Класифікація мов програмування.

Розробка програмного забезпечення для комп’ютерів в даний час виготовляється майже виключно за допомогою мов програмування високого рівня. Дані мови являють собою систему мнемонічних значень з жорстко заданим синтаксисом та симантикою, котрі зрозумілі людині і перетворюються в послідовність машинних команд за допомогою спеціальної програми-транслятора. Існує безліч класифікацій мов програмування. Розглянемо класифікації за стилем програмування.

Стиль – це сукупність правил, що лежать в основі синтаксису і семантики мови програмування. Розрізняють наступні стилі:

1. неструктурний;
2. структурний;
3. логічний;
4. об’єктно-орієнтований ;
5. функціональний.

Розглянемо вище перераховані стилі програмування:

Неструктурне програмування дозволяє використання в явному вигляді команди безумовного переходу (у більшості мов – GOTO). Типове представлення неструктурних мов – ранні версії Бейсіка та Фортрану. Однак в мовах високого рівня наявність команд переходу тягне за собою масу недоліків: програма перетворюється на "локшину" з нескінченними переходами вверх-вниз, її дуже складно супроводжувати та модифікувати. Фактично неструктурний стиль програмування не дозволяє розробляти великі та складні програмні проекти.

Структурный стиль був розроблений в середині 60-х - початку 70-х рр.

Завдання розбивається на велику кількість дрібних підзадач, кожна з яких вирішується своєї процедурою або функцією (декомпозиція задачі). При цьому проектування програми йде за принципом зверху вниз: спочатку визначаються необхідні для вирішення програми модулі, їх входи і виходи, а потім вже ці модулі розробляються. Такий підхід разом з локальними іменами змінних дозволяє розробляти проект силами великого числа програмістів.

Використовуються лише три керуючі конструкції: послідовне виконання, розгалуження і цикл Дана обставина дозволяє при наявності відповідних операторів виключити з мови команду переходу GOTO.

Прикладом мови структурного програмування є Паскаль.

Логічне програмування являє собою спробу покласти на програміста тільки постановку задачі, а пошуки шляхів її вирішення надати транслятора. Логічні мови (Пролог, Симула) мають спеціальні конструкції для опису об'єктів і зв'язків між ними. Наприклад, якщо дано:

брати мають одного батька

"Джон – батько Джека", "Майк – брат Джека", то система логічного програмування повинна зробити висновок:

"Джон – батько Майка. "

В основі функціонального стилю лежить поняття функції як "чорної скриньки", що має вектор параметрів (аргументів) *P* на вході і результат *r* (скаляр) на виході: *F (P) = r.* У функціональних мовах програмування відсутні оператори: всі дії, в тому числі і керуючі конструкції, виконуються за допомогою викликів функцій.

Об'єктно-орієнтоване (ГО) програмування, розроблене в середині 70-х рр. Кернiганом і Річчі і реалізоване в ГО-версіях мов Сі і Паскаль, являє собою відображення об'єктів реального світу, їх властивостей (атрибутів) і зв'язків між ними за допомогою спеціальних структур даних. При об'єктно-орієнтованому підході для об'єкта створюється своя структура даних (клас), що містить як властивості об'єкта (поля), так і процедури для управління об'єктом (методи).

ГО-підхід історично прийшов на зміну структурному підходу, коли програма розбивалася на окремі фрагменти - підпрограми, які отримували вихідні дані, відповідно до своєї логікою обробляли їх і повертали результат.

Для зберігання даних між підпрограмами використовувалися змінні. Недоліком такого підходу була відкритість даних.

На будь-якому етапі дані могли бути змінені, що призвело б до логічної помилки навіть при правильному описі логіки кожної підпрограми окремо. Безумовно, було розроблено безліч способів скорочення такого роду помилок, однак це призводило до додаткових накладних витрат.

Ще один недолік структурного підходу - концептуальне невідповідність предметної області.

Складно уявити чому, наприклад, ці 50 функцій і 30 змінних, можливо описані в різних файлах, являють собою модель баштового крана.

При об'єктно-орієнтованому підході програма являє собою опис об'єктів, їх властивостей (або атрибутів), класів (або сукупностей), відносин між ними, способів їх взаємодії і операцій над об'єктами (або методів).

До найбільш важливим інструментальних засобів ООП відносяться:

- абстрагування;
- інкапсуляція;
- успадкування;
- поліморфізм.

Основні поняття ООП

Клас

Уявіть собі, що ви проектуєте автомобіль. Ви знаєте, що автомобіль повинен містити двигун, підвіску, дві передніх фари, 4 колеса, тощо. Ще ви знаєте, що ваш автомобіль повинен мати можливість набирати і зменшувати швидкість, здійснювати поворот і рухатися заднім ходом. І, що найголовніше, ви точно знаєте, як взаємодіє двигун і колеса, згідно яким законам рухається распредвал і колінвал, а також як влаштовані диференціали. Ви впевнені в своїх знаннях і починаєте проектування.

Ви описуєте всі запчастини, з яких складається ваш автомобіль, а також те, яким чином ці запчастини взаємодіють між собою. Крім того, ви описуєте,

що повинен зробити користувач, щоб машина загальмувала, або включився дальнє світло фар. Результатом вашої роботи буде деякий ескіз. Ви тільки що розробили те, що в ООП називається клас.

Клас – це спосіб опису сутності, що визначає стан і поведінку, залежне від цього стану, а також правила для взаємодії з даної сутністю (контракт).

З точки зору програмування клас можна розглядати як набір даних (полів, атрибутів, членів класу) і функцій для роботи з ними (методів).

З точки зору структури програми, клас є складним типом даних.

У нашому випадку, клас буде відображати сутність – автомобіль. Атрибутами класу будуть двигун, підвіска, кузов, чотири колеса тощо. Методами класу буде «відкрити двері», «натиснути на педаль газу», а також «закачати порцію бензину з бензобака в двигун». Перші два методи доступні для виконання інших класів (зокрема, класу «Водій»). Останній описує взаємодії всередині класу і не доступний користувачеві.

Об'єкт

Ви відмінно попрацювали і машини, розроблені за вашими кресленнями, сходять з конвеєра. Ось вони, стоять рівними рядами на заводському подвір'ї. Кожна з них точно повторює ваші креслення. Всі системи взаємодіють саме так, як ви спроектували. Але кожна машина унікальна. Вони всі мають номер кузова і двигуна, але всі ці номери різні, автомобілі розрізняються кольором, а деякі навіть мають лиття замість штампованих дисків. Ці автомобілі, по суті, є об'єктами вашого класу.

Об'єкт (екземпляр) – це окремий представник класу, який має конкретний стан і поведінку, повністю визначається класом.

Говорячи простою мовою, об'єкт має конкретні значення атрибутів і методи, що працюють з цими значеннями на основі правил, заданих в класі. В даному прикладі, якщо клас - це деякий абстрактний автомобіль з «світу ідей», то об'єкт - це конкретний автомобіль, що стоїть у вас під вікнами.

Інтерфейс

Коли ми підходимо до автомата з кавою або сідаємо за кермо, ми починаємо взаємодію з ними. Зазвичай, взаємодія відбувається за допомогою деякого набору елементів: щілину для приймання монеток, кнопка вибору напою і відсік видачі склянки в кавовому автоматі; кермо, педалі, важіль коробки перемикання передач в автомобілі. Завжди існує деякий обмежений набір елементів управління, з якими ми можемо взаємодіяти.

Інтерфейс – це набір методів класу, доступних для використання іншими класами.

Очевидно, що інтерфейсом класу буде набір всіх його публічних методів в сукупності з набором публічних атрибутів. По суті, інтерфейс специфікує клас, чітко визначаючи всі можливі дії над ним.

Гарним прикладом інтерфейсу може служити панель приладів автомобіля, яка дозволяє викликати такі методи, як збільшення швидкості, гальмування, поворот, перемикання передач, включення фар, і т.п. Тобто всі дії,

які може здійснити інший клас (в нашому випадку – водій) при взаємодії з автомобілем.

При описі інтерфейсу класу дуже важливо дотримати баланс між гнучкістю і простотою. Клас з простим інтерфейсом буде легко використовувати, але будуть існувати завдання, які за допомогою нього вирішити буде не під силу. У той же час, якщо інтерфейс буде гнучким, то, швидше за все, він буде складатися з досить складних методів з великою кількістю параметрів, які будуть дозволяти робити дуже багато, але використання його буде пов'язане з великими труднощами і ризиком помилитися, щось переплутавши.

Прикладом простого інтерфейсу може служити машина з коробкою-автоматом. Освоїти її управління дуже швидко зможе будь-яка блондинка, яка закінчила двотижневі курси водіння. З іншого боку, щоб освоїти управління сучасним пасажирським літаком, необхідно кілька місяців, а то й років наполегливих тренувань. Не хотів би я бути на борту Боїнга, яким керує людина, яка має двотижневий льотний стаж. З іншого боку, ви ніколи не змусите автомобіль піднятися в повітря і перелетіти з Києва до Вашингтона.

Інкапсуляція

Уявімо на хвилинку, що ми опинилися в кінці позаминулого століття, коли Генрі Форд ще не придумав конвеєр, а перші спроби створити автомобіль стикалися з критикою влади з приводу того, що ці коптять монстри забруднюють повітря і лякають коней. Уявімо, що для управління першим паровим автомобілем необхідно було знати, як влаштований паровий котел, постійно підкидати вугілля, стежити за температурою, рівнем води. При цьому для повороту коліс використовувати два важелі, кожен з яких повертає одне колесо окремо. Думаю, можна погодитися з тим, що водіння автомобіля того часу було дуже незручним і важким заняттям.

Тепер повернемося в сьогоднішній день до сучасних чудес автопрому з коробкою-автоматом. Насправді, по суті, нічого не змінилося. Бензонасос все так же поставляє бензин в двигун, диференціали забезпечують поворот коліс на розрізняються кути, колінвал перетворює поступальний рух поршня в обертальний рух коліс. Прогрес в іншому. Зараз всі ці дії приховані від користувача і дозволяють йому крутити кермо і натискати на педаль газу, не замислюючись, що в цей час відбувається з інжектором, дросельною заслінкою і розподілвалом. Саме приховування внутрішніх процесів, що відбуваються в автомобілі, дозволяє ефективно його використовувати навіть тим, хто не є професіоналом-автомеханіком з двадцятирічним стажем. Це приховування в ООП носить назву інкапсуляції.

Інкапсуляція – це властивість системи, що дозволяє об'єднати дані і методи, що працюють з ними, в класі і приховати деталі реалізації від користувача.

Інкапсуляція нерозривно пов'язана з поняттям інтерфейсу класу. По суті, все те, що не входить в інтерфейс, инкапсулируется в класі.

Абстракція

Уявіть, що водій їде в автомобілі по жвавій ділянці руху. Зрозуміло, що в цей момент він не буде замислюватися про хімічний склад фарби автомобіля, особливості взаємодії шестерень в коробці передач або впливу форми кузова на швидкість (хіба що, автомобіль коштує в глухий пробці і водієві абсолютно нічим зайнятися). Однак, кермо, педалі, покажчик повороту (ну і, можливо, попільничку) він буде використовувати регулярно.

Абстрагування – це спосіб виділити набір значущих характеристик об'єкта, виключаючи з розгляду незначущі. Відповідно, абстракція - це набір всіх таких характеристик.

Якби для моделювання поведінки автомобіля доводилося враховувати хімічний склад фарби кузова і питому теплоємність лампочки підсвічування номерів, ми ніколи б не дізналися, що таке NFS.

Поліморфізм

Будь-яке навчання водінню не мало б сенсу, якби людина, яка навчилася водити, скажімо, ВАЗ 2106 не міг потім водити ВАЗ 2110 або BMW X3. З іншого боку, важко уявити людину, яка змогла б нормально керувати автомобілем, в якому педаль газу знаходиться лівіше педалі гальма, а замість керма - джойстик.

Вся справа в тому, що основні елементи управління автомобіля мають одну і ту ж конструкцію і принцип дії. Водій точно знає, що для того, щоб повернути ліворуч, він повинен повернути кермо, незалежно від того, є там гідропідсилювач чи ні.

Якщо людині треба доїхати з роботи до дому, то він сяде за кермо автомобіля і буде виконувати одні і ті ж дії, незалежно від того, який саме тип автомобіля він використовує. По суті, можна сказати, що всі автомобілі мають один і той же інтерфейс, а водій, абстрагуючись від сутності автомобіля, працює саме з цим інтерфейсом. Якщо водієві доведеться їхати по німецькому автобану, він, ймовірно вибере швидкий автомобіль з низькою посадкою, а якщо має бути повертатися з віддаленого маральніка в Гірському Алтаї після дощу, швидше за все, буде обраний УАЗ з армійськими мостами. Але, незалежно від того, яким чином буде реалізовуватися рух і внутрішнє функціонування машини, інтерфейс залишиться колишнім.

Поліморфізм - це властивість системи використовувати об'єкти з однаковим інтерфейсом без інформації про тип і внутрішню структуру об'єкта.

Наприклад, якщо ви читаєте дані з файлу, то, очевидно, в класі, що реалізує файловий потік, буде присутній метод схожий на наступний: byte [] readBytes (int n);

Припустимо тепер, що вам необхідно зчитувати ті ж дані з сокета. У класі, що реалізує сокет, також буде присутній метод readBytes. Досить замінити в вашій системі об'єкт одного класу на об'єкт іншого класу, і результат буде досягнутий.

При цьому логіка системи може бути реалізована незалежно від того, чи будуть дані прочитані з файлу або отримані по мережі. Таким чином, ми абстрагуємося від конкретної спеціалізації отримання даних і працюємо на

рівні інтерфейсу. Єдина вимога при цьому - щоб кожен використовуваний об'єкт мав метод readBytes.

Наслідування

Уявімо себе, на хвилину, інженерами автомобільного заводу. Нашим завданням є розробка сучасного автомобіля. У нас вже є попередня модель, яка відмінно зарекомендувала себе протягом багаторічного використання. Все б добре, але часи і технології змінюються, а наш сучасний завод повинен прагнути підвищувати зручність і комфорт, що випускається і відповідати сучасним стандартам.

Нам необхідно випустити цілий модельний ряд автомобілів: седан, універсал і малолітражний хетч-бек. Очевидно, що ми не збираємося проектувати новий автомобіль з нуля, а, взявши за основу попереднє покоління, внесемо ряд конструктивних змін. Наприклад, додамо гідропідсилювач керма і зменшимо зазори між крилами і кришкою капота, поставимо протитуманні ліхтарі. Крім того, в кожній моделі буде змінена форма кузова.

Очевидно, що всі три модифікації матимуть більшість властивостей колишньої моделі (старий добрий двигун 1970 року народження, непробивна ходова частина, що зарекомендувала себе відмінним чином на вітчизняних дорогах, коробку передач і т.д.). При цьому кожна з моделей буде реалізувати деяку нову функціональність або конструктивну особливість. В даному випадку, ми маємо справу зі спадщиною.

Спадкування - це властивість системи, що дозволяє описати новий клас на основі вже існуючого з частково або повністю позичає функціональністю. Клас, від якого виробляється спадкування, називається базовим або батьківським. Новий клас - нащадком, спадкоємцем або похідним класом.

Необхідно відзначити, що похідний клас повністю задовольняє специфікації батьківського, проте може мати додаткову функціональність. З точки зору інтерфейсів, кожен похідний клас повністю реалізує інтерфейс батьківського класу. Зворотне не вірно.

Дійсно, в нашому прикладі ми могли б зробити з новими автомобілями все ті ж дії, що і зі старим: збільшити або зменшити швидкість, повернути, включити сигнал повороту. Однак, додатково у нас би з'явилася можливість, наприклад, включити протитуманні ліхтарі.

Відсутність зворотної сумісності означає, що ми не повинні очікувати від старої моделі коректної реакції на такі дії, як включення противотуманок (яких просто немає в даній моделі).

Об'єктно-орієнтований аналіз і проектування

Широке поширення методології ООП вплинуло на процес розробки програм. Зокрема, процедурно-орієнтована декомпозиція програм поступилася місцем об'єктно-орієнтованої декомпозиції, при якій окремими структурними одиницями програми стали є не процедури і функції, а класи і об'єкти з відповідними властивостями і методами. Як наслідок, програма перестала бути послідовністю визначених на етапі кодування дій, а стала подієво-керованої.

Найбільш істотним обставиною у розвитку методології ООП стало усвідомлення того факту, що процес написання програмного коду може бути відділений від процесу проектування структури програми. Дійсно, до того як почати програмування класів, їх властивостей і методів, необхідно визначити, чим же є самі ці класи. Більш того, потрібно дати відповіді на такі питання, як: скільки і які класи потрібно визначити для вирішення поставленого завдання, які властивості і методи необхідні для додання класів необхідного поводження, а також встановити взаємозв'язок між класами.

Ця сукупність завдань не стільки пов'язана з написанням коду, скільки з загальним аналізом вимог до майбутньої програмі, а також з аналізом конкретної предметної області, для якої розробляється програма. Всі ці обставини призвели до появи спеціальної методології, що отримала назву методології об'єктно-орієнтованого аналізу і проектування (ООАП).

Життєвий цикл

Поділ процесу розробки складних програмних додатків на окремі етапи сприяло становленню концепції життєвого циклу програми. Під життєвим циклом (ЖЦ) програми розуміють сукупність взаємопов'язаних і наступних у часі етапів:

- Аналізу предметної області і формулювання вимог до програми
- Проектування структури програми
- Реалізації програми в кодах (власне програмування)
- Впровадження програми
- Супроводження програми
- Відмови від використання програми

На етапі аналізу предметної області та формулювання вимог здійснюється визначення функцій, які повинна виконувати розробляється програма, а також концептуалізація предметної області. Результатом даного етапу повинна бути деяка концептуальна схема, яка містить опис основних компонентів і тих функцій, які вони повинні виконувати.

Етап проектування структури програми полягає в розробці детальної схеми майбутньої програми, на якій вказуються класи, їх властивості та методи, а також різні взаємозв'язку між ними. Результатом даного етапу повинна стати деталізована схема програми, на якій вказуються всі класи і взаємозв'язку між ними в процесі функціонування програми. Згідно з методологією ООАП, саме ця схема повинна слугувати вихідною інформацією для написання програмного коду.

Етап програмування навряд чи потребує уточнення, оскільки є найбільш традиційним для програмістів. Поява інструментаріїв швидкої розробки додатків (Rapid Application Development, RAD) дозволило істотно скоротити час, і витрати на виконання цього етапу. Результатом даного етапу є програмний додаток, яке володіє необхідною функціональністю і здатне вирішувати потрібні завдання в конкретній предметній області.

Етапи впровадження та супроводу програми пов'язані з необхідністю налаштування і конфігурації середовища програми, а також з усуненням

виникли в процесі її використання помилок. Іноді в якості окремого етапу виділяють тестування програми, під яким розуміють перевірку працездатності програми на деякій сукупності вихідних даних або при деяких спеціальних режимах експлуатації. Результатом цих етапів є підвищення надійності програмного додатка, що виключає виникнення критичних ситуацій або нанесення збитку компанії, що використовує цю програму.

Методологія ООАП тісно пов'язана з концепцією автоматизованої розробки програмного забезпечення (Computer Aided Software Engineering, CASE). Будь-яке CASE-засіб реалізує ту чи іншу графічну нотацію. Однією з таких нотацій є уніфікована мова моделювання (Unified Modelling Language, UML).

Мова UML є общецелевого мову візуального моделювання, який розроблений для специфікації, візуалізації, проектування та документування компонентів програмного забезпечення, бізнес-процесів і інших систем.