Krzysztof Kluza

Czy kiedyś każdy myślący człowiek będzie mógł tworzyć aplikacje komputerowe?

1. Wprowadzenie

Nie byłoby współczesnej informatyki, gdyby nie prace nad rozwojem języków formalnych. Nikt dziś nie wyobraża sobie pisania programu użytkowego w kodzie maszynowym czy nawet w asemblerze. Znaczne zmiany zaszły zarówno w sferze samej składni języków, ich abstrakcji, jak i w sposobie posługiwania się nimi.

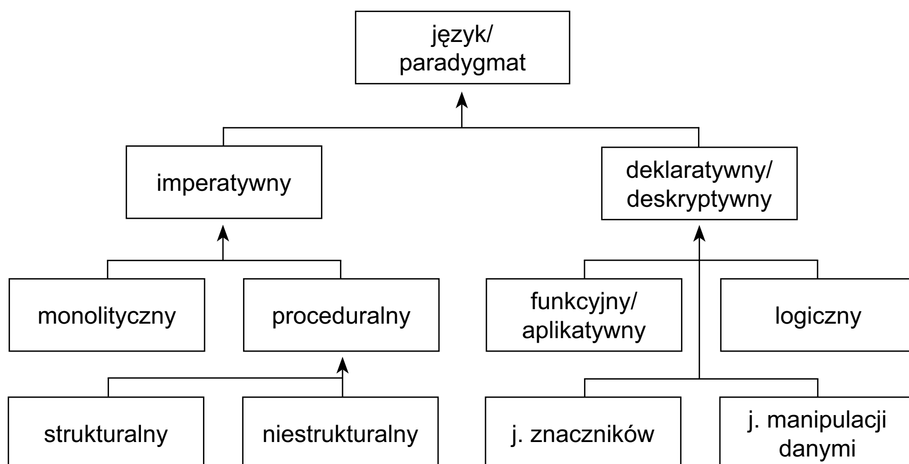
W artykule pokrótce został zaprezentowany temat przeobrażania się języków programowania oraz ewolucji paradygmatów. Omówiono także kwestie reprezentacji opisu składni języków formalnych – zarówno gramatyk, jak i alternatywnych konstruktów. Głównym celem artykułu jest ukazanie współczesnych kierunków rozwoju języków, w szczególności wizualnego podejścia do tworzenia oprogramowania, oraz próba udzielenia odpowiedzi na pytania: dokąd zmierza ewolucja języków w informatyce? czy informatyka odnalazła już swój język[1]? czy kiedyś każdy myślący człowiek będzie potrafił stworzyć aplikację komputerową, nie będąc programistą?

2. Ewolucja języków

Każdy język naturalny podlega procesowi ewolucji. Język sztuczny co prawda nie ulega podobnym zmianom, możemy jednak zaobserwować powstawanie nowych języków, różniących się zarówno poziomem abstrakcji, jak i sposobem ich funkcjonowania – np. paradygmatem programowania – co można nazwać swoistą ewolucją języków sztucznych.

Heller [2006] podjął próbę usystematyzowania paradygmatów programowania w postaci drzewiastej, pokazanej na rycinie 1. Nie jest to kompletna systematyka, wobec czego wyjaśnia, że „jest ekstremalnie trudno, o ile nie niemożliwe, by sklasyfikować wszystkie języki programowania w jedno drzewo kategorii".

[1] Trawestacja tytułu III Krakowskiej Konferencji Kognitywistycznej „Język odnaleziony".



Rycina 1. Systematyka paradygmatów programowania [Heller 2006]

Warto zwrócić uwagę, że jego podział przebiega dwubiegunowo. Z jednej strony mamy języki imperatywne, które odzwierciedlają sekwencyjne szeregowe przetwarzanie maszynowe, z drugiej języki deklaratywne, które nie są sekwencyjne. Tymczasem nasz mózg pracuje równolegle.

2.1. Historia języków programowania

Historia języków programowania sięga lat pięćdziesiątych, gdy powstały asemblery wykorzystujące instrukcje wbudowane w procesor. Są to języki niskopoziomowe, których rozkazy bezpośrednio i jednoznacznie mogą zostać przekształcone do kodu maszynowego. Niedługo potem narodził się język Fortran, który implementował m.in. obsługę tablic, skoków oraz pętlę *for*.

Lata siedemdziesiąte przyniosły rozkwit proceduralnych języków strukturalnych, takich jak Pascal i C, których składnia zawiera słowa kluczowe zaczerpnięte wprost z języka naturalnego, a programista może podzielić kod w wyodrębnione moduły. Języki te wyeliminowały instrukcje skoku z jednej części programu do innej. W tym samym czasie powstały także takie języki, jak SQL czy Prolog, wykorzystujące alternatywne, deklaratywne podejście do formułowania oczekiwań względem maszyny.

W kolejnej dekadzie język C++ rozpropagował paradygmat obiektowy. Jego założeniem jest odzwierciedlanie rzeczywistości w postaci obiektów i związków pomiędzy nimi. Rozwój postępował dalej i poziom abstrakcji wzrastał. Lata dziewięćdziesiąte to rozkwit języka Java, który dzięki Wirtualnej Maszynie Javy uniezależnił programowanie od platformy sprzętowej i systemu operacyjnego.

Obecnie można zaobserwować rozszerzanie obiektowego paradygmatu programowania. Jednym z takich rozszerzeń jest mechanizm aspektów. Umożliwiają one modularyzację zagadnień przekrojowych, takich jak autoryzacja i autentykacja, debugowanie, globalna obsługa wyjątków czy problemów związanych z wydajnością [Stochmiałek 2004].

Warte rozważań jest to, jakie będą kolejne kroki czynione w celu zwiększenia poziomu abstrakcji i ułatwienia pracy programistom. W szczególności interesujące jest to, czy języki wizualne zaczną być integralną częścią fazy implementacji oprogramowania (a nie tylko modelowania i projektowania).

2.2. Programowanie literackie

Jednym ze stylów programowania, o którym w kontekście ewolucji języków warto wspomnieć, jest wymyślony w latach osiemdziesiątych przez Knutha [1984] styl programowania literackiego (piśmiennego). Choć, patrząc z perspektywy czasu, wydaje się on ślepą ścieżką ewolucji, jest to jednak styl programowania, który w maksymalnym stopniu wykorzystuje język naturalny. Przyczynia się to m.in. do minimalizacji liczby popełnianych błędów w oprogramowaniu pisanym w tym stylu.

3. Język i gramatyka

Język formalny to każdy system, w którym stosując dobrze określone reguły z pewnego ustalonego zbioru, można uzyskać wszystkie zdania w tym języku [Rosek 2007]. Najpowszechniejszym formalizmem służącym do opisu składni języka jest gramatyka, czyli zbiór reguł określających zasady tworzenia poprawnych wypowiedzi w danym języku.

Przykładem gramatyki generatywnej dla pewnego podzbioru języka polskiego może być zbiór, do którego należą takie reguły [fragment zbioru reguł za: Rosek 2007]:

<zdanie> ::= <grupa podmiotu> <grupa orzeczenia>
<grupa podmiotu> ::= <określenie podmiotu> <podmiot zasadniczy>
<określenie podmiotu> ::= <przydawka>
<przydawka> ::= wysoki | wesoły
<podmiot zasadniczy> ::= Janek
<grupa orzeczenia> ::= …
…

Tak zdefiniowany zbiór reguł pozwoli utworzyć pewne zdania w języku polskim, np.: „Wysoki Janek jest piekarzem".

4. Alternatywne metody definiowania składni języka

Choć przywykliśmy do tego, że do definiowania języka (w szczególności języka programowania) używamy gramatyki, istnieją języki formalne, dla których standardowa gramatyka nie wydaje się optymalnym konstruktem do opisu ich składni.

4.1. XML i DTD

XML (*Extensible Markup Language*) to uniwersalny język znaczników przeznaczony do reprezentowania różnych danych w ustrukturalizowany sposób. Jest językiem formalnym, którego specyfikacja została opisana za pomocą gramatyki w notacji EBNF [XML 2008]. Jednak do opisu struktury i zawartości konkretnych dokumentów XML nie potrzeba używać notacji EBNF – do formalnego zdefiniowania dokumentu XML można użyć definicji typu dokumentu DTD (*Document Type Definition*), którego struktura jest prosta i czytelna dla użytkownika. Dla przykładu poniżej zamieszczono fragment pliku XML (z wewnętrznie umieszczoną definicją typu dokumentu) ze spisem znajomych osób:

```
<?xml version=”1.0”?>
<!DOCTYPE znajomi [
	<!ELEMENT znajomi (osoba+)>
	<!ELEMENT osoba (imie+, nazwisko)>
	<!ELEMENT imie (#PCDATA)>
	<!ELEMENT nazwisko (#PCDATA)>
]>
<znajomi>
	<osoba>
	<imie>Krzysztof</imie>
	<imie>Marek</imie>
	<nazwisko>Kluza</nazwisko>
	</osoba>
	…
</znajomi>
```

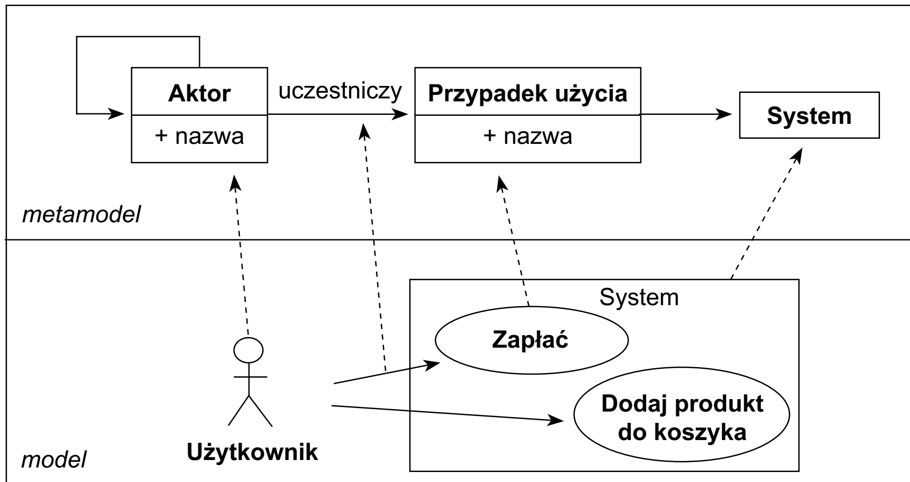
W związku z tym, że struktura samego DTD jest niezgodna ze standardem XML, obecnie następcą DTD jest XML Schema [XMLSchema 2006], który jest co prawda znacznie bardziej skomplikowany, ale posiada o wiele większe możliwości.

4.2. Metamodel MOF

Do definiowania języków, których zdaniami są ciągi symboli, możemy użyć gramatyki, np. w notacji EBNF, która z jednego symbolu może wywieść inny symbol. Jednak w przypadku, gdy zamiast ciągu symboli mamy inne artefakty – jak elementy wizualne diagramu – bardziej naturalne wydaje się użycie wizualnego odpowiednika gramatyki.

Do wizualnej definicji języków wizualnych (ale nie tylko) ma służyć metamodel MOF. Standard ten został wprowadzony przez Object Management Group, a wypracowany na potrzeby formalnej definicji składni języka UML [MOF 2006].

Na rycinie 2 zaprezentowano uproszczony metamodel dla diagramu przypadków użycia UML [wzorowany na: Robin 2006].



Rycina 2. Uproszczony metamodel dla diagramu przypadków użycia UML

5. Modelowanie – programowanie wizualne?

Znany aforyzm mówi, że jeden obraz znaczy więcej niż tysiąc słów. W informatyce można by sparafrazować tę sentencję i powiedzieć: jeden model znaczy więcej niż kilka stron opisu. Modele istotnie stanowią ważną część procesu analizy i projektowania systemów informatycznych. W fazie implementacji służą one jednak właściwie tylko jako wytyczne dla programistów, a nie jako integralna część oprogramowania.

Programowanie wizualne może stać się kolejnym etapem na drodze ewolucji języków programowania.

5.1. EBNF a metamodel UML

Modele wydają się nam dużo bardziej naturalne niż kod programu, a jeden z najpopularniejszych języków modelowania – UML – formalnie można opisać za pomocą metamodelu.

Skoro metamodel UML jest na tym samym poziomie abstrakcji co gramatyka EBNF (metamodel definiuje strukturę modelu, a gramatyka danego języka strukturę programu napisanego w tym języku), interesującym podejściem wydaje się dokonanie tłumaczenia między modelem a tradycyjnym kodem programu.

Od czasu publikacji Alanena i Porresa [2003] próby takie są podejmowane, jednak póki co rozważania te sprawiają wrażenie czysto akademickie. Nie są one ustandaryzowane i zarówno metoda, zgodnie z którą ma się odbywać takie przejście, jak i sama reprezentacja jest niejednoznaczna (np. niektóre rodzaje relacji w gramatyce można przedstawić na różne sposoby w metamodelu).

5.2. MDA

MDA to inicjatywa grupy OMG wyznaczająca nowy paradygmat rozwoju oprogramowania. Paradygmat ten definiuje trzy poziomy abstrakcji modeli:
- model biznesowy (nieprecyzujący zakresu odpowiedzialności oprogramowania, a jedynie określający reguły biznesowe oraz słownictwo biznesowe),
- model niezależny od platformy,
- model zależny od platformy,

a także zasady ich transformacji [MDA 2003]. MDA podnosi zatem poziom abstrakcji tworzenia oprogramowania poprzez uniezależnienie procesu tworzenia oprogramowania od platformy technologicznej.

5.3. xUML

Wykonywalny UML – xUML (*Executable UML*), profil UML zaproponowany przez Stevena Mellora – może stanowić jeden z filarów, na których wspiera się MDA (*Model -Driven Architecture*) [Mellor i Balcer 2002].

xUML nawet bardziej niż styl programowania piśmiennego niweluje rozdźwięk pomiędzy dokumentacją a kodem – graficzne modele bowiem w całości zarówno stanowią specyfikację projektu, jak i mogą zostać automatycznie przekształcone na kod implementujący tę specyfikację. Rycina 3 prezentuje tabelę ukazującą poszczególne elementy systemu i ich odpowiedniki w modelu xUML [Mellor i Balcer 2002].

Koncepcja	Nazwany	Modelowany jako	Wyrażony jako
świat jest pełen obiektów	dane	klasy, atrybuty, asocjacje, ograniczenia	diagram klas UML
obiekty mają cykle życia	sterowanie	stany, zdarzenia, przejścia, procedury	diagram maszyny stanów UML
obiekty robią rzeczy na różnych etapach	algorytmy	akcje	język akcji

Rycina 3. Tabela ukazująca elementy systemu i ich realizację w xUML

Choć modelowanie logiki biznesowej w języku UML odbywa się na wysokim poziomie abstrakcji, w xUML-u każda procedura składa się ze zbioru akcji, których poziom abstrakcji jest relatywnie niski (model zależny od platformy). Na dodatek ani notacja, ani składnia języka akcji nie są ustandaryzowane, zdefiniowano jedynie semantykę takich akcji [Mellor i Balcer 2002].

5.4. DSML

Obecnie wysoki poziom abstrakcji modelowania zarówno struktury systemu, jak i pojedynczych zachowań umożliwiają jedynie języki DSML (*Domain-Specific Modeling Languages*), czyli wizualne języki przeznaczone do modelowania w określonej dziedzinie. Ich zastosowanie jest jednak ograniczone do bardzo konkretnych problemów, a samo modelowanie zwykle przebiega również w ściśle określony sposób. Na rycinie 4 przedstawiono prosty przykładowy diagram DSML aplikacji działającej w dziedzinie kurierskiej.

Podsumowanie

Podobno na początku XXI wieku jeden z twórców UML-u – Ivar Jacobson – przepowiadał, że wkrótce UML stanie się uniwersalnym językiem programowania. Obecnie nic nie wskazuje na to, by jego proroctwo miało się szybko ziścić. Co prawda istnieją aplikacje, które z modeli potrafią wygenerować wiele linii kodu (np. z diagramu klas UML wiele programów generuje kod struktury powiązanych ze sobą klas), co może oszczędzić znaczną ilość czasu, jednak wciąż pozostają pewne aspekty (głównie dotyczące zachowania systemu), które muszą być zaprogramowane standardowymi metodami.

Jednak już dziś w zespole tworzącym (implementującym) oprogramowanie jest miejsce dla osób, które niekoniecznie muszą programować. Od takich ludzi będą jednakże wymagane szczególne umiejętności w zakresie modelowania i projektowania systemów, a także dobra znajomość dziedziny modelowanego problemu.

Aby tworzyć kompletne oprogramowanie, wciąż będzie potrzebna znajomość tradycyjnych technik programowania. Stan ten mógłby ulec zmianie tylko wówczas, gdyby rozwiązano problem dostatecznie precyzyjnego języka definiującego zachowanie systemu, ale będącego na znacząco wyższym poziomie abstrakcji niż docelowy kod źródłowy programu.

BIBLIOGRAFIA

Alanen M., Porres I. (2003). *A relation between context-free grammars and meta object facility meta-models*. Technical Report 606, TUCS Turku Center for Computer Science.

Heller C. (2006). *Cybernetics Oriented Programming (CYBOP) an Investigation on the Applicability of Inter-disciplinary Concepts to Software System*. Tux Tax.

Knuth D.E. (1984). *Literate Programming.* „The Computer Journal" 27, 2, s. 97–111.

MDA (2003). *MDA Guide Version 1.0.1* [dostęp: 2009–03–12] Dostępny w Internecie: <http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf>.

Mellor S., Balcer M. (2002). *Executable UML: A foundation for model-driven architecture*, chapter 1.1. *Raising the Level of Abstraction*, Addison Wesley.

MOF (2006). *Meta Object Facility (MOF) Core Specification Version 2.0* [dostęp: 2009–03–12], Dostępny w Internecie: <http://www.omg.org/docs/formal/06-01-01.pdf>.

Robin J. (2006). *Lecture 8: Meta-modeling with MOF2 and UML2 Profiles* [dostęp: 2009–03–12], Dostępny w Internecie: <http://www.cin.ufpe.br/~in0980/2006/>.

Rosek J. (2007). *Wykład z Teorii Języków i Automatów*. Instytut Informatyki UJ.

Stochmiałek M. (2004). *Wprowadzenie do programowania aspektowego* [dostęp: 19–05–2004], Dostępne w Internecie: <http://www.codeguru.pl/article-143.aspx>.

XML (2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition) Specification* [dostęp: 2009–03–12], Dostępne w Internecie: <http://www.w3.org/TR/2008/REC-xml-20081126/>.

XMLSchema (2006). *XML Schema 1.1* [dostęp: 2009–03–12], Dostępne w Internecie: <http://www.w3.org/XML/Schema>.