



HumanStore – maszyna wirtualna realizująca założenia Model Driven Architecture

Mgr inż. Andrzej Krawczyk

Podstawowym założeniem Model Driven Architecture jest separacja logiki biznesowej aplikacji od platformy ją realizującej. Według niezależnego konsorcjum Object Management Group logika i struktura aplikacji powinna być opisana w języku UML jako model niezależny od platformy (PIM). Realizację tak wyspecyfikowanej logiki mają zapewnić transformacje na kod związany z konkretną platformą (PSM). Dzięki takiemu podejściu konkretne rozwiązania mogą być całkowicie niezależne od rozwijających się technologii – ten sam model może być realizowany zarówno w obecnych technologiach (CORBA, Enterprise JavaBeans, XML/SOAP, COM+, .NET), jak również w tych, które dopiero powstaną.

Maszyna wirtualna HumanStore transformuje model aplikacji opisany w UML na gotową do użycia aplikację, bez konieczności korzystania z żadnych dodatkowych narzędzi. Dzięki określeniu ścisłych wymagań dotyczących usług, jakie mają być zaimplementowane na platformie docelowej, HumanStore spełnia założenia MDA oraz wskazuje drogę do transformacji niezależnych modeli na dowolne platformy.

1. WSTĘP

Wśród procesów tworzenia dzieł technologicznych, proces tworzenia oprogramowania jest wyjątkowy. Pomimo ograniczeń w postaci metodologii i języka kodowania, nie ma logicznej konieczności przyjęcia pewnego określonego sposobu rozwiązania powstałych problemów i dopuszczalne jest wiele rozwiązań równorzędnych. Dlatego tworzenie oprogramowania często przypomina tworzenie szeregu indywidualnych „utworów informatycznych”. W przypadku małych systemów można to porównać do sytuacji, w której kilku twórców wspólnie komponuje utwór, który ma być dojrzały, sprawny i funkcjonalny. Niestety, gdy rośnie ilość zagadnień, które ma obejmować system, indywidualizm i arbitralność przyjmowanych rozwiązań zaczyna być jednym z najbardziej krytycznych problemów i generatorem najpoważniejszych kosztów.

Patrząc z tej perspektywy na historię rozwoju inżynierii oprogramowania widać, że problem ten identyfikowano już od początku, dążąc do sformułowania metodologii, która w jakiś sposób usystematyzowałaby cykl życia tworzonego systemu.

Przykładem sprawdzonego rozwiązania systematyzującego metodologię tworzenia aplikacji docelowych jest HumanStore – specjalizowana maszyna wirtualna z elementami narzędzi CASE. Jej pierwsza wersja powstała w 1999 roku na bazie analizy dotychczas stosowanych metodologii konstrukcji oprogramowania oraz doświadczeń zespołu programistów, który wciąż pracuje nad jej rozwojem.

Poniżej przedstawiono metodologie, z których czerpie HumanStore oraz główne założenia i korzyści, jakie daje jego zastosowanie.

2. ŹRÓDŁA HumanStore

2.1. PROGRAMOWANIE OBIEKTOWE

Metodologia obiektowa była ostatnim wielkim osiągnięciem inżynierii oprogramowania i jest powszechnie dziś stosowana. Znakomicie się sprawdza przy wytwarzaniu wszelkiego rodzaju gotowych komponentów, z których mogą korzystać informatycy przy tworzeniu konkretnych aplikacji.

Niestety dla realizacji idei automatyzacji i porządkowania procesu wytwarzania oprogramowania jest to krok niewystarczający. Oto przykłady wciąż nierozwiązanych problemów:

- Komponenty (klasy, typy) mają różne interfejsy; można korzystać z komponentów na wiele różnych sposobów.
- Indywidualnie tworzone projekty mają często zaawansowaną logikę biznesową; samo programowanie obiektowe daje niewielkie wsparcie w tej dziedzinie.
- Kod, choćby najlepiej napisany, nie jest dobrą formą opisu przyjętego rozwiązania i nie nadaje się jako materiał do komunikacji pomiędzy klientem, projektantem a programistą.

2.2. WZORCE PROJEKTOWE

Gotowe wzorce projektowe są dużym wsparciem dla programistów, ponieważ znakomicie usprawniają proces budowy złożonego systemu. Ogólnie rzecz biorąc, określają one, jak i kiedy należy użyć określonego komponentu oraz jakich użyć konstrukcji. Przyjęcie określonych wzorców daje w dużym uproszczeniu przepis na to, jak budować konkretną aplikację.

Wzorce projektowe, choć rozwiązują problemy konstrukcyjne szkieletu aplikacji lub problemy konstrukcyjne wielu elementów systemu, mają szereg poważnych wad, między innymi:

- Bardzo słabo wspierają logikę biznesową aplikacji.
- Logika zachowania konkretnej aplikacji, czy też nowo tworzonego elementu systemu jest

umiejscowiona we fragmentach przewidzianych przez wzorzec projektowy i przez to trudna do zlokalizowania.

– Kod realizujący logikę działania danego fragmentu systemu jest ściśle związany zarówno z charakterystyką wzorca jak i interfejsami użytych komponentów.

2.3. UML

Unified Modelling Language to bogaty zestaw narzędzi, używanych przede wszystkim do wizualizacji projektu informatycznego na różnych etapach jego tworzenia. Diagramy UML umożliwiają jasne i precyzyjne przedstawienie projektu systemu informatycznego w różnych aspektach. Na przykład, diagramy użycia jednoznacznie ustalają zastosowania realizowanego systemu. Z kolei diagramy przepływu sterowania pokazują interakcje pomiędzy komponentami, a diagramy klas definiują statyczną strukturę i relacje pomiędzy obiektami tych klas.

Pomiędzy opisem w UML a konkretną implementacją systemu istnieje jednak duża przestrzeń trudna do zdefiniowania i zautomatyzowania, co jest źródłem, między innymi, następujących problemów:

– Model UML określa, że typ A ma asocjację do typu B o liczności 1..N; nie wiadomo jednak, jak taka relacja jest zaimplementowana w konkretnym systemie.

– Diagram przepływu sterowania określa przejścia z jednej operacji do drugiej, ale nie definiuje sposobu, w jaki to się odbywa.

Z powodu tego typu trudności, UML jest postrzegany przez większość projektantów i programistów wyłącznie jako sposób na specyfikację i dokumentowanie tworzonego oprogramowania.

2.4. MDA

Model Driven Architecture dąży do automatyzacji generowania jak największej ilości kodu na podstawie modelu opisanego za pomocą UML lub wybranego profilu. Projekt ten zakłada stworzenie takiego zestawu transformacji, by na podstawie modelu, określanego jako PIM (Platform Independent Model) wraz z CIM (Computation Independent Model) można było generować PSM (Platform Specific Model), który następnie może być kompilowany do modelu implementacyjnego.



3. HumanStore – NAJWAŻNIEJSZE ZAŁOŻENIA

HumanStore jest maszyną wirtualną, która narzuca pewne ściśle określone wzorce projektowe poprzez swoją architekturę, zestaw rozkazów i cechy działania. Głównym jej założeniem jest całkowita izolacja poszczególnych warstw aplikacji. Zbiór instrukcji tej maszyny nie tylko nie zawiera rozkazów dostępu do systemu, ale również nie zawiera deklaracji wskazujących sposób implementacji struktur danych i przepływy sterowania tworzonej w nim aplikacji.

Podstawowe różnice między HumanStore a maszynami typu Java lub .NET to:

- Brak programowego dostępu do elementów interfejsu użytkownika. Definicja interfejsu użytkownika przypomina definicję raportu, który się rysuje i definiuje, co ma się na nim pokazać. Nawet w przypadku, gdy jest to bardzo złożony interfejs zawierający np. grafikę wektorową definiuje się go całkowicie statycznie.
- Brak programowej możliwości określania sposobu przechowywania danych. Można jedynie określić, czy pewna klasa zawiera dane lokalne użytkownika, czy też dane współdzielone.
- Brak możliwości programowego definiowania szczegółów implementacji kontenerów danych. Osoba definiująca model biznesowy aplikacji ma wyłącznie jeden kontener danych: kolekcję.

3.1. MODELOWANIE LOGICZNEJ STRUKTURY DANYCH

Źródłem definicji struktur danych w HumanStore są diagramy klas UML. Kompilator przyjmuje abstrakcyjną definicję, w której określa się jedynie relacje i licznosci. Warstwa modelu konkretnego rozwiązania nie pozwala na definiowanie, w jaki sposób kompilator zrealizuje dane zlecenie. Można określać dodatkowe atrybuty behawioralne, takie jak uporządkowanie (sekwencja), a w przyszłości również przewidywaną licznosc, najczęstszy sposób dostępu (np. poprzez klucz) czy komunikację z innymi systemami.

Dzięki takiemu podejściu model danych można realizować na różne sposoby – od operacji na obiektach przechowywanych w pamięci operacyjnej, poprzez przechowywanie obiektów w bazach danych (MySQL, MSSQL, Oracle), aż po wymianę pomiędzy serwisami. To parametry kompilacji i dostępność kontenerów danych określają, gdzie i w jaki sposób dane są przechowywane, skąd pochodzą i jakie są miejsca docelowe ich transmisji. W uproszczeniu można powiedzieć, że sposób implementacji zależy od transformacji użytej przy ostatecznym generowaniu aplikacji.

3.2. MODELOWANIE ZACHOWANIA SYSTEMU

Przyjęty model danych – definiowany na wysokim poziomie abstrakcji, wyłącznie w postaci

logicznych relacji – pozwala na definiowanie dynamicznego zachowania systemu na równie abstrakcyjnym poziomie. Można przyjąć dowolny język, którego kompilator będzie generować kod wykonywalny w maszynie HumanStore. Niestety większość konstrukcji dostępnych w takich językach jak C++ na przyjętym poziomie abstrakcji jest bezużyteczna. Dlatego jako podstawowy język opisujący zachowanie HumanStore (a raczej stan wybranych obiektów i kolekcji po zajściu jakiegoś zdarzenia) wybrano imperative OCL (Object Constraint Language).

Możliwości modelowania zachowania systemu w języku OCL ograniczone zostały wyłącznie do definiowania wartości, jakie mają przyjąć atrybuty obiektów oraz do tworzenia i usuwania obiektów.

Możemy napisać:

```
context Person::CreateTask(ToDo: string, DueDate: date)
post:
self.MyTasks.newElement(element.ToDo = ToDo and element.DueDate =
DueDate)
```

Nie możemy jednak w tej warstwie modelu określić, jak nowe zadanie będzie wyświetlane, ani gdzie zostanie zapisane (lub wysłane).

Diagramy klas oraz opis zachowania w imperative OCL dają kompletny opis modelu biznesowego dowolnej aplikacji. Realizacja jej wykonania zależy od konkretnego systemu.

3.3. DEFINIOWANIE INTERFEJSÓW UŻYTKOWNIKA

Interfejs jest modelowany jako statyczna warstwa prezentacji danych. Każdy komponent interfejsu użytkownika musi mieć zadeklarowane, jakie dane prezentuje. Charakter komponentu decyduje, jaką postać przybierze dana prezentacja. Każdy komponent, w zależności od swojej złożoności, pozwala na zadeklarowanie operacji, jaka ma być wykonana w wyniku interakcji użytkownika. Najprostszym komponentem jest pole tekstowe, dla którego definiujemy, co ma być w nim wyświetlone.

Dotychczas HumanStore był profilowany jako maszyna pracująca na rzecz aplikacji biznesowych. W związku z tym posiada zaimplementowane komponenty interfejsu związane z realizacją zadań związanych z wyświetlaniem i edycją informacji tego rodzaju. Należą do nich: pola tekstowe, pola wyboru, przyciski, tabele, arkusze kalkulacyjne, proste edytory tekstu oraz – ze względu na charakter realizowanych projektów – również diagramy Gantta i wyspecjalizowane diagramy czasoprzestrzenne.

Bez względu na rodzaj komponentu, każdy z tych elementów ma trzy rodzaje wyrażeń

zapisanych w OCL:

- wyrażenia określające, co pokazywać (source expression),
- wyrażenia określające, co należy zrobić w przypadku akcji użytkownika (destination expression),
- wyrażenia określające stan (możliwa edycja, tylko do odczytu, schowany); (state expression).

Najprostszy komponent – pole tekstowe ma po jednym wyrażeniu z każdego rodzaju.

Przykładowo, dla formatki pokazującej dane kontaktu są to następujące wyrażenia:

```
src:          self.Turnover
dst:          self.Turnover = UserInput
state: if(self.isKindOf(Company)) then CS_ENABLED else CS_HIDDEN end
if
```

Definicja tabeli wymaga już więcej wyrażeń:

```
src:          self.MyTasks `wyświetlana kolekcja
```

Oraz dla każdej kolumny:

```
src:          self.ToDo; self.StartDate; self.EndDate
```

Najważniejszą cechą definicji interfejsów jest fakt, iż są one wyłącznie klientem maszyny wirtualnej. Na ich poziomie nie jest realizowany żaden model biznesowy, gdyż osoba definiująca interfejs nie ma możliwości ingerowania w prezentowane dane poza deklarowaniem powyższych wyrażeń.

Obecnie zaimplementowany jest kompilator interfejsu do systemu Windows. Możliwe jest jednak zaimplementowanie dowolnych innych transformacji – np. do stron HTML.

4. WYDAJNOŚĆ I SKALOWALNOŚĆ APLIKACJI WYKONYWANYCH W HumanStore

W systemach generowanych automatycznie częstym problemem jest wydajność ostatecznego produktu. Osiągnięcie w nich dobrej wydajności jest praktycznie niewykonalne w jednej iteracji implementacji i jest jedną z podstawowych barier, powodujących małą popularność tego rodzaju systemów na rynku.

HumanStore funkcjonuje jako system używany komercyjnie od ośmiu lat. Przez ten czas przeszedł optymalizację poszczególnych elementów technologicznych lub zmianę zasad ich działania. Każda następna wersja jest budowana w oparciu o doświadczenia i szczegółowe

pomiary wydajnościowe systemu na bazie działających złożonych aplikacji komercyjnych.

Obecna, czwarta wersja systemu jest już wersją dojrzałą, a aplikacje w niej tworzone są wydajnościowo porównywalne z podobnymi aplikacjami pisanymi w innych środowiskach programistycznych. Praktycznym dowodem na wydajność systemu są tworzone w nim aplikacje, z których korzysta na co dzień kilka tysięcy użytkowników.

Pierwszą aplikacją, rozwijaną wyłącznie jako model UML/OCL jest TILOS – system do zarządzania przedsięwzięciami w czasie i przestrzeni (Time-Location Planning), który łączy funkcjonalność programów typu CAD z zarządzaniem projektami. Produkt ten jest sprzedawany na całym świecie i uznawany za jedną z najlepszych aplikacji do zarządzania projektami liniowymi.

Drugą aplikacją modelowaną w całości w HumanStore jest HumanWork – system to pracy grupowej wokół realizowanych projektów. Jego użytkownicy mogą odwzorować strukturę swojej organizacji, planować i monitorować przebieg projektów, przechowywać całą dokumentację techniczną i procesową oraz modelować, uruchamiać i monitorować procesy przepływu pracy i obiegu dokumentów. W ramach wdrożeń, HumanWork został dostosowany do potrzeb szpitali, firm produkcyjnych, budowlanych oraz do prowadzenia projektów IT w bankowości. Maszyna HumanStore w wersji serwera aplikacji Serwisu Web pozwala na uruchomienie dedykowanych aplikacji HumanWork, dostępnych przez przeglądarki WWW, bez zmian w modelu.

5. NAJWAŻNIEJSZE SKŁADOWE HumanStore

5.1. MASZYNA WIRTUALNA

Transformacja modelu na działającą aplikację jest możliwa jedynie w sytuacji, gdy środowisko docelowe jest zbudowane na wzorcach projektowych zapewniających silne wsparcie dla wykonywania meta kodu. W maszynie HumanStore zaimplementowano wiele atomowych rozkazów przewidujących istnienie złożonych komponentów potrafiących je obsłużyć.

Przykładowo, w odróżnieniu od takich maszyn jak Java czy .NET, maszyna HumanStore, zamiast ograniczać się do mechanizmów typu garbage collector, posiada rozkazy dostępu do obiektów i kolekcji.

HumanStore izoluje kod wykonujący logikę biznesową od komponentów odpowiedzialnych za realizację zadań związanych z przechowywaniem danych, ich transportem czy prezentacją, dając w zamian bogaty zbiór narzędzi umożliwiających wydajną implementację tych komponentów, dostosowaną do środowiska i zadań służących konkretnemu celowi.

5.2. ZARZĄDZANIE PAMIĘCIĄ PODRĘCZNAŁ OBIEKTÓW

Mapowanie języka obiektowego na bezpośredni dostęp do odpowiednich obszarów pamięci, tak jak to się dzieje w C++ czy Java, oznacza konieczność deklaracji przez programistę, w jaki sposób informacje zawarte w obiektach mają być przesłane do innych mediów i jak mają być z nich czytane. W skrajnym przypadku, w kodzie opisującym jakąś logikę będą się znajdowały wyrażenia SQL zgodne składniowo z konkretną bazą danych. Natomiast maszyna HumanStore pracująca na obiektach, pozwala na dołączanie komponentów odpowiedzialnych między innymi za ich dostępność, przechowywanie i transmisję.

Jednym z najważniejszych komponentów HumanStore jest komponent odpowiedzialny za mapowanie struktury obiektowej na relacyjne bazy danych. Komponent ten implementuje szereg złożonych algorytmów i stosuje zaawansowane strategie synchronizacji danych pomiędzy pamięcią a zawartością bazy. Zapewnia to pełną konsystencję danych, redukuje ruch w sieci, znacząco zmniejsza ilość koniecznych połączeń z serwerem oraz, co najważniejsze, akceleroje szybkość wykonywanych operacji poprzez ich transakcyjne wykonanie w pamięci komputera. Dzięki temu, nawet tak zaawansowane aplikacje biznesowe jak HumanWork mają zapewnioną płynną pracę nawet przy kilkuset użytkownikach jednego serwera baz danych.

6. CHARAKTERYSTYKA IMPERATIVE OCL W HumanStore

W czasie, gdy podejmowano decyzję o użyciu języka OCL do specyfikacji dynamicznego zachowania systemu nie istniało pojęcie imperative OCL i nie istniała specyfikacja takiego języka. Przyjęto wtedy założenie, że składnia języka kompilowanego przez HumanStore powinna być identyczna z OCL 1.3, czyli taka, aby standardowy analizator poprawności wyrażzeń OCL nie wykazywał błędów.

Z funkcjonalnego punktu widzenia, w HumanStore zostały zaimplementowane wszystkie elementy języka OCL. Pominięto jedynie niektóre konstrukcje językowe, które przykładowo pozwalały na skrótowy zapis dłuższych wyrażzeń oznaczających dokładnie to samo. Konstrukcja kompilatora pozwala na ewentualne uzupełnienie tych elementów, co jednak nie rozszerzy w znaczący sposób możliwości specyfikacji określonych akcji.

Przykładem może być proponowany w oryginale skrótowy zapis:

```
self.SomeCollection.Value
```

Oznaczający sumę `Value` wszystkich elementów kolekcji. HumanStore wymaga pełnego zapisu formalnego:

```
self.SomeCollection.sum(element.Value)
```

Podstawową różnicą nie jest składnia, lecz sposób interpretacji w czasie analizy składniowej wyrażeń zawartych w post condition. Podczas gdy zapis `self.A = self.B and self.C = self.D` w oryginalnym języku jest interpretowany boolean expression, analizator HumanStore traktuje je jako złożenie dwóch assignment expression. Dla odpowiednich wartości 1,2,3,4 wyrażenie nie zwróci wartości fałsz, lecz zmieni wartości tych atrybutów odpowiednio na 2, 2, 4, 4.

Opisane podejście ma dużo zalet. Zapis OCL:

```
context Class::privatize(a : Attribute)
pre:
  self.feature->includes(a) and a.visibility = #public
post:
  a.visibility = #private and
  self.feature->exists(getter: Operation|
    getter.name.body = 'get'.concat(a.name.body) and
    getter.parameter->isEmpty and
    getter.isQuery)
```

Zapis HumanStore Imperative OCL:

```
context Class::privatize(a : Attribute)
pre:
  self.feature.includes(a) and
  a.visibility = public
post:
  a.visibility = private and
  self.operations.newElement(
    element.name = Concat('get', a.name.body) and element.isQuery = true)
```

Zauważmy, że zapis proponowany przez HumanStore jest bardzo bliski oryginalnemu post condition, a kod imperatywny jest jednocześnie faktycznym zapisem stanu obiektów po jej wykonaniu. Nawet imperatywny zapis `self.operations.newElement()` interpretowany jako „element, którego wcześniej nie było” nie łamie ogólnej zasady, że jest to jedno wyrażenie

opisujące zarówno akcję, jak i stan.

7. PODSUMOWANIE

Ideę konstrukcji gotowej aplikacji na podstawie modelu można zrealizować w prostszy i wydajniejszy sposób, niż przy użyciu narzędzi tworzących transformacje do modelu specyficznego dla danej platformy. Wymaga to zdefiniowania standardowych wymagań i interfejsów, jakie te środowiska powinny zapewniać na wyższym poziomie, niż poziom instrukcji maszyny wirtualnej typu Java. Przykładem takiego rozwiązania jest maszyna HumanStore, która dzięki sprawdzonej funkcjonalności i wydajności realizuje podstawowy cel Model Driven Architecture.

LITERATURA:

- [1] Booch, G., Rumbaugh, J., Jacobson, I. (1999), The Unified Modeling Language User Guide, Redwood City, CA: Addison Wesley Longman Publishing Co. Inc.
- [2] Bézivin, J, Gérard, S, Muller, P-A, Rioux, L Metamodelling for MDA
- [3] Mellor, S; Balcer, M: "Executable UML: A foundation for model-driven architecture", Preface, Addison Wesley, 2002
- [4] MDA Distilled, Principles of Model Driven Architecture, Stephen Mellor, Kendall Scott, Axel Uhl, Dirk Weise, Addison-Wesley Professional, 2004
- [5] MDA Explained, The Model Driven Architecture: Practice and Promise, Anneke Kleppe, Jos Warmer, Wim Bast, Addison-Wesley, 2003