УДК 004.4'242

Федорченко В. М.

КАРКАС ДЛЯ ПІДТРИМКИ МОДЕЛЬ-ОРІЄНТОВАНОЇ РОЗРОБКИ НА ОСНОВІ СПРОЩЕНОЇ ІНФРАСТРУКТУРИ ТРАНСФОРМАЦІЇ XML-МОДЕЛЕЙ

Ефективне застосування модель-орієнтованої методології для розробки програмних проектів потребує особливого середовища. У роботі розглядаються основні аспекти побудови такого середовища, яке базується на спрощеній інфраструктурі трансформації XML-моделей до конфігурації IoC-контейнера. Основна увага присвячена створенню особливого каркаса, який вирішує проблеми композиції та взаємодії об'єктів, що виникають при автоматичній трансформації моделей.

Ефективне повторне використання будь-якого формалізованого знання між різними проектами можливе лише за умови, що ці проекти мають деяку спільну онтологічну основу [1]. Ця основа є гарантом того, що знання, представлене в певній формі, не втратить своєї актуальності в іншому проекті, де планується повторне використання. У контексті модель-орієнтованої розробки [2; 3] це означає, що для організації повторного використання деякої моделі з одного проекту має існувати трансформація до будь-якої інтерпретованої моделі в іншому проекті (у найкращому випадку трансформація взагалі непотрібна). Визначення трансформацій типу модель - модель також є предметом для повторного використання, тому бажано, щоб спільний онтологічний рівень проектів існував на якомога вищому рівні абстракції у ланцюжку перетворень модель - модель.

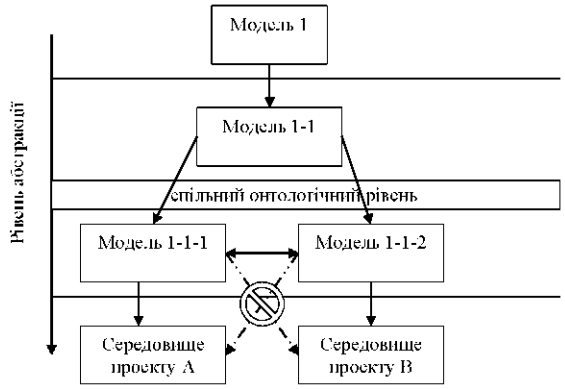

Рис. 1. Повторне використання моделей та трансформацій відповідно до спільного онтологічного рівня (моделі, що визначені на рівні 1-1-1 та 1-1-2, не можуть бути повторно використані в іншому проекті без додаткової горизонтальної трансформації)

Спрощена інфраструктура для трансформації XML-моделей ґрунтується на використанні ідей компонент-орієнтованої розробки (CBD [4]) та сервіс-орієнтованої архітектури (SOA [5]). Будь-яка ОО-система під час виконання може бути уявлена як надзвичайно великий граф взаємозалежних об'єктів, що змінюється з часом; отже, має існувати деякий початковий стан такого графа. Використання патерну інверсія керування (inversion of control [6]) для опису такого початкового стану графа об'єктів (сервісів) системи дозволяє забезпечити основні властивості компонентів: можливість багаторазового використання, незалежність від специфічного контексту, здатність до композиції (через сумісні інтерфейси) тощо [7].

Головна ідея спрощеної інфраструктури для трансформації XML-моделей полягає у використанні XML-конфігурації IoC-контейнера [8] як кінцевої моделі у ланцюжку перетворень. Фактично така модель є формалізацією тривіальної концептуальної мапи (concept map [1]), де концептам відповідають екземпляри класів, а ребра навантажені єдиним типом відношення - «залежить від» («використовує»). Очевидно, що будь-яку концептуальну мапу можна представити у вигляді такої конфігурації IoC-контейнера, оскільки завжди існує таке перетворення графа (що відповідає концептуальній мапі), при якому зберігається його оригінальний семантичний зміст. Для цього достатньо замінити типізовані ребра на конструкцію з пари ребер типу «залежить», що з'єднують проміжну вершину-концепт (яка буде репрезентувати тип відношення):

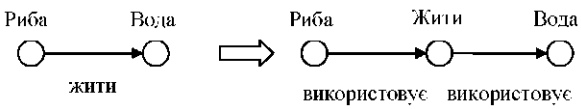

Рис. 2. Приклад перетворення довільної концептуальної мапи до вигляду, що відповідає конфігурації IoC-контейнера

Ефективна реалізація концептів у вигляді програмних об'єктів або їх композиції є важливою задачею, що постає при використанні такого підходу до модель-орієнтованої розробки. У пропонованій роботі розглядається підхід до вирішення цієї задачі через створення особливого каркаса (framework), який підтримує визначення складних концептів шляхом композиції вже існуючих реалізацій концептів.

Отже, каркас має репрезентувати певний онтологічний базис, на основі якого ефективно вирішується задача композиції та взаємодії концептів. На сьогодні не існує загальноприйнятого стандарту для онтології вищого рівня (upper ontology), оскільки кожен з існуючих претендентів (SUMO [9], GFO [10]) має свої особливості та не є універсальною онтологією для будь-якого домену знань [1]. Специфічні базові онтології, такі які CEO (Core Enterprise Ontology [11]), є більш зручними для моделювання у деяких доменах, але мають обмежену область застосування. За відсутності загальної онтології вищого рівня єдиним базовим поняттям для каркаса буде поняття деякого об'єкта, що існує в об'єктно-орієнтованому середовищі інформаційної системи.

Для вирішення задачі композиції та взаємодії довільних об'єктів пропонуються такі категорії (які визначаються через загальні поняття, що існують у програмуванні взагалі):

- сутність: репрезентує пасивний об'єкт, що існує на деякому проміжку часу та має деякі характеристики (стан);
- процес: об'єкт, що репрезентує деяку мету у вигляді маніпуляцій із сутностями (стану)
  - операція: процес, метою якого є деякий новий стан деякої підмножини сутностей, що існували на момент виклику;
  - постачальник: процес, метою якого є «доставка» у місце виклику деякої сутності; характерною рисою цього процесу є незмінність множини сутностей, що існували на момент виклику.

Треба зазначити, що ці категорії не мають жодного відношення до реальної сутності концептів, а лише визначають ролі об'єктів, що беруть участь у тій чи іншій взаємодії: наприклад, об'єкт, що репрезентує концепт «кімната», може бути як сутністю (наприклад, для операції «переміщення предмета»), так і постачальником (наприклад, для доставки сутності «стілець»).

Розглянемо основні аспекти реалізації каркаса на основі таких категорій для платформи Microsoft.NET. У рамках цієї платформи сутностями можуть бути будь-які об'єкти-контейнери даних: такі об'єкти передусім інкапсулюють деяку структуру даних та визначають зручні методи доступу. До стандартних (для .NET Framework) об'єктів-сутностей належать реалізації

інтерфейсів колекцій (IList, IDictionary тощо), спеціалізовані класи-контейнери даних (для реляційних даних - DataSet, для XML-структури - XmlDocument тощо). Каркас не вимагає особливого інтерфейсу для доступу до таких об'єктів-сутностей, оскільки існуючого в .NET механізму відображень (reflection) цілком достатньо для доступу до стану сутностей.

Об'єкти-процеси становлять динамічну складову моделі. Ці об'єкти інкапсулюють правила зміни стану системи взагалі, отже, основним очікуваним результатом роботи об'єктів-операцій є зміна стану сутностей, відповідно, об'єкти-постачальники реалізують логіку представлення та вибірки даних з об'єктів-сутностей до контекстів операцій. Об'єкти, що будуть відповідати поняттям «операція» та «постачальник», мають реалізовувати відповідні уніфіковані інтерфейси. Це необхідно для того, щоб ці об'єкти мали необхідну здатність до композиції на рівні конфігурації IoC-контейнера, а також для подальшого визначення типових варіантів їх композицій (C#):

```
public interface IOperation {
 void Execute(object context);
}
public interface IOperation<ContextT> {
 void Execute(ContextT context);
}
public interface IProvider {
 object Get(object context);
}
public interface IProvider<ContextT, ResultT> {
 ResultT Get(ContextT context);
}
```

Розглянемо варіанти композиції, що доступні в рамках цих інтерфейсів. Для об'єктів-постачальників існують такі типи принципово різних варіантів композиції:

- з'єднання типу «ланцюг»: складний постачальник, який послідовно викликає інші постачальники, при цьому результат викликів попередніх постачальників є контекстом для наступного;
- з'єднання типу «пошук»: складний постачальник, що послідовно викликає інші постачальники, поки не отримає результат, який відповідає деяким умовам;
- з'єднання типу «об'єднання»: складний постачальник, що викликає деяку множину інших постачальників та об'єднує їх результати.

Для об'єктів-операцій доступний лише один варіант композиції, а саме складна операція типу «ланцюг», що складається з послідовності викликів інших операцій або постачальників, при цьому контекст викликів може залежати від усіх попередніх викликів у ланцюжку.

Описаних вище типів композиції достатньо для опису фактично будь-яких складних процесів; єдиною перешкодою у разі, коли опис таких композицій буде генеруватися автоматично, є гармонізація типів у викликах (платформа .NET передбачає строгу типізацію). Для вирішення цієї проблеми необхідно додати до каркаса поняття «перетворювач типів» (C#):

```
public interface ITypeConverter {
 bool CanConvert(Type fromType, Type toType);
 object Convert(object o, Type toType);
}
```

Безумовно, «перетворювачі типів» є об'єктами-постачальниками, і до них можуть бути застосовані всі типові для дій-постачальників варіанти композиції для визначення складних перетворювачів типів.

За своєю суттю каркас є інтегруючим та «ненав'язливим» (non-intrusive) (не вимагає залежності усієї системи від інтерфейсів каркаса); отже, має надавати можливість використовувати у ролі об'єктів-операцій та об'єктів-постачальників навіть ті об'єкти, що не реалізують відповідних інтерфейсів. Для цього каркас містить класи-адаптери (спеціальні класи, що «обгортають» інший довільний об'єкт), які працюють зі стороннім об'єктом за допомогою інтерфейсу відображень (reflection): виклик методу (InvokeMethod), виклик властивості (GetProperty), отримання значення поля (GetField) тощо.

Таким чином, базовий каркас для підтримки модель-орієнтованої розробки на основі спрощеної інфраструктури складається з трьох основних підсистем:

- Підсистема постачальників: реалізації композиції для основних типів, адаптери.
- Підсистема операцій: універсальна реалізація композиції типу «ланцюг», адаптери.
- Підсистема конвертації типів: реалізації конверторів для типізованих постачальників та операцій, а також для типізованих інтерфейсів колекцій (IDictionary<>, IList<> тощо).

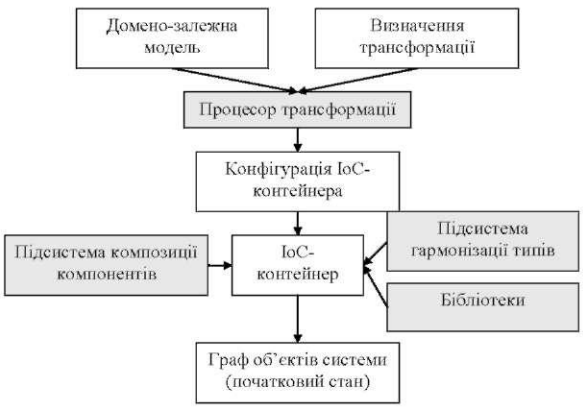

Рис. 3. Загальна схема перетворення моделі та місця впливу каркаса

Для ілюстрації розглянемо визначення абстрактної операції типу «ланцюг» та її трансформації до моделі конфігурації IoC-контейнера. Нехай зазначена операція буде складатися з композиції таких класів (C#, фрагменти):

```
// операція-ланцюг
public class Chain : IOperation<IDictionary<
string, object» {
 public IOperation<IDictionary<string,
    object>>[] Operations { get; set; }
 public void Execute (IDictionary<string,
    object> context) {
 for (int i=0; i<Operations.Length; i++)
  Operations[i].Execute (context);
}
}
// одина з можливих ланок ланцюга
public class ChainOperationCall:
OperationCall, IOperation<IDictionary<string,
    object>> {
 public IProvider<IDictionary<string, object>,
    bool> RunCondition { get; set; }
public IProvider ContextProvider { get; set; }
public IOperation Operation { get; set; }
 public void Execute(IDictionary<string, object>
context) {
 if (RunCondition!=null)
  if (!RunCondition.Provide(context)) return;
  Operation.Execute( ContextProvider.
    Provide(context) );
 }
}
```

Клас Chain агрегує один (чи більше) об'єкт з інтерфейсом, що сумісний з інтерфейсом самого «ланцюга», і просто викликає їх по черзі. Відповідно вся додаткова логіка має бути реалізована у класах-обгортках, що будуть використовуватись як ланки ланцюга: клас ChainOperationCall дозволяє визначити умову виконання ланки та правило формування контексту для операцій, що інкапсулюються у ланці. Ця реалізація композиції типу «ланцюг» цілком достатня для опису простих послідовних процесів, наприклад дій типу «завантажити запис, змінити поле, зберегти запис» (XML, модель):

```
<chain name="doSetProductTitle">
 <prv-call target="productLoader"
    result="product">
 <context><const-prv
    value="someProductId"/></context>
</prv-call>
<op-call target="setTitle">
 <context><entry key="product"/></context>
</op-call>
<op-call target="productSave">
 <context><entry key="product"/></context>
</op-call>
</chain>
```

Трансформація до конфігурації IoC-контейнера (Winter4NET [12]) для визначення ланцюга на основі вищенаведених класів може мати такий вигляд (XSL):

```
<xsl:template match='chain'>
 <component name='{@name}' type='NReco.
Operations.Chain,NReco'>
 <property name='Operations'>
 <list>
 <xsl:for-each select='*'>
 <entry>
 <xsl:apply-templates select='.' mode='chain-
operation'/>
 </entry>
 </xsl:for-each>
 </list>
 </property>
 </component>
</xsl:template>
<xsl:template match='op-call' mode='chain-
    operation'>
 <component type='NReco.Operations.Chain
    OperationCall,NReco' singleton='false'>
 <property name='Operation'>
 <xsl:choose>
  <xsl:when test='@target'><ref name='{@
    target}'/></xsl:when>
  <xsl:otherwise><xsl:apply-templates
    select='target/*'/></xsl:otherwise>
 </xsl:choose>
 </property>
 <xsl:iftest='count(context/*)>0'>
 <property name='ContextProvider'>
 <xsl:apply-templates select='context/*'/>
 </property>
</xsl:if>
<xsl:iftest='count(condition/*)>0'>
 <property name='RunCondition'>
 <xsl:apply-templates select='condition/*'/>
 </property>
 </xsl:if> "
 </component>
</xsl:template>
```

Останній крок - включення моделі через інструкцію перетворення до конфігурації IoC-контейнера:

```
<components>
 <xsl-transform>
  <xml file="model.xml"/>
  <xsl file="transform.xsl"/>
 </xsl-transform>
</components>
```

Слід зазначити, що можливість використання рекурсивних трансформацій (коли для перетворення деякого доменного поняття визначається трансформація не одразу до конфігурації IoC-контейнера, а до понять, для яких уже існує така

трансформація) дозволяє забезпечити високий рівень повторного використання саме правил трансформації. Ітеративний процес визначення доменних понять різного рівня абстракції може тривати до моменту, коли можна буде визначити модель програмної системи (або її частини) у формі, яка є оптимальною для вирішення деякої задачі саме в конкретних умовах.

На основі підходу, описаного в цій роботі, було розроблено каркас для підтримки комерційної модель-орієнтованої розробки веб-застосувань та впроваджено у фірмі «NewtonIdeas» (http://www.newtonideas.com). Завдяки відсутності використання будь-якої онтології вищого рівня та зосередженню саме на проблемі композиції та взаємодії абстрактних об'єктів каркас було успішно застосовано для розробки понад 100 різноманітних програмних проектів різного рівня складності. У стані розробки знаходиться також полегшена версія такого каркаса з відкритим програмним кодом (open source) під назвою NReco (http://code.google.com/p/nreco/).

Подальший розвиток каркаса може бути зосереджений на визначенні більш складних композицій, характерних для паралельних обчислень (у цій роботі основна увага була присвячена послідовним процесам).

1. *Gasevic D., Djuric D., Devedzic V.* Model Driven Architecture and Ontology Development. Springer, 2006. ISBN 3-54032-180-2.
2. *Beydeda S., BookM., Gruhn V.* Model-Driven Software Development. Springer, 2005. ISBN 3-54025-613-X.
3. *Czarnecki K.* Model-Driven Software Development: Technology, Engineering, Management. Wiley, 2006. ISBN 0-47002-570-0.
4. *Brown A. W.* Large-Scale, Component-Based Development. Prentice Hall PTR 2000. ISBN 0-13-088720-X.
5. *Apperly H., Hofman R. and others.* Service- and Component-based Development: Using Select Perspective™ and UML. Addison Wesley, 2003. ISBN 0-321-15985-3.
6. *Fowler M.* Inversion of Control Containers and the Dependency Injection Pattern. 2004, online: http://martinfowler.com/articles/injection.html
7. *Szyperski C., Gruntz D., Murer S.* Component Software: Beyond Object-Oriented Programming (2nd edition). Addison-Wesley Professional, 2002. ISBN 0-20117-888-5.
8. *Walls C. and Breidenbach R.* Spring in Action, Second Edition. Manning, 2007. ISBN 1-93398-813-4.
9. *Niles I., Pease A.* Towards a Standard Upper Ontology / In Proceedings of the 2nd International Conference on Formal Ontology in Information Systems, Volume 2001. ACM Press 2001. ISBN 1-58113-377-4.
10. *Heller B., Herre H.* Ontological categories in GOL / In Process Theories: Crossdisciplinary Studies in Dynamic Categories. Springer, 2004. ISBN: 1-40201-751-0.
11. *Bertolazzi P., Krusich C., Missikoff M.* An Approach to the Definition of a Core Enterprise Ontology: CEO. OES-SEO 2001, Int. Workshop on Open Enterprise Solutions: Systems, Experiences, and Organizations, Rome, September 14-15, 2001.
12. Lightweight .NET IoC container. Winter4Net, online: http://www.winter4.net/

V. Fedorchenko

FRAMEWORK FOR MODEL-DRIVEN DEVELOPMENT BASED ON THE LIGHTWEIGHT XML-MODELS TRANSFORMATION INFRASTRUCTURE

Effective usage of model-driven methodology for software project development requires special environment. This paper examines the basic aspects of building such an environment, which is based on the lightweight infrastructure that performs transformations from XML-models to IoC-container configuration. When automatic model transformations are used, object composition and interaction problems occur. Main attention of this work is devoted to the creation of the special framework that solves these problems.