РОЗШИРЕНИЙ СПОСІБ ОПИСУ ВІДНОШЕНЬ МІЖ ОБ'ЄКТАМИ В ОБ'ЄКТНО-ОРІЄНТОВАНІЙ ПАРАДИГМІ ПРОГРАМУВАННЯ

УДК 004.432

СЕМЕНЯКІН Володимир Сергійович
магістрант кафедри програмного забезпечення НТУУ «Київський політехнічний інститут».
Наукові інтереси: об'єктно-орієнтоване програмування, філософія програмування, імітаційне моделювання.
e-mail: semenyakinVSG@gmail.com

ЗАБОЛОТНЯ Тетяна Миколаївна
к.т.н., старший викладач кафедри програмного забезпечення НТУУ «Київський політехнічний інститут».
Наукові інтереси: сучасні інформаційні технології, об'єктно-орієнтоване програмування, автоматизована обробка природномовних текстових даних
e-mail: tetiana.zabolotnia@gmail.com

ВСТУП

На початку комп'ютерної ери задачею мов програмування був опис дій, які повинні виконуватись ЕОМ, у вигляді, зрозумілому для програміста [1]. Для реалізації цієї задачі було створено різноманітні асемблери, що давали змогу записувати команди процесора не у вигляді кодів команд і адрес пам'яті, а за допомогою символьного запису.

Коли розробка програмного забезпечення (ПЗ) з наукової/технічної задачі перетворилась на комерційну, на ринку ПЗ зросла конкуренція і виникла необхідність створення великих програмних систем за обмежений проміжок часу. Відтепер код мав задовольняти умові легкого розуміння записаної ним семантики у стислий часовий відрізок, а мова програмування мала виконувати точне і прозоре відображення семантики у символьний вигляд.

З початку роль мови високого рівня виконували мова програмування С та інші процедурні мови, що оперували поняттями *змінних* (даних, що зберігають стан програмної системи) та *функцій* (реалізацій алгоритмів, що впливають на стан). Згодом, коли розробники парадигм програмування звернули увагу на те, що при пізнанні об'єктів реального світу людина оперує поняттями *стану об'єкту* (тим, як він сприймається) і *методів роботи з ним* (тим, які дії над ним можливо виконувати), виникла об'єктно-орієнтована парадигма програмування. Опис сутностей програмної системи в об'єктно-орієнтованій (далі - ОО) парадигмі програмування виявився досить стислим, зручним і точним, і, за рахунок цього, більшість сучасних мов програмування, якими виконується промислова розробка програмних систем, реалізують саме ОО парадигму [1].

Як було вказано вище, ОО парадигма програмування передбачає точний опис об'єктів реального світу, але вона не дає змоги стисло характеризувати відношення між об'єктами. Так, наприклад, при необхідності змінити перелік методів класу, доступних іншим класам, необхідно застосовувати проміжні сутності, що виконують диспетчеризацію запитів до об'єктів класу; опис додаткових сутностей захаращує код програми і потребує більшого часу на його аналіз.

Таким чином, дослідження можливих шляхів модифікації опису відношень між об'єктами у ОО парадигмі є актуальним.

1. ПОСТАНОВКА ЗАДАЧІ

1.1. Термінологія

Для зручності будемо називати об'єкт, що лежить у фокусі уваги, *фокальним об'єктом* (відповідно, його клас - *фокальним класом*), а усі інші об'єкти – *об'єктами навколишнього середовища (*відповідно, їх класи - *класами навколишнього середовища)*. При цьому приймається, що нащадки фокального класу і базові класи фокального об'єкту також належать множині класів навколишнього середовища (нащадок і базовий клас пов'язані відношеннями композиції).

Станом будь-якого об'єкту будемо називати сукупність значень усіх його полів. У даній статті приймається, що:

– зміна стану та отримання інформації щодо стану будь-якого об'єкту відбувається лише через його методи;

– у програмі не можуть існувати автономні функції (функції, що не належать жодному класу).

Під *відношеннями* між об'єктами будемо розуміти спосіб, у який вони взаємодіють. Існуючі у рамках ОО парадигми типи відношень, що розглядаються у даній статті, пропонується згрупувати наступним чином:

– структурні відношення описують міру залежності між класами. До цієї групи можна включити відношення is-a ("є") та has-a ("містить") [2]. Відношення останнього типу поділяють на композицію (відношення володіння об'єктом) і агрегацію (відношення включення об'єкту, знання про об'єкт) [2]. У даній статті розглянуто лише описані два типи відношення has-a, оскільки відношення is-a ідеологічно відрізняється від has-a і спосіб його опису потребує окремого дослідження. Користуючись запропонованою вище термінологією, можна сформулювати принцип організації структурних відношень таким чином: "фокальний об'єкт повинен належити (мати відношення типу композиція) до найменшого переліку об'єктів навколишнього середовища";

– відношення доступності описують перелік методів фокального об'єкту, які доступні об'єктам навколишнього середовища. В ОО парадигмі відношення цього типу описуються за допомогою механізму інкапсуляції [1]. Користуючись запропонованою вище термінологією, можна сформулювати принцип організації відношень доступності наступним чином: "кожен об'єкт навколишнього середовища, що взаємодіє з фокальним об'єктом, повинен мати доступ до мінімального переліку методів фокального об'єкту, що безпосередньо стосуються логіки, за якою відбувається взаємодія".

1.2. Обгрунтування вибору мети дослідження

Розглянемо способи опису двох вищезгаданих груп відношень об’єктів у сучасних ОО мовах програмування.

Опис структурних відношень. У ОО програмуванні єдиною мовною одиницею, через яку може здійснюватись доступ до фокального об'єкту, є посилання на нього. З початку посилання виступало виключно числом, що описує адресу фокального об'єкту у пам'яті ЕОМ і, відповідно, посилання не виконувало опису структурних відношень між об'єктами. Однак з виникненням поняття *автоматичного підрахунку посилань* [7] останні стали визначатись більш складними об'єктами, а їх запис став дескриптивним з точки зору опису структурних відношень між об'єктами. У сучасних мовах програмування автоматичний підрахунок посилань виконується або у рамках існуючого синтаксису (boost sharedPointer [8], за допомогою шаблонів C++), або шляхом введення нових слів мови (*auto reference counting* (*ARC*) [9], за допомогою слів, що належать мові). Таким чином, у сучасних мовах програмування опис структурних відношень дійсно реалізується, але у даній статті досліджується, яким чином можна зробити опис цих відношень більш уніфікованим.

Опис відношень доступності. Більшість сучасних ОО мов програмування включають у себе наступні рівні доступу до методів фокального об'єкту:

- public - доступ мають усі об'єкти навколишнього середовища і методи фокального об'єкту;
- protected – доступ мають методи фокального об'єкту і об'єкт-нащадок класу фокального об'єкту;
- private - доступ мають виключно методи фокального об'єкту.

Примітка: у мові програмування C++ додатково існує механізм, що здійснює обмеження доступу до методів фокального об'єкту - концепція константних методів класу та константних об’єктів [1].

Слід зазначити, що існуючі мови програмування дозволяють описати рівні доступу лише у рамках інтер-

фейсу класу і не мають можливості частково обмежити чи розширити множини методів фокального класу, доступних для об'єктів навколишнього середовища. При необхідності динамічно змінювати ці множини використовуються групи об'єктів, організовані з використанням шаблону проектування *proxy* [2]. Реалізація зміни множини доступних методів фокального об'єкту у подібний спосіб є громіздкою і не зручною.

1.3. Постановка мети та задач дослідження

Таким чином, виходячи із наведених вище аргументів, **метою** даного дослідження стало забезпечення можливості динамічного змінювання структурних відношень та відношень доступності без застосування проміжних сутностей за рахунок розробки нового способу опису міжоб'єктних відношень в ОО парадигмі програмування.

У відповідності до поставленої мети **задачами дослідження** є:

- вивчення способів побудови відношень між об'єктами в ОО мовах програмування на предмет виявлення можливостей щодо забезпечення їх динамічного змінювання;
- визначення критеріїв оцінювання якості реалізації опису відношень між об'єктами в ОО парадигмі;
- розробка способу опису відношень доступності;
- розробка способу подання метаданих класу, що описують структурні відношення і відношення доступності між об'єктами;
- аналіз ефективності розробленого способу опису міжоб'єктних відношень на прикладі його можливої реалізації у межах модифікованої мови програмування C++.

2. РОЗШИРЕНИЙ СПОСІБ ОПИСУ МІЖОБ'ЄКТНИХ ВІДНОШЕНЬ

Критерії оцінювання якості реалізації опису відношень між об'єктами у ОО парадигмі

Перш ніж говорити про новий спосіб опису взаємозв'язку між об'єктами, необхідно сформулювати критерії, за якими буде виконуватись його порівняння з існуючими рішеннями. Для оцінювання якості пропонується застосувати наступні критерії:

– лаконічність опису базової функціональності. Під базовою функціональністю мається на увазі функціональність, закладена при описі класу, функціональність "за замовчуванням", без врахування можливого рефакторингу [3]. Під лаконічністю розуміється стислість опису функціональності; великий обсяг коду негативно впливає на час його написання і потребує більшого часу на його читання. Оцінка за даним критерієм може бути виражена в обсязі коду: чим він менший, тим вища якість способу опису відношень між об'єктами;

– лаконічність модифікації функціональності. Під модифікацією функціональності мається на увазі рефакторинг коду – зміна поглядів на предметну область чи на спосіб її опису. Оцінка за даним критерієм також може бути також виражена в обсязі коду;

– дескриптивність – суб'єктивний показник: наскільки зрозумілим і, як наслідок, зручним є використання опису відношення між об'єктами. Для отримання якісної оцінки за цим критерієм потрібно проводити опитування користувачів мови програмування, що реалізує спосіб опису взаємодії між об'єктами. Для формулювання чисельного визначення оцінок за даним критерієм потрібно виконати окреме дослідження, що буде проведене у майбутньому.

2.1. Селективний спосіб реалізації інкапсуляції

При використанні способу реалізації інкапсуляції, розробленого у результаті даного дослідження, посилання розглядається як проміжна сутність, через яку виконується взаємодія з об'єктом. Нижче наведено основні принципи, які пропонується покласти в основу нового способу реалізації інкапсуляції:

Принцип розподілу ролей. Ідея інкапсуляції полягає в описі певних груп методів фокального об'єкту, що доступні певним об'єктам навколишнього середовища. Можна сказати, що кожна така група визначає спосіб, у який об'єкт навколишнього середовища взаємодіє з фокальним об'єктом. У цьому сенсі далі будемо казати, що кожен об'єкт навколишнього середовища має певну **роль** при взаємодії з фокальним об'єктом.

Примітка: поняття ролі, що вводиться тут, не має прямого відношення до поняття ролі у рольорієнтованому програмуванні [4] або у парадигмі DCI (Data context interaction) [3]. Обидва вказані підходи до програмування розглядають роль як прояв об'єкту у певному контексті, у той час як у способі, що пропонується, роль розглядається як характеристика механіз-

мів доступу до фокального об'єкту з боку об'єкту навколишнього середовища.

Принцип інкапсуляції на рівні посилань. Роль, що описує особливості взаємодії з фокальним об'єктом, визначається для кожного посилання окремо. Тобто деякий об'єкт навколишнього середовища може зберігати декілька посилань на фокальний об'єкт, і кожне з посилань може описувати свою роль, в межах якої відбувається взаємодія з фокальним об'єктом через нього.

На основі поданих принципів для опису інкапсуляції об'єкту з використанням нового способу пропонується виконання наступних кроків:

1. Описати клас. Для цього необхідно описати:

1.1. Поля класу (стан об'єкту) та його методи.

1.2. Основні ролі, які можуть бути використані для взаємодії з класом (детальніше про спосіб опису метаданих класу див. п.2.2.).

2. Отримати доступ до об'єкту класу через посилання:

2.1. У будь-якому місці, де реалізується взаємодія з фокальним об'єктом, описати посилання, через яке буде відбуватись взаємодія, специфікувавши його роллю, котра буде виконуватись через нього. При цьому роль може бути створена на місці опису посилання, якщо це необхідно.

2.2. Виконати взаємодію з фокальним об'єктом через посилання.

Запропонований спосіб реалізації інкапсуляції було вирішено назвати *селективним* (від англ. *select* – "обирати"), оскільки він дозволяє обирати методи, до яких можливо отримати доступ після декларації груп доступу в описі класу.

Ілюстрацію селективного способу реалізації інкапсуляції можна побачити на рис. 1.

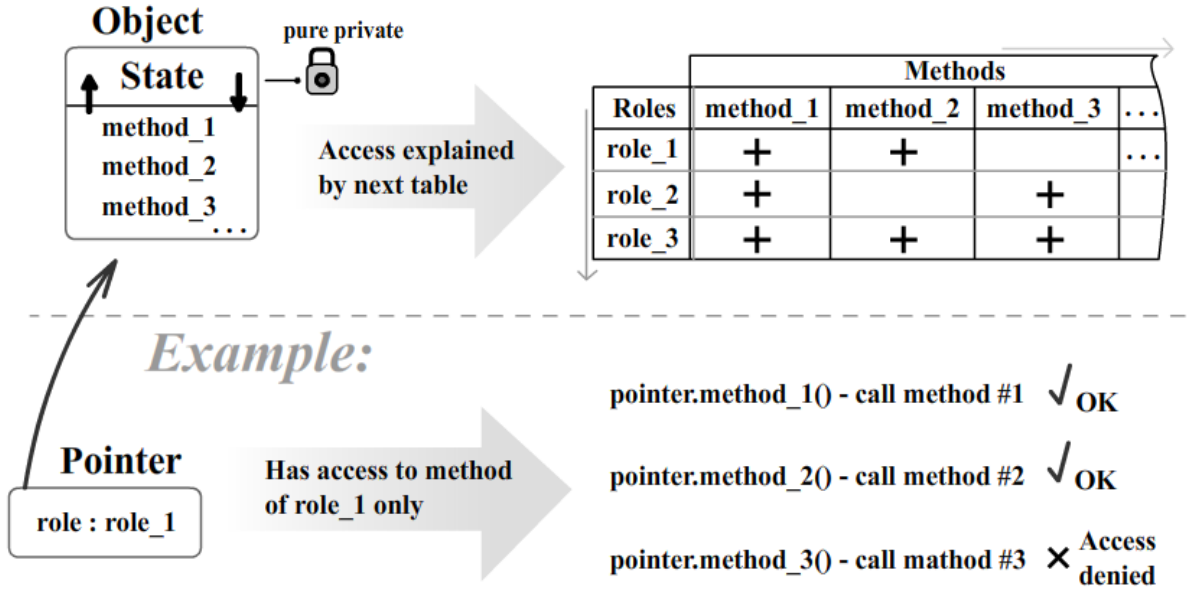

Рисунок1 – Діаграма, що ілюструє селективний спосіб організації інкапсуляції

2.2. Опис ролей за допомогою атрибутів класу

При описі ролей у будь-якій мові програмування доцільно використовувати додаткові мовні конструкції, які безпосередньо описують особливості відношень між об'єктами. Оскільки ці конструкції беруть на себе опис допоміжних особливостей класу, будемо їх називати *метаданими* класу.

Деякі сучасні мови програмування вже зараз підтримують концепцію метаданих. Так, наприклад, у сімействі мов для віртуальної машини .NET існує поняття *атрибутів класу* [5], у мові програмування java для подібних цілей введено поняття *анотацій* [6].

У наступних пунктах розглянуто можливу реалізацію опису метаданих класу у межах модифікованої мови програмування C++. У якості одиниці опису метаданих приймається поняття *атрибуту C++* (тут

приймається термінологія .NET). Під атрибутом розуміється певний ідентифікатор (ім'я атрибуту), котрий має тип (тип атрибуту), і у відповідність якому ставиться перелік ідентифікаторів-аргументів, зміст яких визначається типом атрибуту. При цьому у якості аргументу атрибуту можуть використовуватись інші атрибути.

2.3. Опис відношень за допомогою атрибутів

Як було вказано у пункті 1.2, існуючи мови програмування підтримують опис структурних відношень або за допомогою спеціальних слів мови, або у рамках існуючого синтаксису. Головним недоліком обох вказаних способів є те, що вони дозволяють описувати лише структурні відношення, не залишаючи можливості додавати до посилань нові атрибути.

У даній статті пропонується абстрагувати поняття структурного відношення і ввести у перелік атрибутів класу атрибути для опису відношень цього типу.

3. ДОСЛІДЖЕННЯ МОЖЛИВОЇ РЕАЛІЗАЦІЇ ЗАПРОПОНОВАНОГО СПОСОБУ ОПИСУ ВІДНОШЕНЬ МІЖ ОБ'ЄКТАМИ

У попередніх розділах було визначено ідею нового способу опису відношень між об'єктами. А будь-який механізм, що включається у перелік можливостей мови програмування, потребує текстового запису у вигляді синтаксичних конструкцій мови. Тому наступний розділ статті присвячено вирішенню питання побудови синтаксу для реалізації запропонованого способу опису міжоб'єктних відношень, що також дозволить глибше зрозуміти деталі даного способу.

При описі синтаксису необхідно спиратись на логіку, на основі якої виконується читання коду, адже читання коду – це дія, що займає найбільший час при розробці ПЗ. При реалізації частини коду, для якої критична інкапсуляція методів, більшу частину часу виконується читання інтерфейсу фокального класу (у наслідок того, що саме інтерфейси описують методи як члени класів) і декларацій змінних типу фокального класу (через змінні виконується взаємодія з об'єктом). Отже, саме у цих місцях коду необхідно описати нові синтаксичні конструкції.

3.1. Декларація метаданих класу

Опис особливостей ролі будемо виконувати за допомогою синтаксису опису метаданих класу. Декларацію атрибутів пропонується виконувати наступним чином:

"{"<attribute-type> "}" (<attribute-identifier>) ":" ("[")<arguments-list>("]")";", де

- *attribute-type* (тип атрибуту). На основі поданого у даній статті матеріалу можна описати два типи атрибутів: *access* та *owning*. Вони описують, відповідно, структурні відношення і відношень доступності;

- *attribute-identifier* (ідентифікатор атрибуту) - ім'я атрибуту. Може бути відсутній - атрибут при цьому буде називатись *анонімним*. Анонімні атрибути зручно використовувати при описі аргументів інших атрибутів для визначення атрибутів-аргументів у точці опису атрибуту (див. приклад нижче);

- *arguments-list* (список аргументів). Перелік через кому аргументів, що ставляться у відповідність атрибуту. Так, наприклад, для атрибуту типу *access* це буде перелік методів, до яких буде відкритий доступ через роль. Також у якості аргументу може виступати вираз з атрибутів типу *attribute-type*, записаних за правилами арифметики множин (див. рис. 2).

Примітка №1: запис у лапках означає належність рядка словам мови.

Примітка №2: запис у дужках означає необов'язковість використання слів, що записуються у дужках.

Таким чином, опис класу з метаданими груп доступу, що зображені на рис. 1, та одним повним описом ролі *test_role* буде виглядати наступним чином:

```
// Class declaration
//
class A
{
    method_1();
    method_2();
    method_3();

    // Declaration of metadata:
    //
    {access} role_1: method_1, method_2;
    {access} role_2: method_1, method_3;
    {access} role_3: role_1, role_2;
    {role} test_role: [{access}: role_3, {owning}: has_a];
```

// For role declaration using anonymous attributes, that declaring on point

// (There are next anonimus attributes here: access, that includes access of role 3

// and owning, that explained as has_a owning type).

}

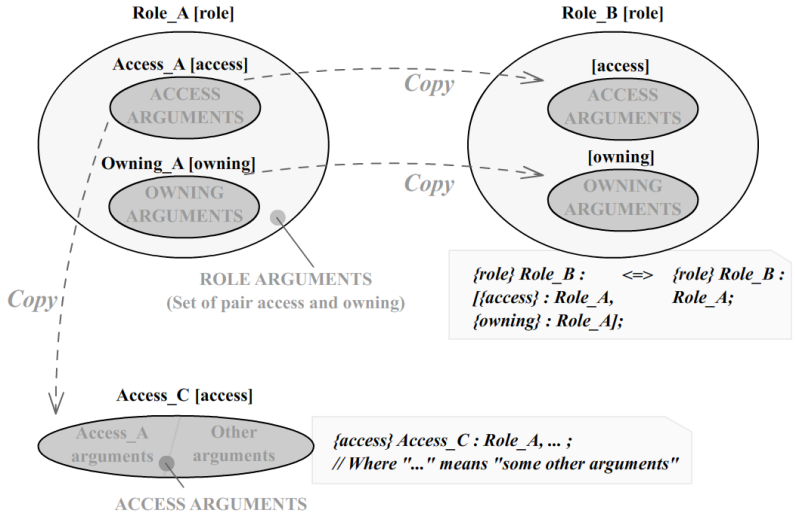

Рисунок 2 - Діаграма, що ілюструє передачу аргументів між атрибутами

Також пропонується визначити наступні атрибути за замовчуванням для підтримки звичних для сучасних мов програмування понять:

- стандартні атрибути доступу: *private*, *protected*, *public*.

При їх описі може виконуватись неявне розширення іншими атрибутами, тобто:

{access} public: /* Public methods names */;

{access} protected: (*public*), /* Protected methods names */;

{access} private: (*protected*), /* Private methods names */;

Примітка: тут запис у круглих дужках означає те, що дане слово писати необов'язково, воно за замовчуванням мається на увазі у визначеному місці;

- стандартні ролі:

{role} this:[*{access}: private*, *{owning}:* knows_a]

{role} child-default:[*{access}: protected*, *{owning}:* has_a].

{role} public-default:[*{access}: public*, *{owning}:* knows_a].

3.2. Декларація посилань

Роль посилань пропонується декларувати за синтаксисом шаблонів у C++ (у трикутних дужках). Декларація посилання буде виглядати наступним чином:

<reference-class>"<"<attribute-identifier>">" <reference-identifier>

Іншими словами, можливо у точці опису посилання визначити роль "з нуля" або передати існуючу роль у якості атрибуту ролі, що описується (тут під attribute-identifier розуміється ідентифікатор існуючої ролі).

```
// Pointer declaration
//
// Getting class metadata from role:
foo(A<[{access}: test_role, {owning}: func_owning]> pointer)
{
    pointer.method_1(); // Calling this method is OK
    pointer.method_2(); // Calling this method is OK
    pointer.method_3(); // Semantic error - access denied
}

// Role extending
foo(A<[{access}: role_1, method_3, {owning}: func_owning]> fullPointer)
{
    fullPointer.method_3();
```

```
    // Call of method_3() is OK here, because it performed
    // for new role with wider access.
}
```

4. ОЦІНЮВАННЯ ЯКОСТІ РОЗРОБЛЕНОГО СПОСОБУ

Аналіз якості розробленого способу будемо виконувати на основі його реалізації, синтаксис якої був описаний у розділі 3 за критеріями, встановленими у пункті 2.

4.1. Аналіз за критерієм лаконічності опису базової функціональності

Для аналізу опису структури класу та декларації посилань, що виконують опис відношень доступності за замовчуванням (без зміни груп доступу за замовчуванням), виконаємо опис класу, що включає у себе три методи з різними рівнями доступу: private, protected і public. Опис будемо виконувати у рамках синтаксису C++ (зліва) та модифікованого C++ (справа), синтаксис якого був описаний у розділі 3.

Таблиця 1 –

Порівняння C++ та модифікованого C++ за лаконічністю опису базової функціональності

C++	Модифікований C++
Декларація класу	
Class A { // state declaration private: void fooA(); protected: void fooB(); public: void fooC(); }	Class A { // state declaration void fooA(); void fooB(); void fooC(); *{access} public:* fooA; *{access} protected:* fooB; *{access} private:* fooC; }
Декларація посилань	
… p<A> thePointer; *// "p" is name of "has_a" auto-reference* …	… A<*{owning}:* has_a> thePointer; *// class explain reference access by default* …

У вказаному прикладі обсяг коду по рядкам залишається однаковим, але, разом з тим, можна відмітити декілька недоліків:

1. Запис імен методів у один рядок може ускладнити читання коду.

2. При застосуванні статичного поліморфізму для методів класу [1] ім'я методу перестає виконувати його вичерпний опис, і опис ролі, група доступу якої визначається лише іменем методу, може стати неоднозначним.

Вказаних недоліків планується позбутись при дослідженні реалізацій розширеного способу опису відношень між об'єктами у ОО парадигмі.

4.2. Аналіз за лаконічністю модифікації функціональності

У класичному C++ розширення неможливе без зміни коду класу. Тут вважається, що усі методи класу були описані як *public*.

Таблиця 2 -

Порівняння C++ та модифікованого C++ за лаконічністю модифікації функціональності

C++	Модифікований C++
Зміна інкапсуляції класу	
Class Role_1proxy { private: // Here we have implementation of owning p<A> _proxyField; public: void fooA() { _proxyField.fooA(); } void fooC() { _proxyField.fooC(); } }	У модифікованій C++ створення *proxy* не потрібно.
Декларація посилань	
Role_1proxy thePointer;	A<*{access}: public*, fooC, *{owning}:* has_a> thePointer;

Очевидно, що обсяг коду став набагато меншим при використанні запропонованого у статті способу опису відношень між об'єктами.

ВИСНОВКИ

Після проведення аналізу способів побудови відношень між об'єктами в існуючих ОО мовах програмування з урахуванням його результатів запропоновано розширений спосіб опису міжоб’єктних відношень і, як складову частину способу, розроблено спосіб реалізації інкапсуляції в ОО парадигмі. На основі сформульованих критеріїв оцінювання якості реалізації опису відношень порівняно запис коду на C++ із записом на модифікованому C++, і, як результат, доведено ефективність способу за двома з трьох встановлених критеріїв. Для перевірки за критерієм дескриптивності потрібно виконати соціологічне дослідження сприйняття програмістами з деякої контрольної групи реалізації розширеного способу опису відношень між об'єктами.

Таким чином, у статті виконані усі поставлені задачі і досягнута мета - забезпечення можливості динамічного змінювання міжоб’єктних відношень без застосування проміжних програмних сутностей.

У подальшому планується виконати програмну реалізацію синтаксичного доповнення до мови програмування C++, що додає описані у статті елементи синтаксису. Це дозволить на практиці перевірити переваги і недоліки запропонованого способу.

ЛІТЕРАТУРА:

1. Бьерн Страуструп. Язык программирования C++: пер. з англ. /Бьерн Страуструп ; [пер. Николай Мартынов]. – изд. 2-ге, допов. – М.: Бином, 2008. – 1104 с. – ISBN 5-7989-0226-2. – ISBN 5-7940-0064-3. – ISBN 0-201-70073-5 (серія).
2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns : англ. /Erich Gamma; NY.: Addison Wesley, 1994. – 416 с. – ISBN 0201633612 (серія).
3. James Coplien, Gertrud Bjørnvig. Lean Architecture for Agile Software Development : англ. /James Coplien; W.: John Wiley & Sons, 2010. – 376 с. – ISBN 0932633579 (серія).
4. Kasper Bilsted Graversen. The nature of roles :PhD thesis : /Kasper Bilsted Graversen. – Copenhagen, IT University of Copenhagen Copenhagen, September 2006. – 278 p. – Bibliography :p. 260-275.
5. Attributes Tutorial [Електронний ресурс] : Microsoft Developer Network. – 2013. – Режим доступу до статті : http://msdn.microsoft.com/en-us/library/aa288454(v=vs.71).aspx
6. Annotations [Електронний ресурс] : The Java Tutorials. – 2013. – Режим доступу до статті: http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html
7. Wilson R., Paul R.. Uniprocessor Garbage Collection Techniques : англ. / Wilson R. ; London, UK: Springer-Verlag, 2009. – 376 с. – ISBN 3-540-55940-4 (серія).
8. Boost shared pointers [Електронний ресурс] : From freeform area for Inkscape development and discussion. – 2013. – Режим доступу до статті : http://wiki.inkscape.org/wiki/index.php/Boost_shared_pointers
9. Objective-C [Електронний ресурс] : From Wikipedia, the free encyclopedia. – 2013. – Режим доступу до статті: http://en.wikipedia.org/wiki/Objective-C#Automatic_Reference_Counting