. Вікна схожі на двері, оскільки являють собою отвори в стінах, але в іншому властивості зазначених сутностей розрізняються. Вікна зазвичай (хоча й не завжди) проектують так, щоб через них можна було дивитися, але не проходити.

Стіни, двері та вікна рідко існують самі по собі, тому необхідно вирішити, як вони будуть стикуватися один з одним. Які суті ви виберете і які відносини між ними вирішите встановити, очевидно, визначається в залежності від того, як ви збираєтеся

використовувати кімнати в будинку, як будете переміщатися між ними, а також від загального стилю і обстановки, які входять у ваш задум.

Будівельників, обслуговуючий персонал та мешканців цікавлять різні речі. Водопровідники звернуть увагу на труби, крани і вентиляційні отвори. Вас як домовласника це особливо не обходить, якщо не вважати випадків, коли зазначені елементи перетинаються з тими, які потрапляють у ваше поле зору, - вас хвилює, наприклад, де труба вмонтована в підлогу і в якому місці даху відкривається вентиляція.

В UML всі сутності подібного роду моделюються як класи. Клас - це абстракція сутностей, що є частиною вашого словника. Клас представляє не індивідуальний об'єкт, а цілу їх сукупність. Так, умоглядно ви можете вважати, що "стіна" - це клас об'єктів з деякими загальними властивостями, такими як висота, довжина, товщина, що несе це стіна чи ні, і т.д. При цьому конкретні стіни будуть розглядатися як окремі екземпляри класу "стіна", одним з яких є, наприклад, "стіна в південно-західній частині мого кабінету".

Багато мов програмування безпосередньо підтримують концепцію класів. І це чудово, оскільки в такому випадку створювані вами абстракції можуть бути безпосередньо відображені в конструкціях мови програмування, навіть якщо мова йде про абстракції не програмних сутностей типу "покупець", "торгівля" або "розмова".

Терміни і поняття

Класом (Class) називається опис сукупності об'єктів із загальними атрибутами, операціями, відносинами і семантикою. Графічно клас зображується у вигляді прямокутника.

Імена

У кожного класу має бути ім'я, що відрізняє його від інших класів. Ім'я класу - це текстовий рядок. Взяте саме по собі, воно називається простим ім'ям; до складеного імені спереду додано ім'я пакета, куди входить клас. Ім'я класу в одному пакеті повинно бути унікальним. При графічному зображенні класу показується тільки його ім'я:

Атрибути

Атрибут - це іменована властивість класу, що включає опис безлічі значень, які можуть приймати екземпляри цієї властивості. Клас може мати будь-яке число атрибутів або не мати їх зовсім. Атрибут представляє деяку властивість сутності, спільну для всіх об'єктів даного класу. Наприклад, у будь-якої стіни є висота, ширина і товщина; при моделюванні клієнтів можна задавати прізвище, адресу, номер телефону та дату народження. Таким чином, атрибут є абстракцією даних об'єкта або його стану. У кожний момент часу будь-який атрибут об'єкта, що належить даному класу, має цілком певне значення. Атрибути представлені в розділі, який розташований під ім'ям класу, при цьому вказуються тільки їх імена:

Операції

Операцією називається реалізація послуги, яку можна запросити у будь-якого об'єкта класу для впливу на поведінку. Іншими словами, операція - це абстракція того, що дозволено робити з об'єктом. У всіх об'єктів класу є загальний набір операцій. Клас може містити будь-яке число операцій або не містити їх зовсім. Наприклад, для всіх об'єктів класу Rectangle (Прямокутник) з бібліотеки для роботи з вікнами, що міститься в пакеті мови Java, визначені операції переміщення, зміни розміру та опитування значень властивостей. Часто (хоча не завжди) звернення до операції об'єкта змінює його стан або його дані. Операції класу зображуються в розділі, розташованому нижче розділу з атрибутами. При цьому можна обмежитися лише іменами. Більш детальна специфікація виконання операції здійснюється за допомогою приміток і діаграм діяльності.

Організація атрибутів та операцій

При зображенні класу необов'язково відразу показувати всі його атрибути і операції. Як правило, це просто неможливо - їх занадто багато для одного малюнка, - та й не потрібно (оскільки для цього подання системи лише невелика підмножина атрибутів і

операцій має значення). З цих причин клас зазвичай згортають, тобто зображують лише деякі з наявних атрибутів і операцій, а то й зовсім опускають їх. Таким чином, порожній розділ у відповідному місці прямокутника може означати не відсутність атрибутів або операцій, а тільки те, що їх не вважали за потрібне зобразити. Явним чином наявність додаткових атрибутів або операцій можна позначити, поставивши в кінці списку три крапки.

Обов'язки

Обов'язки (Responsibilities) класу - це свого роду контракт, якому він повинен підкорятися. Визначаючи клас, ви заявляєте, що всі його об'єкти мають однотипний стан і ведуть себе однаково. Висловлюючись абстрактно, відповідні атрибути та операції якраз і є тими властивостями, за допомогою яких виконуються обов'язки класу. Наприклад, клас Wall (Стіна) відповідає за інформацію про висоту, ширину і товщину. Клас FraudAgent (Агент), який зустрічається в додатках з обробки кредитних карток, відповідає за оцінку платіжних вимог - законні вони, підозрілі або підроблені. Клас TemperatureSensor (Датчик Температури) відповідає за вимір температури і подачу сигналу тривоги у випадку перевищення заданого рівня.

Моделювання класів краще всього починати з визначення обов'язків сутностей, які входять до словника системи. На цьому етапі особливо корисні будуть такі методики, як застосування CRC-карток та аналіз прецедентів. В принципі число обов'язків класу може бути довільним, але на практиці добре структурований клас має щонайменше один обов'язок, з іншого боку, їх не повинно бути і дуже багато. При уточненні моделі обов'язки класу перетворюються в сукупність атрибутів і операцій, які повинні найкраще забезпечити їх виконання.

Графічно обов'язки зображують в особливому розділі в нижній частині піктограми класу. Їх можна вказати також у примітці.

Інші властивості

При створенні абстракцій вам найчастіше доводиться мати справу з атрибутами, операціями та обов'язками. Для більшості моделей цього цілком достатньо, щоб описати найважливіші риси семантики класів. Але іноді доводиться візуалізувати чи специфікувати інші особливості, як то: видимість окремих атрибутів і операцій, специфічні для конкретної мови програмування властивості операції (наприклад, чи є вона поліморфної або константної) або навіть винятки, які об'єкти класу можуть порушувати чи обробляти. Ці та багато інших особливостей теж можна виразити в UML, але в такому випадку потрібно звернення до більш розвиненим можливостям мови.

Нарешті, класи рідко існують самі по собі. При побудові моделей слід звертати увагу на групи взаємодіючих між собою класів. В UML такі спільноти прийнято називати кооперації і зображати на діаграмах класів.

Типові прийоми моделювання

Словник системи

За допомогою класів зазвичай моделюють абстракції, витягнуті з розв'язуваної задачі або технології, що застосовується для її вирішення. Такі абстракції є складовою частиною словника вашої системи, тобто представляють суті, важливі для користувачів і розробників.

Для користувачів не складає труднощів ідентифікувати велику частину абстракцій, оскільки вони створені на основі тих сутностей, які вже використовувалися для опису системи. Щоб допомогти користувачам виявити ці абстракції, найкраще вдатися до таких методів, як використання CRC-карток та аналіз прецедентів. Для розробників ж абстракції є зазвичай просто елементами технології, яка була задіяна при виконанні завдання.

Моделювання словника системи включає в себе наступні етапи:

1. Визначення, які елементи користувачі та розробники застосовують для опису задачі або її рішення. Для відшукання правильних абстракцій використовуються CRC-картки та аналіз прецедентів.

2. Виявлення для кожної абстракції відповідної їй множини обов'язків. Важливо, щоб кожен клас був чітко визначений, а розподіл обов'язків між ними добре збалансовано.

3. Розробка атрибутів та операцій, необхідних для виконання класу своїх обов'язків.

У міру того як будуються все більш складні моделі, виявиться, що багато класів природним чином поєднуються в концептуально і семантично споріднені групи. В UML для моделювання таких груп класів використовуються пакети.

Розподіл обов'язків у системі

Якщо у проект входить щось більше, ніж пара нескладних класів, належить подбати про збалансований розподіл обов'язків. Це означає, що треба уникати занадто великих або, навпаки, занадто маленьких класів. Кожен клас повинен добре робити щось одне. Якщо абстрактні класи занадто великі, то модель буде важко модифікувати та повторно використовувати. Якщо ж вони занадто малі, то доведеться мати справу з такою великою кількістю абстракцій, що ні зрозуміти, ні керувати ними буде неможливо. Мова UML здатна допомогти у візуалізації та специфікації балансу обов'язків.

Моделювання розподілу обов'язків у системі включає в себе такі етапи:

1. Ідентифікація сукупності класів, які спільно відповідають за деяку поведінку.

2. Визначення обов'язків кожного класу.

3. Аналіз отриманих класів як єдиного цілого і розбиття тих з них, у яких занадто багато обов'язків, на менші - і навпаки, крихітні класи з елементарними обов'язками об'єднання в більші.

4. Перерозподіл обов'язків так, щоб кожна абстракція стала з розумною мірою автономною.

5. Аналіз, як класи кооперуються один з одним, і перерозподіл обов'язків з таким розрахунком, щоб жоден клас в рамках кооперації не робив дуже багато або дуже мало.

Не програмні сутності

Модельовані сутності можуть не мати аналогів в програмному забезпеченні. Наприклад, частиною робочого процесу в моделі підприємства роздрібної торгівлі можуть бути люди, що відправляють накладні, та роботи, які автоматично упаковують замовлені товари для доставки зі складу за місцем призначення. У додатку зовсім не обов'язково виявляться компоненти для подання цих сутностей (на відміну від сутності "клієнт ", для зберігання інформації про яку, власне, і створюється система).

Моделювання не програмних сутностей проводиться таким чином:

1. Моделюються сутності у вигляді класів.

2. Для того, щоб відрізнити ці сутності від зумовлених будівельних блоків UML, необхідно створити за допомогою стереотипів новий будівельний блок, описати його семантику і зіставити з ним ясний візуальний образ.

3. Якщо модельований елемент є апаратним засобом з власним програмним забезпеченням, можна змоделювати його у вигляді вузла, який дозволив би надалі розширити його структуру.

Примітивні типи

Іншою крайністю є сутності, взяті безпосередньо з мови програмування, що використовується при виконанні завдання. Зазвичай до таких абстракцій відносять примітивні типи, наприклад цілі, символи, рядки та типи, які програміст створює сам.

Моделювання примітивних типів проводиться таким чином:

1. Моделюється сутність, яка зображується за допомогою нотації класу з відповідним стереотипом.

2. Якщо потрібно задати пов'язаний з типом діапазон значень, можна скористатися обмеженнями.

Рис. показує, що такі сутності можна моделювати в UML як типи або перерахування, які зображуються точно так само, як класи, але з явно вказаними стереотипом. Сутності, подібні цілим числам (представлені класом Int), моделюються як типи, а з допомогою обмежень можна явно вказати діапазон прийнятих значень.

Поради

При моделюванні класів в UML завжди пам'ятайте, що кожному класу повинна відповідати деяка реальна сутність або концептуальна абстракція з області, з якою має справу користувач або розробник. Добре структурований клас має такі властивості:

є чітко окресленої абстракцією деякого поняття зі словника проблемної області або області рішення;

містить невеликий, точно певний набір обов'язків і виконує кожну з них;

підтримує чіткий поділ специфікацій абстракції та її реалізації;

зрозумілий і простий, але в той же час допускає розширення та адаптацію до нових завдань.

Зображуючи клас в UML, дотримуйтеся наступних правил:

показуйте тільки ті його властивості, які важливі для розуміння абстракції в даному контексті;

розділяйте довгі списки атрибутів і операцій на групи відповідно з їх категоріями;

показуйте взаємопов'язані класи на одній і тій же діаграмі.

Лекція 3. Відносини між об'єктами

Типи відносин

Самі по собі об'єкти не представляють ніякого інтересу: тільки в процесі взаємодії об'єктів реалізується система. За висловом Інгалса: "Замість процесора, який безсоромно перемелює структури даних, ми отримуємо співтовариство добре вихованих об'єктів, які чемно просять один одного про послуги". Літак, за визначенням, "сукупність елементів, кожен з яких за своєю природою прагне впасти на землю, але за рахунок спільних безперервних зусиль долають цю тенденцію". Він летить тільки завдяки узгодженим зусиллям своїх компонентів.

Відносини двох будь-яких об'єктів грунтуються на припущеннях, якими один володіє відносно іншого: про операції, які можна виконувати, і про очікувану поведінку. Особливий інтерес для об'єктно - орієнтованого аналізу і проектування становлять три типи відносин між класами:

* залежності, які описують існуючі між класами відносини використання (включаючи відносини уточнення, трасування і зв'язування);
* узагальнення, що зв'язують узагальнені класи зі спеціалізованими;
* асоціації, що представляють структурні відносини між об'єктами.

Кожен з цих типів дозволяє по-різному комбінувати абстракції.

Побудова мережі відносин схожа на створення збалансованого розподілу обов'язків між класами. Варто злегка перестаратися - і ви отримаєте заплутаний клубок відносин; в результаті модель виявиться абсолютно незрозумілою. З іншого боку, надмірне спрощення призведе до того, що залишиться нерозкритим багатство зв'язків між різними елементами системи.

Для кожного з названих типів відносин мова UML надає графічне зображення, як показано на рис.1. Ця нотація дозволяє візуалізувати модельовані відносини незалежно від застосовуваної мови програмування, так щоб підкреслити їх найбільш важливі складові: ім'я, сутності, що зв'язуються, та властивості.

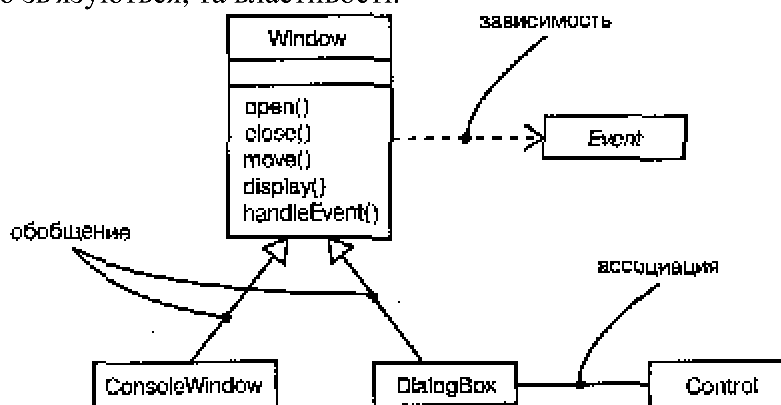


Рис.1. Приклад відносин між класами

Залежністю (Dependency) називають відносини використання, згідно з яким зміна в специфікації одного елемента (наприклад, класу Event) може вплинути на інший елемент, який його використовує (в даному випадку - клас Window), причому зворотне не обов'язково. Графічно залежність зображується пунктирною лінією зі стрілкою, спрямованої від даного елемента на той, від якого він залежить. Використовуйте залежності, коли хочете показати, що один елемент використовує інший.

Найчастіше залежності застосовуються при роботі з класами, щоб відобразити в сигнатурі операції той факт, що один клас використовує інший в якості аргументу. В UML дозволяється визначати залежності і між іншими елементами, наприклад примітками або пакетами.

Узагальнення (Generalization) – це відносини між загальною сутністю (суперкласом, або батьком) і її конкретним втіленням (субклассом або нащадком). Узагальнення іноді називають відносинами типу "це є ", маючи на увазі, що одна сутність (наприклад, клас BayWindow) є приватним вираженням іншого, більш загального (скажімо, класу Window). Узагальнення означає, що об'єкти класу-нащадка можуть використовуватися скрізь, де зустрічаються об'єкти класу-батька, але не навпаки. Іншими словами, нащадок може бути підставлений замість батька. При цьому він успадковує властивості батьків, зокрема його атрибути і операції. Часто, хоча і не завжди, у нащадків є і свої власні атрибути і операції, крім тих, що існують у батька. Операція нащадка з тією ж сигнатурою, що і в батька, заміщає операцію батька; цю властивість називають поліморфізмом (Polymorphism). Графічно відносини узагальнення зображується у вигляді лінії з великою не зафарбованою стрілкою, спрямованої на батька, як показано на рис. 2. Застосовуйте узагальнення, коли хочете показати відносини типу "батько / нащадок ".

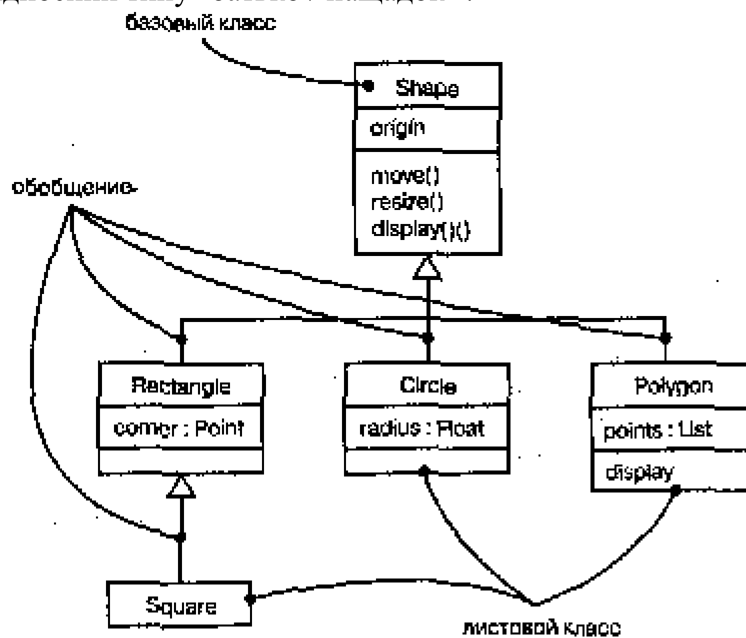


Рис.2. Приклад відносин узагальнення

Клас може мати одного або декількох батьків або не мати їх зовсім. Клас, у якого немає батьків, але є нащадки, називається базовим (base) або кореневим (root), а той, у якого немає нащадків, - листовим (leaf). Про клас, у якого є тільки один батько, кажуть, що він використовує одиночне спадкування (Single inheritance); якщо батьків кілька, мова йде про множинне спадкування (Multiple inheritance).

Узагальнення найчастіше використовують між класами і інтерфейсами, щоб показати відносини спадкування. В UML можна створювати відносини узагальнення і між іншими елементами, зокрема пакетами.

Асоціацією (Association) називається структурне відношення, яке показує, що об'єкти одного типу якимось чином пов'язані з об'єктами іншого типу. Якщо між двома класами визначена асоціація, то можна переміщатися від об'єктів одного класу до об'єктів іншого. Серед вказаних типів відносин асоціація найбільш абстрактне і найбільш слабке відношення. Ідентифікація асоціацій між класами часто проводиться на етапі аналізу і ранішніх стадіях проектування, коли необхідно визначити загальні залежності між абстракціями.

Цілком припустимі випадки, коли обидва кінці асоціації відносяться до одного й того ж класу. Це означає, що з об'єктом деякого класу дозволено зв'язати інші об'єкти з того ж класу. Асоціація, що зв'язує два класи, називається бінарною. Можна, хоча це рідко буває необхідним, створювати асоціації, що зв'язують відразу декілька класів; вони називаються n-арні. Графічно асоціація зображується у вигляді лінії, що з'єднує клас сам із собою або з іншими класами. Використовуйте асоціації, коли хочете показати структурні відносини. (Асоціації і залежності, на відміну від відносин узагальнення, можуть бути рефлексивними.) Крім описаної базової форми існує чотири додатки, які можна застосувати до асоціацій.

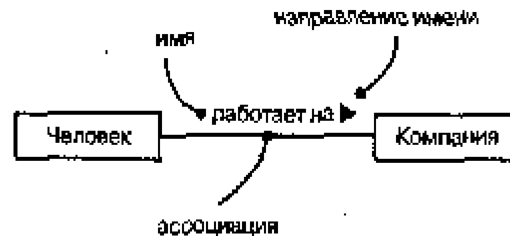


Рис.3. Приклад асоціації

Ім'я. Асоціації може бути присвоєно ім'я, яке описує природу відносини. Щоб уникнути можливих двозначностей в розумінні імені, достатньо за допомогою чорного трикутника вказати напрям, в якому воно повинно читатися, як показано на рис. 3. Зазвичай ім'я асоціації не вказується, якщо тільки ви не хочете явно задати для неї рольові імена чи у вашій моделі настільки багато асоціацій, що виникає необхідність посилатися на них і відрізняти один від одного. Ім'я буде особливо корисним, якщо між одними і тими ж класами існує кілька різних асоціацій.

Роль. Клас, що бере участь в асоціації, грає в ній деяку роль. По суті, це "обличчя", яким клас, що знаходиться на одній стороні асоціації, звернений до класу з іншого її боку. Ви можете явно позначити роль, яку відіграє клас в асоціації. На рис. 4 показано, що клас Людина, що грає роль працівника, асоційований з класом Компанія, що грає роль роботодавця. Ролі тісно пов'язані з семантикою інтерфейсів.

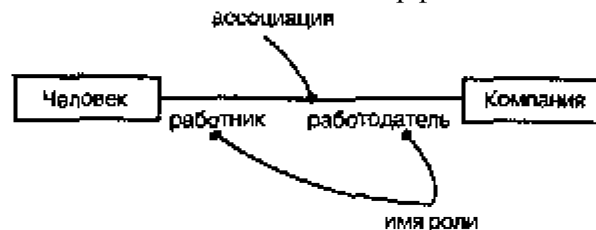


Рис.4. Ролі.

Кратність. Асоціації відображають структурні відносини між об'єктами. Часто при моделюванні буває важливо вказати, скільки об'єктів може бути пов'язано допомогою одного примірника асоціації. Це число називається кратністю (Multiplicity) ролі асоціації та записується або як вираз, значенням якого є діапазон значень, або в явному вигляді, як показано на рис. 5. Вказуючи кратність на одному кінці асоціації, ви тим самим говорите, що на цьому кінці саме стільки об'єктів повинно відповідати кожному об'єкту на протилежному кінці. Кратність можна задати «дорівнює одиниці» (1), можна вказати діапазон: «нуль або одиниця» (0.. 1), «багато» (0..*), «одиниця або більше» (1..*). Дозволяється також вказувати певне число (наприклад, 3).

Така властивість асоціації має ще назву «множинність». Відповідно усьому викладеному, існує три різновиди множинної асоціації:

1) взаємно-однозначна залежність (один до одного);

2) залежність «один-множина» (один до багатьох);

3) залежність «множина-множина» (багато до багатьох).

Взаємно-однозначне відношення означає дуже вузьку асоціацію. Наприклад, між класом Sale і класом CreditCardTransaction: кожному продажу відповідає тільки одна

транзакція з цією кредитною карткою, а кожна транзакція відповідає тільки одному продажу. Приклад другого типу асоціації: кожний екземпляр класу Button (Кнопка) зв’язаний тільки з одним екземпляром класу Window (Вікно), але, у свою чергу, кожний екземпляр класу Window може мати декілька об’єктів класу Button. І, нарешті, третій тип асоціації (багато до багатьох) ілюструє такий приклад. Кожний покупець, тобто екземпляр класу Customer, може ініціювати транзакцію по продажу товару з декількома продавцями (екземпляри класу SalePerson), та навпаки. На рис. 5 наведений приклад асоціації з визначенням кратності відносин.

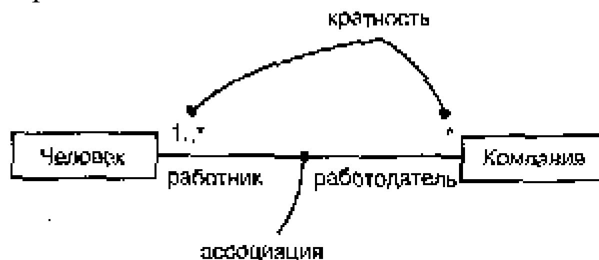


Рис. 5. Приклад нанесення на діаграму кратності відносин.

Агрегування. Проста асоціація між двома класами відображає структурне відношення між рівноправними сутностями, коли обидва класи знаходяться на одному концептуальному рівні і ні один не є більш важливим, ніж інший. Але іноді доводиться моделювати відношення типу "частина / ціле ", в якому один з класів має більш високий ранг (ціле) і складається з кількох менших за рангом (частин). Ставлення такого типу називають агрегування; воно зараховане до відносин типу « має» (з урахуванням того, що об'єкт-ціле має кілька об'єктів-частин). Агрегування є окремим випадком асоціації і зображується у вигляді простої асоціації з не зафарбованим ромбом з боку «цілого».

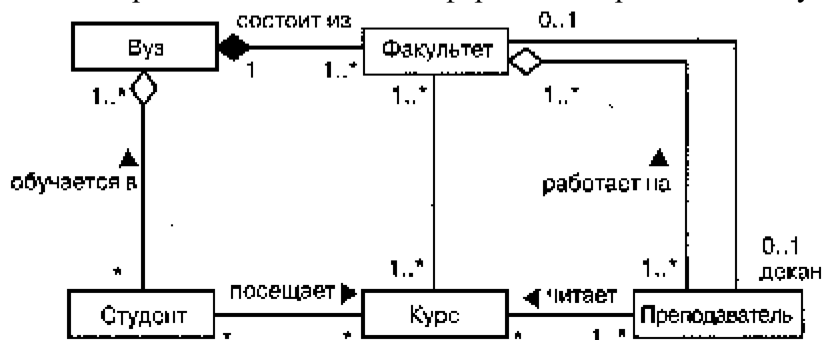


Рис.6. Приклад різних типів асоціацій.

Поради

При моделюванні відносин в UML дотримуйтесь наступних правил:

* Використовуйте залежність, тільки якщо відносини, що моделюються, не є структурними;
* Використовуйте узагальнення, тільки якщо має місце відношення типу "є";
* Множинне успадкування часто можна замінити агрегуванням;
* Остерігайтеся циклічних відносин узагальнення;
*Підтримуйте баланс у відносинах узагальнення: ієрархія наслідування не повинна бути ні занадто глибокої (бажано не більше п'яти рівнів), ні занадто широкою (краще вдатися до проміжних абстрактним класів);
* Застосовуйте асоціації передусім там, де між об'єктами існують структурні відносини.

При зображенні відносин в UML керуйтеся нижченаведеними рекомендаціями:

* Вибравши один із стилів оформлення ліній (прямі або похилі), надалі намагайтеся його дотримуватися. Прямі лінії підкреслюють, що сполуки йдуть від родинних сутностей до одного спільного батька. Похилі лінії дозволяють істотно заощадити простір у складних діаграмах. Якщо ви хочете привернути увагу до різних груп відносин, застосовуйте одночасно обидва типи ліній;

* Уникайте перетину ліній;

* Показуйте тільки такі відносини, які необхідні для розуміння особливостей групування елементів моделі; приховуйте несуттєві (особливо надлишкові) асоціації.

Лекція 4. Діаграми класів

Діаграми класів при моделюванні об'єктно - орієнтованих систем зустрічаються частіше інших. На таких діаграмах показується безліч класів, інтерфейсів, кооперацій і відносин між ними.

Діаграми класів використовуються для моделювання статичного вигляду системи з точки зору проектування. Сюди здебільшого відноситься моделювання словника системи, кооперацій та схем. Крім того, діаграми класів складають основу ще двох діаграм - компонентів і розгортання.

Діаграми класів важливі не тільки для візуалізації, специфікації та документування структурних моделей, але також для прямого і зворотного проектування виконуваних систем.

Діаграмою класів (Class diagram) називають діаграму, на якій показано безліч класів, інтерфейсів, кооперацій і відносин між ними. Її зображують у вигляді безлічі вершин і дуг.

Діаграми класів зазвичай містять такі сутності:

* Класи;
* Інтерфейси ;
* Кооперації ;
* Відносини залежності, узагальнення та асоціацій .

Подібно до всіх інших діаграм, вони можуть включати в себе примітки та обмеження.

Також в діаграмах класів можуть бути присутніми пакети або підсистеми, які застосовуються для групування елементів моделі в більші блоки. Іноді в ці діаграми поміщають екземпляри, особливо якщо потрібно візуалізувати їх тип (можливо, динамічний).

Типові приклади застосування

Діаграми класів застосовують для моделювання статичного вигляду системи з точки зору проектування. У цьому поданні найзручніше описувати функціональні вимоги до системи - послуги, які вона надає кінцевому користувачеві.

Зазвичай діаграми класів використовуються в таких цілях:

для моделювання словника системи; моделювання словника системи передбачає прийняття рішення про те, які абстракції є частиною системи, а які – ні; за допомогою діаграм класів ви можете визначити ці абстракції та їх обов'язки;

для моделювання простих кооперацій; кооперація - це спільнота класів, інтерфейсів та інших елементів, що працюють спільно для забезпечення деякого кооперативного поведінки, більш значущого, ніж сума складових його елементів; наприклад, моделюючи семантику транзакцій в розподіленій системі, ви не зможете зрозуміти процеси, що відбуваються, дивлячись на один-єдиний клас, оскільки відповідна семантика забезпечується декількома спільно працюючими класами; за допомогою діаграм класів вдається візуалізувати і специфікувати ці класи і відносини між ними;

для моделювання логічної схеми бази даних; логічну схему можна уявляти собі як креслення концептуального проекту бази даних; у багатьох сферах діяльності потрібно зберігати стійку (persistent) інформацію в реляційної або об'єктно - орієнтованої базі даних; моделювати схеми також можна за допомогою діаграм класів.

Прості кооперації

Класи не існують самі по собі. Будь-який клас функціонує спільно з іншими, реалізуючи семантику, що виходить за межі кожного окремого елемента. Таким чином, окрім визначення словника системи, потрібно приділити увагу візуалізації, специфікації, конструюванню та документуванню різних способів спільної роботи, яка увійшла до словника сутностей. Для моделювання такого "співробітництва" застосовуються діаграми класів.

Створюючи діаграму класів, ви просто моделюєте частину сутностей і відносин, що описуються в поданні системи з точки зору проектування. Бажано, щоб кожна діаграма описувала тільки одну кооперацію.

Моделювання кооперації здійснюється наступним чином:

1. Ідентифікуйте механізми, які ви збираєтеся моделювати. Механізм - це деяка функція або поведінка частини модельованої системи, що з'являється в результаті взаємодії спільноти класів, інтерфейсів та інших сутностей. (Механізми, як правило, тісно пов'язані з прецедентами.)

2. Для кожного механізму ідентифікуйте класи, інтерфейси та інші кооперації, які беруть участь в даній кооперації. Ідентифікуйте також відносини між цими сутностями.

3. Перевірте працездатність системи за допомогою прецедентів. По ходу справи ви, можливо, виявите, що деякі її частини виявилися пропущені, а деякі - семантично неправильні.

4. Заповніть ідентифіковані елементи змістом. Що стосується класів, почніть з правильного розподілу обов'язків; пізніше можна буде перетворити їх в конкретні атрибути та операції.

Як приклад на рисунку показані класи, взяті з реалізації автономного робота. Основна увага тут приділена тим класам, які беруть участь у механізмі руху робота по заданій траєкторії. Як видно з малюнка, існує один абстрактний клас Мотор з двома конкретними нащадками, МоторПоворотногоМеханизма та ГлавныйМотор, які успадковують п'ять операцій їх батьків. Ці два класи, в свою чергу, є частиною класу Привод. Клас АгентТраектории зв'язаний ставленням асоціації "один до одного" з класом Привод, відношенням "один до багатьох" - з класом ДатчикСтолкновений. Для класу АгентТраектории не показано ні атрибутів, ні операцій, хоча наведені обов'язки.

Логічна схема бази даних

Багато модельованих систем містять стійкі об'єкти, тобто такі, які можна зберігати в базі даних і згодом отримувати при необхідності. Для цього найчастіше використовують реляційні, об'єктно-орієнтовані або гібридні об'єктно-реляційні бази даних. UML дозволяє моделювати логічні схеми баз даних, так само як і самі фізичні бази.

Діаграми класів UML включають в себе, як окремий випадок, діаграми "сутність-зв'язок" (ER діаграми), які часто використовуються для логічного проектування баз даних. Але якщо в класичних ER діаграмах увага зосереджена тільки на даних, діаграми класів - це крок вперед: вони дозволяють моделювати також і поведінку. У реальній базі даних подібні логічні операції зазвичай трансформуються в тригери або збережені процедури.

Пряме і зворотне проектування

Моделювання, звичайно, важлива річ, але слід пам'ятати, що основним результатом діяльності групи розробників є не діаграми, а програмне забезпечення. Всі зусилля щодо створення моделей спрямовані тільки на те, щоб в обумовлені терміни написати програму, яка відповідає потребам користувачів і бізнесу. Тому дуже важливо, щоб ваші моделі і засновані на них реалізації відповідали один одному, причому з мінімальними витратами з підтримання синхронізації між ними.

Іноді UML застосовується так, що створювана модель згодом не відображається ні в якій програмний код. Якщо ви, наприклад, моделюєте за допомогою діаграм дій бізнес-процесу, то в багатьох діях братимуть участь люди, а не комп'ютери. В інших випадках необхідно моделювати системи, складовими частинами яких на даному рівні абстракції є тільки апаратні засоби (хоча на іншому рівні абстракції цілком може виявитися, що вони містять вбудовані процесори з відповідним програмним забезпеченням).

Але найчастіше моделі все-таки перетворюються в код. Хоча UML не визначає конкретного способу відображення на який-небудь об'єктно-орієнтована мова, він проектувався з урахуванням цієї вимоги. Найбільшою мірою це стосується до діаграм

класів, зміст яких легко відображається на такі відомі об'єктно - орієнтовані мови програмування, як Java, C, Smalltalk, Eiffel, Ada, ObjectPascal і Forte. Крім того, в проект UML була закладена можливість відображення на комерційні об'єктні мови, наприклад Visual Basic.

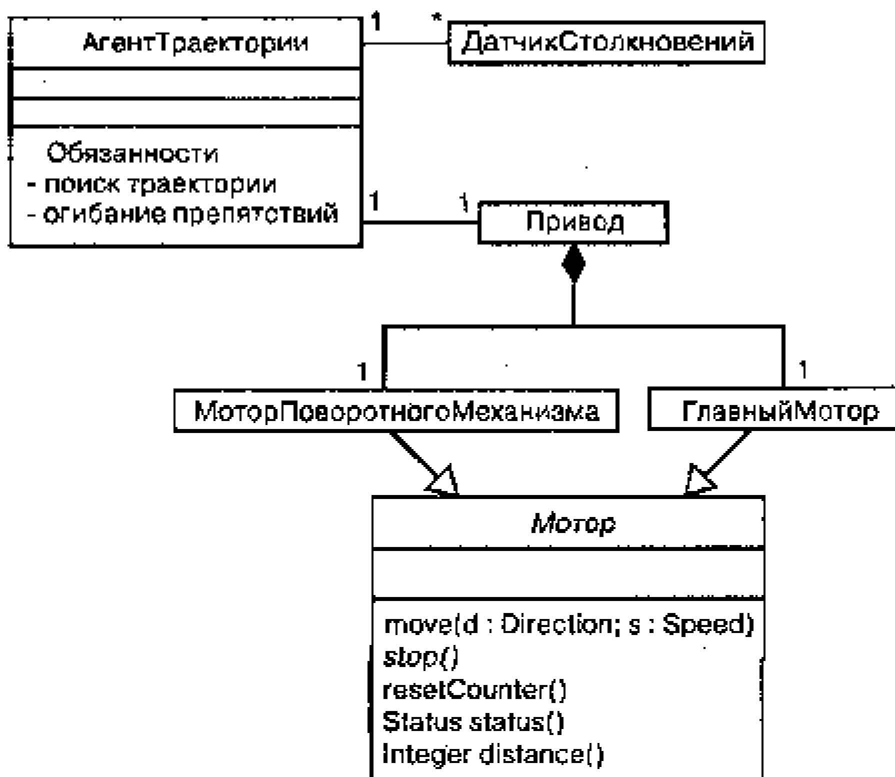


Рис. Прості кооперації

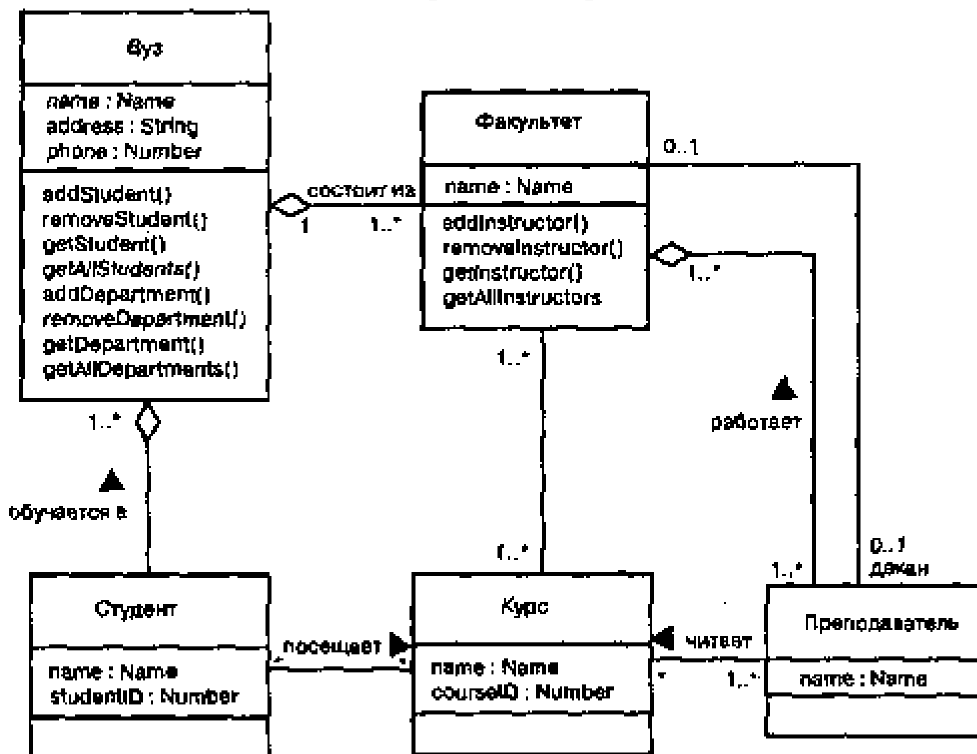
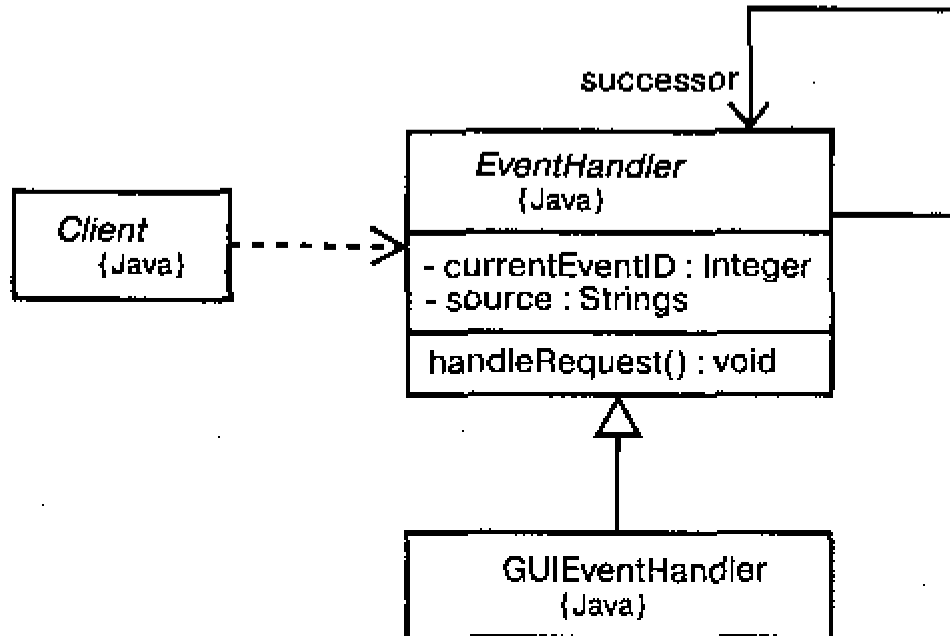


Рис. Логічна схема БД

Прямим проектуванням (Forward engineering) називається процес перетворення моделі в код шляхом відображення на деякий мову реалізації. Процес прямого проектування призводить до втрати інформації, оскільки написані на мові UML моделі семантично багатшими будь-якого з існуючих об'єктно - орієнтованих мов. Фактично саме ця різниця і є основною причиною, по якій ми, крім коду, потребуємо і в моделях. Деякі структурні властивості системи, такі як кооперації, або її поведінкові особливості, наприклад взаємодії, можуть бути легко візуалізовані в UML, але в чистому коді наочність втрачається.



Зворотним проектуванням (Reverse engineering) називається процес перетворення в модель коду, записаного на якій-небудь мові програмування. В результаті цього процесу ви отримуєте величезний обсяг інформації, частина якої знаходиться на нижчому рівні деталізації, ніж необхідно для побудови корисних моделей. У той же час зворотне проектування ніколи не буває повним. Як уже згадувалося, пряме проектування веде до втрати інформації, так що повністю відновити модель на основі коду не вдасться, якщо тільки інструментальні засоби не включали в коментарях до вихідного тексту інформацію, що виходить за межі семантики мови реалізації.

Поради

Створюючи діаграму класів на мові UML, пам'ятайте, що це всього лише графічне зображення статичного вигляду системи з точки зору проектування. Узята у відриві від інших, жодна діаграма класів не може і не повинна охоплювати цей вид цілком. Діаграми класів описують його вичерпно, але кожна окремо - лише один з його аспектів.

Добре структурована діаграма класів має такі властивості:

• загострює увагу тільки на одному аспекті статичного вигляду системи з точки зору проектування;

• містить лише елементи, істотні для розуміння даного аспекту;

• показує деталі, відповідні необхідному рівню абстракції, опускаючи ті, без яких можна обійтися;

• не настільки лаконічна, щоб ввести читача в оману щодо важливих аспектів семантики.

При зображенні діаграми класів керуйтеся наступними правилами:

• дайте діаграмі ім'я, пов'язане з її призначенням;

• розташовуйте елементи так, щоб звести до мінімуму число пересічних ліній;

• просторово організуйте елементи так, щоб семантично близькі сутності розташовувалися поряд;

• щоб привернути увагу до важливих особливостей діаграми, використовуйте примітки та колір;

• намагайтеся не показувати занадто багато різних видів відносин; як правило, у кожній діаграмі класів повинні домінувати відносини якого-небудь одного виду.

Лекція №5. Поглиблене вивчення класів

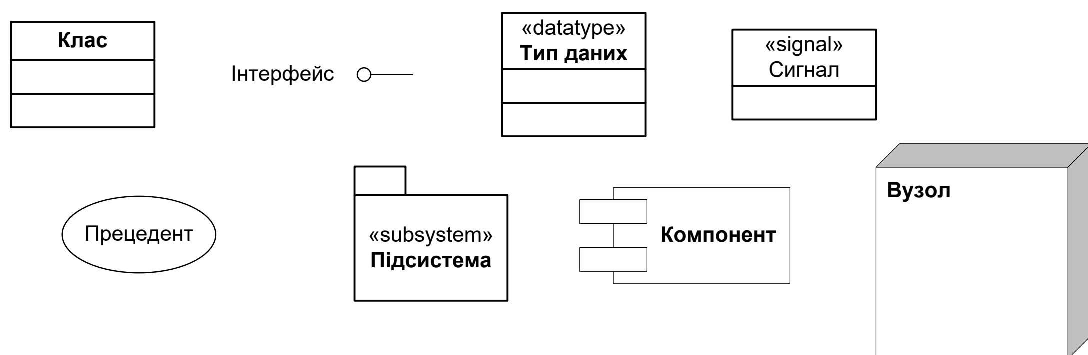
Класи - це найважливіші будівельні блоки об'єктно - орієнтованих систем. Проте класи є всього лише одним з різновидів більш загальних будівельних блоків UML - класифікаторів. Класифікатор - це механізм, що описує структурні і поведінкові властивості. До числа класифікаторів відносяться класи, інтерфейси, типи даних, сигнали, компоненти, вузли, прецеденти (варіанти використання) і підсистеми.

Класифікатори

У процесі моделювання розробник стикаєтеся з абстракціями сутностей, що належать реального світу, і сутностей, введених ним для вирішення задачі. Наприклад, при створенні системи замовлень, що працює через мережу Internet, швидше за все, доведеться включити в проект клас Клієнт (описує замовника) і клас Транзакція (артефакт реалізації, відповідний атомарній дії). У деталізованій системі, ймовірно, буде ще компонент Ціни, примірники якого розміщуються на всіх клієнтських вузлах. Кожна з перелічених абстракцій фігурує в моделі разом зі своїми конкретними екземплярами, тому поділ логіки та примірників сутностей є важливою частиною моделювання.

Деякі сутності в UML не мають екземплярів. До їх числа відносяться, наприклад, пакети і відношення узагальнення. У загальному значенні ті елементи моделювання, які можуть мати екземпляри, називаються класифікаторами. Ще важливіше те, що класифікатори характеризуються структурними (у формі атрибутів) та поведінковими (у формі операцій) властивостями. Усі примірники одного класифікатора мають загальні властивості.

Найважливішим видом класифікатора в UML є клас. Класом називається опис сукупності об'єктів із загальними атрибутами, операціями, відносинами і семантикою. Однак крім класів існують і інші класифікатори.



Інтерфейс - набір операцій, які використовуються для того, щоб специфікувати послуги, що надаються класом або компонентом;

Тип даних - тип, значення якого не індивідуалізовані; сюди входять примітивні вбудовані типи, такі як числа і рядки, а також перераховані типи, наприклад Boolean;

Сигнал - специфікація асинхронного стимулу, використовуваного для зв'язку між екземплярами;

Компонент - фізична частина системи, що відповідає специфікації набору інтерфейсів і забезпечує їх реалізацію;

Вузол - фізичний елемент, який існує під час виконання програми і являє собою обчислювальний ресурс; зазвичай він володіє, як мінімум, деякою пам'яттю, а іноді і здатністю до обробки даних;

Прецедент, або варіант використання, - опис сукупності послідовностей дій (у тому числі варіантних), виконуваних системою, які викликають спостережувану зміну, що представляє інтерес для конкретного ектора;

Підсистема - сукупність елементів, з яких окремі складають специфікацію поведінки інших елементів.

Видимість

Одна з деталей, найбільш істотних для атрибутів і операцій класифікаторів, - їх видимість. Видимість властивості говорить про те, чи може воно використовуватися іншими класифікаторами. Природно, це має на увазі видимість самого класифікатора. Один класифікатор може " бачити" інший, якщо той перебуває в області дії першого і між ними існує явні або неявні відносини. В UML можна визначити три рівні видимості:

* Public (відкритий) - будь-який зовнішній класифікатор, який "бачить" даний, може користуватися його відкритими властивостями. Позначається знаком + (плюс) перед ім'ям атрибута або операції;

* Protected (захищений) - будь-який нащадок даного класифікатора може користуватися його захищеними властивостями. Позначається знаком # (дієз);

* Private (закритий) - тільки даний класифікатор може користуватися закритими властивостями. Позначається символом - (мінус).

ToolBar
-CURRENTSelection #COUNTTool : unsigned int
+getTool() -compact()

Видимість властивостей класифікатора визначають для того, щоб приховати деталі його реалізації і показати тільки ті особливості, які необхідні для здійснення обов'язків, продекларованих абстракцією. У цьому і полягає основна причина приховування інформації, без чого не обійтися при створенні надійної і гнучкої системи. Якщо символ видимості явно не вказаний, звичайно передбачається, що властивість є відкритим. Відносини дружності (Friendship) дозволяють класифікатору показувати іншим свої закриті деталі.

Область дії

Ще однією важливою характеристикою атрибутів і операцій класифікатора є область дії (Scope). Задаючи область дії деякої властивості, тим самим вказують, чи буде воно проявляти себе по-різному в кожному примірнику класифікатора, або одне і те ж значення властивості буде розділятися (тобто спільно використовуватися) усіма екземплярами. В UML визначено два види областей дії:

* Instance (екземпляр) - у кожного примірника класифікатора є власне значення даної властивості;

* Classifier (класифікатор) - всі примірники класифікатора спільно використовують загальне значення даної властивості.

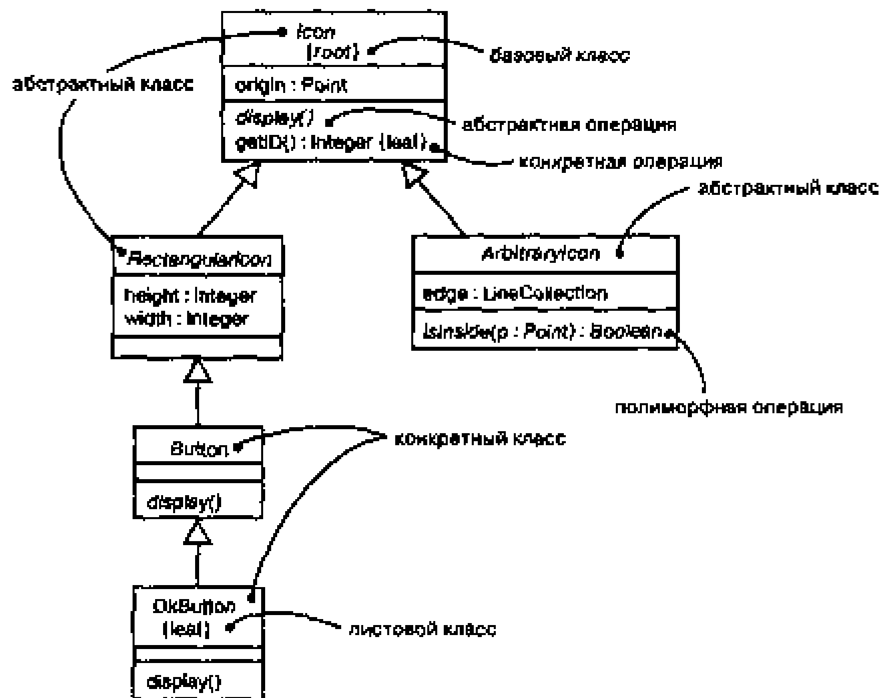
На рисунку показано, що ім'я властивості, яке має область дії classifier, підкреслюється. Якщо підкреслення відсутня, передбачається область дії instance.

Як правило, властивості модельованих класифікаторів мають область дії екземпляра. Властивості з областю дії класифікатора найчастіше застосовуються для опису закритих атрибутів, загальних для всіх примірників, наприклад для генерації унікальних ідентифікаторів або в операціях, що створюють екземпляри класу.

ToolBar
-CURRENTSelection #COUNTTool : unsigned int
+getTool() -compact()

Абстрактні, кореневі, листові і поліморфні елементи

Відносини узагальнення використовуються для моделювання ієрархії класів, верхні рівні якої займають загальні абстракції, а нижче знаходяться спеціалізовані. Всередині цієї ієрархії деякі класи визначають як абстрактні, тобто не мають безпосередніх примірників. В UML ім'я абстрактного класу пишуть курсивом. Наприклад, як видно з рисунку, абстрактними є класи Icon, Recfcangularlconi ArbitraryIcon. Навпаки, конкретними називаються класи, які можуть мати безпосередні екземпляри (на рис. це Button і OkButton).



При моделюванні класу часто виникає потреба поставити йому властивості, успадковані від більш загальних класів і, навпаки, надати можливість більш спеціалізованим класам успадковувати особливості даного. Така семантика легко забезпечується для класів засобами UML. Можна знайти й такі класи, у яких немає нащадків. Вони називаються листовими і задаються в UML за допомогою властивості leaf, написаного під ім'ям класу. Наприклад, клас OkButton на рис. є листовим і тому не має нащадків.

Рідше використовується, хоча і залишається досить корисною, можливість задати клас, який не має батьків. Такий клас називається кореневим і специфікується за допомогою властивості root, записаного під його ім'ям. На рис. кореневим є клас Icon. Якщо є декілька незалежних ієрархій спадкування, то початок кожної зручно позначати таким способом.

Операції можуть мати схожі властивості. Як правило, операції є поліморфними, - це означає, що в різних місцях ієрархії класів можна визначати операції з однаковими сигнатурами. При цьому ті операції, які визначені в класі - нащадку, перекривають дію тих, що визначені в батьківських класах. Коли під час виконання системи надходить якесь повідомлення, операція по його обробці викликається поліморфно, - іншими словами,

вибирається та, яка відповідає типу об'єкта. На рис. операції display і isInside являються поліморфними, a Icon:: display (), крім того, ще й абстрактна. Це означає, що вона неповна і реалізацію повинен надати нащадок. В UML імена абстрактних операцій пишуться курсивом, як і у випадку з класами. Навпаки, операція Icon:: getID () є листовою, на що вказує слово leaf. Це означає, що дана операція не поліморфна і не може бути перекрита іншою.

Кратність

При роботі з класом розумно припустити, що може існувати будь-яка кількість його примірників - якщо, звичайно, це не абстрактний клас, у якого взагалі не існує безпосередніх примірників, хоча в його нащадків їх може бути скільки завгодно. У деяких випадках, однак, число примірників класу потрібно обмежити. Найчастіше виникає необхідність задати клас, у якого

* Немає жодного примірника - тоді клас стає службовим (Utility), що містить лише атрибути та операції з областю дії класу;

* Рівно один примірник - такий клас називають синглетним (Singleton), а вказану кількість екземплярів;

* Довільне число примірників - варіант за замовчуванням.

Кількість примірників класу називається його кратністю. У загальному значенні кратність - це діапазон можливих кардинальних чисел деякої сутності (кратність зустрічається також у асоціацій). В UML кратність класу задається виразом, написаним у правому верхньому кутку його піктограми. Наприклад, як показано на рис., клас NetworkController являється синглетним, а у класу ControlRodimeется рівно три екземпляри.

Атрибути

На найвищому рівні абстракції, моделюючи структурні властивості класу (тобто атрибути), ви просто записуєте їхні імена. Зазвичай цього цілком достатньо, щоб читач міг зрозуміти загальне призначення моделі. На додаток до цього, як було описано вище, можна визначити видимість, район дії і кратність кожного атрибута. Крім того, можна задати тип, початкове значення і змінність атрибутів. А для позначення безлічі логічно пов'язаних атрибутів допустимо використовувати стереотипи.

Повна форма синтаксису атрибута в мові UML наступна:

```
[visibility]name[multiplicity][: type]
[= initial-value][{property-string}]
```

Нижче наведені приклади об'яв атрибутів:

- `origin` - тільки ім'я;
- `+ origin` - видимість і ім'я;
- `origin : Point` - ім'я і тип;
- `head : *Item` - ім'я і складний тип;
- `name [ 0..1 ] : String` - ім'я, кратність і тип;
- `origin : Point = (0,0)`- ім'я, тип і початкове значення;
- `id : Integer {frozen}`- ім'я і властивість.

Рязом з атрибутами можна використовувати три властивості:

- `changeable`(змінюваний) - обмежень на зміну значень атрибута не встановлено;
- `addOnly`(тільки додається) - дозволяється додавати нові значення для атрибутів з кратністю більше одиниці, але створене значення не може бути змінено або видалено;
- `frozen`(заморожений) - після ініціалізації об'єкта не можна змінювати значення його атрибутів.

Якщо явно не обумовлено протилежне, то всі атрибути змінювані (changeable). При моделюванні постійних або одноразово задаються атрибутів можна використовувати властивість frozen.

Операції

На найвищому рівні абстракції при моделюванні поведінкових характеристик класу (тобто його операцій і сигналів) ви просто записуєте їхні імена. Це зазвичай буває достатньо, щоб читач міг зрозуміти загальне призначення моделі. Крім цього, як описувалося в попередніх розділах, ви можете визначити видимість і область дії кожної операції. Можна задати також її параметри, тип значення, семантику паралелізму і деякі інші властивості. Ім'я операції спільно з її параметрами (включаючи тип значення, якщо таке є) називають сигнатурою операції. Для опису множини логічно пов'язаних операцій, таких, наприклад як допоміжні функції, можна використовувати стереотипи.

Повний синтаксис операції в UML такий:

```
[visibility] name [(parameter-list)] [: return-type]
[ {property-string}]
```

Нижче наводяться деякі допустимі оголошення операцій:

- `display`- тільки ім'я;
- `+ display`- видимість та ім'я;
- `set (n: Name, s:String)`- ім'я і параметри;
- `get ID() : Integer`- ім'я та повертаєме значення;
- `restart() {guarded}`- ім'я та властивість.

Сигнатура операції може містити нуль або більше параметрів, кожен з яких має наступний синтаксис:

```
[direction] name : type [= default-value].
```

Параметр direction може приймати будь-яке з нижчеперелічених значень:

* In - вхідний параметр, який не може бути модифікований;

* Out - виходить параметр, який може бути змінений, щоб передати інформацію викликала процедурою;

* Inout - вхідний параметр, який може бути змінений.

Крім описаної раніше властивості leaf для операцій визначені ще чотири властивості.

Шаблони класів

Шаблоном називається параметризований елемент. У таких мовах програмування, як C++ або Ada, передбачена можливість створювати шаблони класів, що визначають сімейства класів (можна задавати також шаблони функцій, що визначають сімейства функцій). Параметрами шаблону можуть бути класи, об'єкти або значення. Шаблон не можна використовувати безпосередньо; спочатку його потрібно інстанцірувати, тобто конкретизувати. Процес інстанцірування - це скріплення формальних параметрів шаблону з фактичними. В результаті з шаблону класу виходить конкретний клас, з яким можна працювати як з будь-яким іншим.

В UML визначено чотири стандартних стереотипа, що застосовуються до класів:

* Metaclass -визначає класифікатор, всі об'єкти якого є класами;

* Powertype -визначає класифікатор, всі об'єкти якого є нащадками даного батька;

* Stereotype -визначає, що даний класифікатор є стереотипом, який можна застосувати до інших елементів;

* Utility -визначає клас, атрибути та операції якого знаходяться в області дії всіх класів.