Benedykt Burgielski
Akademia Finansów i Biznesu Vistula – Warszawa

Inżynieria oprogramowania. Przegląd metodyk

Streszczenie

W artykule dokonano przeglądu niektórych metodyk inżynierii oprogramowania oraz oceniono ich wady i zalety. Przedstawiono najnowsze tendencje szczególnie w obszarze rozwiązań typu BPMN. Pokazano przyczyny, które wymusiły stworzenie narzędzi w ramach inżynierii oprogramowania. W kolejnych częściach artykułu zawarto opisy trzech metodyk – obiektowej, strukturalnej i BPMN. W zakończeniu porównano i oceniono przydatność poszczególnych metodyk.

Słowa kluczowe: inżynieria oprogramowania, metodyki, BPMN.

Kody JEL: L86

Wstęp

Proces pisania programów komputerowych jest procesem wyjątkowym, nie tylko ze względu na jego efekty, zarówno naukowe, jak i ekonomiczne. Może on prowadzić do doświadczeń o charakterze estetycznym, podobnym do tych, których doświadczamy czytając poezje czy oglądając obrazy – jest sztuką. Dane zagadnienie opracowane przez różnych programistów będzie miało zawsze inny charakter, co jest domeną sztuki (Knuth 1968). Nauka w określonych warunkach daje zawsze takie same rozwiązania. Programowanie komputerów jest dziedziną bardzo szybko się rozwijającą. Poczynając od zastosowań czysto numerycznych, takich jak obliczenia rozwiązań równań, numeryczna ekstrapolacja czy całkowanie funkcji matematycznych, po zastosowania „nie numeryczne", takie jak: sortowanie, przetwarzanie danych, programy tłumaczące z jednego języka na inny, rozpoznawanie obrazów (sztuczna inteligencja) czy też symulacja zagadnień spotykanych w życiu codziennym.

Gwałtowny i powszechny rozwój wykorzystania środków informatyki w biznesie i życiu codziennym zapoczątkował lawinowe zapotrzebowanie na oprogramowanie obsługujące te zastosowania. Programy stawały się coraz dłuższe i coraz bardziej skomplikowane. Stworzyło to potrzebę ujednolicenia zasad pisania programów komputerowych i ich dokumentacji, czyli inżynierii oprogramowania.

Oprogramowanie jako produkt możemy podzielić na dwie podstawowe kategorie:

- oprogramowanie ogólnego zastosowania tzw. produkty z półki – wytworzone i sprzedawane przez firmy dla szerokiego grona odbiorców (brak identyfikacji konkretnych potrzeb poszczególnych klientów);
- oprogramowanie wytworzone na zamówienie i na potrzeby określonego klienta.

Większość dostępnego oprogramowania należy do pierwszej kategorii, ale największe nakłady przeznaczane są na produkty należące do kategorii drugiej.

Kryzys oprogramowania

W początkach ery programowania jedna osoba mogła przeanalizować problem, opracować jego rozwiązanie i napisać gotowy program. Ale z biegiem czasu wszystko uległo zmianie, tworzenie różnego rodzaju złożonych systemów, których domaga się świat biznesu, wymaga podejścia zespołowego. Wiedza stała się zbyt specjalistyczna, aby jedna osoba była w stanie pojąć różne aspekty biznesu, zrozumieć problem, zaprojektować rozwiązanie, przetłumaczyć je na kod programu i wdrożyć na sprzęcie, sprawdzając przy tym, czy wszystkie elementy współpracują ze sobą.

W skład zespołu muszą wchodzić analitycy kontaktujący się z klientem i rozumiejący jego problem, projektanci, którzy stworzą projekt rozwiązania, programiści, którzy to rozwiązanie zamienią w program, i inżynierowie systemowi, którzy wdrożą system. W procesie tworzenia oprogramowania nie można pominąć nikogo z wymienionych, należy odpowiednio wykorzystać ich umiejętności i dać im wystarczająco dużo czasu na wykonanie przypadających im zadań. Rezultatem procesu musi być także tworzenie łańcucha produktów roboczych, których wykonanie będzie świadczyło o postępie prac i pozwoli na egzekwowanie odpowiedzialności za wykonanie poszczególnych zadań. Proces musi zapewniać łączność między poszczególnymi etapami pracy, co pomaga w powstawaniu nowych idei.

W USA większość z 250 mld dolarów wydawanych rocznie na tworzenie oprogramowania jest tracona na opóźnione, niedokończone i przerwane projekty.

- 27% projektów powstaje w założonym czasie, budżecie i funkcjonalności,
- 33% projektów przekracza czas, budżet i ma mniejszą funkcjonalność,
- 40% projektów jest przerywanych,
- 53% projektów przekracza koszty o 51% i więcej,
- 68% projektów przekracza czas o 51% i więcej.

Do najważniejszych przyczyn, które wywołały kryzys na rynku oprogramowania należy zaliczyć:

- dużą złożoność systemów informatycznych,
- niepowtarzalność poszczególnych przedsięwzięć,
- nieprzejrzystość procesu budowy oprogramowania (np. trudności w ocenie stopnia zaawansowania prac),
- pozorną łatwość wytwarzania i dokonywania poprawek.

Przyczyną podstawową jest rosnąca skala złożoności oprogramowania. Przykładem może być proces realizacji systemu operacyjnego Windows 2000, w którym można wyróżnić kroki milowe:

- październik 1997 roku – 1. beta wersja (NT 5.0),

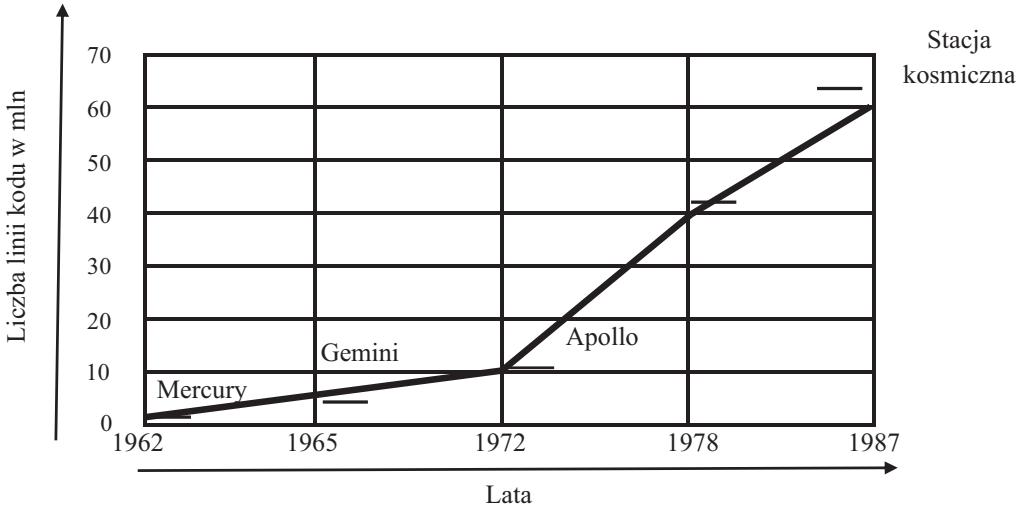
- kwiecień 1999 roku – 3. beta wersja,
- styczeń 2000 roku – wersja komercyjna,
- lipiec 2000 roku – 1. zestaw poprawek,
- 40-60 mln linii kodu,
- ok. 2000 programistów.

Projekt trwał kilka lat, a w ostatniej fazie pracowało nad nim kilka tysięcy osób.

Dobrym przykładem wzrostu złożoności oprogramowania są projekty realizowane przez NASA. Na wykresie 1 pokazano kilka projektów uszeregowanych według ilości linii kodu.

Wykres 1

Wzrost złożoności oprogramowania



Źródło: Unold (2015).

Można zauważyć, że ilość ta wzrosła od 1 mln do 60 mln na przestrzeni 25 lat.

Czy tworzenie oprogramowania to wciąż sztuka, czy już tylko przemysł? Bezpowrotnie minęły czasy, gdy aplikacje tworzone były przez zapaleńców-samotników w przysłowiowym garażu. Dzisiaj firmy zajmujące się tworzeniem oprogramowania to sprawnie zarządzane przedsiębiorstwa, w których programiści są tylko niewielką – chociaż niezwykle istotną – częścią zespołu. Nowoczesne oprogramowanie nie może powstać bez pracy doświadczonych analityków i projektantów, zespołu testerów, a często także grupy zajmującej się wdrożeniem projektu. Nawet przedstawiciele takich profesji, jak np. graficy, są potrzebni do tworzenia nie tylko gier komputerowych, ale też oprawy graficznej aplikacji użytkowych. Duże projekty informatyczne wymagają dodatkowo opieki od strony bizneso-

wej. Umiejętne zarządzanie całym procesem powstawania projektu informatycznego to nie tylko nadzorowanie kwestii technicznych, ale też relacji z klientem, sprawdzanie kruczków prawnych w umowach, tworzenie strategii sprzedaży i wdrażania gotowego oprogramowania. Wiele zależy od organizacji zaplecza marketingowego. Sporo ciekawych programów przechodzi bez echa tylko dlatego, że niewiele osób dowiaduje się o ich istnieniu. Nie można jednak zapominać, że w centrum tego procesu cały czas znajduje się grupa programistów, którzy zamieniają pomysły klienta i projektantów w działający system. Wszyscy, którzy wiążą swoją zawodową przyszłość z programowaniem, powinni zdawać sobie sprawę, że doskonała znajomość Javy, C++ czy C# i różnych technik programistycznych nie wystarcza do przygotowania nowoczesnej aplikacji.

Środowiska programistyczne już od dawna tworzone są z myślą o pracy grupowej i różnych etapach tworzenia oprogramowania. Są to odrębne, ale współpracujące ze sobą produkty: dla analityków, projektantów, programistów czy osób zarządzających projektem. Nie sposób przedstawić wszystkie zagadnienia związane z tworzeniem oprogramowania. Wystarczy odwiedzić dowolną księgarnię informatyczną, aby przekonać się, jak olbrzymia wiedza związana jest chociażby z analizą systemową. Jednak rzut oka na kolejne etapy tworzenia oprogramowania pozwoli się zorientować, jakie czynności wykonywane są w kolejnych częściach całego procesu i jakiego typu narzędzia są potrzebne.

Silne i słabe strony programowania strukturalnego oraz jego ograniczenia

Programowanie strukturalne opiera się na zastosowaniu modułów, które realizują pojedyncze zadania, funkcje. Te moduły są logicznie i obliczeniowo spójne. Zbudowane na podstawie tych ideologii programy (strukturalne) są łatwiejsze do testowania, wyłapywania błędów i utrzymania niż programy nie posiadające takiej struktury (Dijkstra 1968; Knuth, Floyd 1971; Dahl i in. 1972). Niezależne moduły mogą być niezależnie kodowane i testowane. Elementy sterowania zawarte w takich programach pozwalają na testowanie całego programu „z góry na dół" (*top-down*), moduł za modułem. W momencie pojawienia się błędu, łatwo jest szybko błędny kod izolować, ponieważ program dzielony jest na moduły. Podczas fazy utrzymania (eksploatacji), niezależne moduły mogą być wymieniane, modyfikowane lub nawet usuwane bez powodowania efektu replikowania błędów.

Dekompozycja funkcjonalna jest metodologią sterowaną procesami (*process driven methodology*). Chociaż logiczne moduły są niezależne, to już dane nie, co może prowadzić do niepożądanych zjawisk (*ripple effects*) podczas testowania, usuwania błędów i utrzymania oprogramowania.

Wejścia i powiązane idee

Zanim zaczniemy projektować program, używając funkcjonalnej dekompozycji, musimy zdefiniować logiczne struktury danych, główne procesy i zasadnicze funkcjonalności.

Często wyniki takiej analizy są zawarte w dokumentacji pod nazwą „specyfikacja wymagań". Ponadto, jest bardzo pożądane, aby określić fizyczne struktury danych, struktury zbiorów, architekturę baz danych, jak również wymagane struktury danych na wejściu i wyjściu oraz wykorzystywane algorytmy. Do dokumentowania strukturalnego projektu software mogą być użyte takie narzędzia, jak:

- szablony HIPO,
- diagramy struktur (*structure charts*),
- logiczne schematy blokowe,
- schematy Nassi-Shneidermana,
- drzewa decyzyjne,
- tablice decyzyjne,
- pseudokod,
- strukturalny język naturalny (np. strukturalny angielski),
- schematy IPO.

Sterowania

Dobrze zaprojektowany program „strukturalny" składa się z modułów realizujących jedną funkcję, połączonych elementami sterującymi (*control*), a swoim wyglądem przypomina strukturę organizacji wojskowej lub zhierarchizowaną strukturę przedsiębiorstwa. Każdy moduł jest przedstawiony w postaci prostokąta. Na samej górze tej struktury znajduje się pojedynczy moduł zwany pniem (*root*) albo główny moduł sterowania. Od niego odchodzą pozostałe. Inaczej mówiąc, moduł pnia „woła" te moduły, które bezpośrednio od niego odchodzą. Te moduły stanowią drugi poziom tej struktury i czasem nazywa się je „dziećmi". Moduły z drugiego poziomu „wołają" te, które są do nich „podczepione". Te podczepione moduły stanowią trzeci poziom struktury programu. I tak dalej.

Moduł „wołający", inaczej zwany rodzicem (*parent*) przekazuje „w dół" dane i polecenia sterujące do modułu „wołanego" i z kolei otrzymuje dane i sterowania od modułu wołanego („w górę").Po wywołaniu modułu z niższego poziomu, sterowanie zawsze powraca do modułu wywołującego.

Wyzwalacze (zdarzenia uruchamiające)

Alternatywnym podejściem jest rozpoczęcie od zdarzenia uruchamiającego działanie program (*trigger event*). Zdarzenie takie rozpoczyna działanie programu lub powoduje zmianę stanu ze stanu oczekiwania na stan pracy. Niektóre programy są „wyzwalane" przez zdarzenie asynchroniczne, takie jak np. pojawienie się jakiejś transakcji, klienta w kolejce (programy obsługi masowej) lub przez jakieś przerwanie. Inne aplikacje są uruchamiane poprzez zdarzenia związane z czasem (*clock driven*). Na przykład programy mogą być urucha-

miane o tej samej porze co tydzień, dane naukowe mogą być gromadzone z częstotliwością co sekundę itd. Element sterowania najwyższego poziomu powinien potrafić obsługiwać takie zdarzenia i odpowiadać na nie poprzez uruchomienie programu.

Struktury danych

Następną techniką projektowania struktury programów jest analiza struktur danych. Każdy program przyjmuje dane wejściowe i przetwarza jest do postaci wymaganej na wyjściu. Można powiedzieć, że to dane kierują procesem projektowania softu.

Należy więc analizować dane wyjściowe i określić kolejność, w jakiej różne wyjściowe podstruktury są składane. Następnie należy określić wejściowe podstruktury i określić algorytmy, które z tych struktur będą generować struktury wyjściowe.

To struktury danych będą wymuszać zarówno porządek, w którym logiczne zadania (*task*) muszą być wykonane, jak i logiczne struktury, które wspomagają te zadania.

Ogólnie, sekwencyjne struktury danych wywołują (implikują) sekwencyjną logikę, struktury warunkowe implikują warunkową logikę, a struktury powtarzające się (*repetitive*) implikują logikę powtarzalną.

Charakterystyka metod obiektowych

Metody obiektowe projektowania systemów informatycznych wywodzą się z potrzeb zaistniałych ze strony programowania obiektowego. Programiści wykorzystujący takie języki, jak Ada, Smalltalk czy Simula 67 interesowali się takimi zagadnieniami, które pozwoliłyby patrzeć w innym sposób na otaczający nas świat i najlepiej odzwierciedlałyby własności tych języków w modelowaniu otaczającej rzeczywistości. W metodach obiektowych nie występuje zdecydowane oddzielenie fazy analizy od fazy projektu. W fazie projektu występuje uszczegółowienie dokonań fazy analizy.

Metody obiektowe są obecnie trwałym składnikiem kultury tworzenia oprogramowania, aczkolwiek nadal są ignorowane przez wielu głównych producentów oprogramowania. Wobec wzrastającego zapotrzebowania na systemy rozproszone, najbardziej naturalne wydaje się podejście obiektowe dzięki uwzględnieniu w nim możliwości ukrywania informacji i wymiany komunikatów. Coraz większe przywiązywanie wagi do kosztów może też doprowadzić do uznania, że kluczową kwestią w programowaniu, projektowaniu i analizie jest możliwość wielokrotnego używania aparatu programistycznego (wyłaniający się rynek komponentów dostarcza dużą liczbę wielokrotnie używanych programów). Większość ekspertów wierzy, że pomyślny rozwój produktów komponentowych będzie wymagał skupienia zwiększonej uwagi na architekturze oprogramowania i wzmożenia nacisku na rozwój modelowania obiektowego.

Metody obiektowe - podejście historyczne

Wzrost popularności obiektowości w informatyce spowodował prawdziwą eksplozję metodyk i notacji określanych jako „obiektowe", wśród nich: BON, Catalysis, DOOS (Wirfs/Brock) (Wirfs, Brock i in. 1990), EROOS, Express, Fusion, Goldberg/Rubin, MainstreamObjects, Martin/Odell, MOSES, Objectory (Jacobson), OMT (Rumbaugh) (Rumbouhg i in. 1991), OOA/OOD (Coad/Yourdon) (Coad, Yourdon 1989; Coad, Yourdon 1991), OOAD (Booch 1994), OSA, Sintropy, OOSA (Shlaer/Mellor) i inne. Mimo różnic zarówno w podejściu, jak i w przeznaczeniu, metodyki te mają ze sobą wiele wspólnego. Dość powszechne stało się odczucie, że różnice między notacjami wprowadzanymi w tych metodykach są drugorzędne.

UML ma (w założeniu) dokonać unifikacji notacji używanych w istniejących metodykach. W odróżnieniu od metodyk obiektowych, które oprócz notacji ustalają także postępowanie w każdej fazie projektu, UML jest tylko zestawem pojęć i notacji. Może on być użyty do dowolnej metodyki. Zdaniem autorów, UML swoim zakresem przykrywa większość elementów i faz procesów analizy i projektowania.

Metoda OMT była metodą sterowaną danymi; wywodziła się z projektowania relacyjnych baz danych. W metodzie OMT naśladowano podejście Coada polegające na dodawaniu operacji do opisów utworzonych przy użyciu encji i typów w celu zbudowania modelu klasowego, lecz zastosowano inną notację. Metoda OMT była nie tylko sterowana danymi, lecz oddzielała procesy od danych, używając diagramów przepływu danych, niezależnie od schematów występujących klas. Uwypuklała użycie diagramów przejść stanów do opisu cyklu życia egzemplarzy (*entity*). Dawała bardzo pożyteczną wskazówkę, jak połączyć diagramy obiektowe z bazami danych. Zaletą metody OMT jest jej relacyjny charakter. Metoda OMT była trudna do nauczenia i niewiele pomagała w wyrażaniu reguł i ograniczeń biznesowych.

Metoda OMT była (swego czasu) jedną z najpopularniejszych metod analizy obiektowej. Zapoczątkowały ją prace Michaela Blahy, Billa Premerlaniego i Jamesa Roumbaugha oraz ich kolegów z firmy General Electric. Podejście to ma silne korzenie w tradycyjnej metodzie strukturalnej i pozwala korzystać z bogatej, lecz skomplikowanej i szczegółowej notacji. Złożoność była częściowo ceną za możliwość korzystania z pewnych narzędzi, które pozwalały automatycznie generować nagłówki kodu. Na notacji OMT jest zasadniczo oparty język UML.

Metoda OOSE była obiektową metodą analizy i projektowania, która wywodziła się z metody Jacobsona, zwanej Objectory (od *Object Factory*). Główną zaletą tej metody była koncepcja rozpoczynania analizy od rozważania przypadków użycia. Ta technika stanowiła poważny krok w kierunku analizy obiektowej i została szeroko przyjęta. Była to pierwsza metoda, która chociaż częściowo, obejmowała proces tworzenia. W metodzie OOSE zaleca się, by jej przypadki użycia były połączone z trzema rodzajami obiektów implementacji – obiektami encji, obiektami interfejsu i obiektami sterownika. Jednak wprowadzenie obiektów sterownika może później w procesie analizy poważnie naruszyć zasadę kapsułkowania, powodując powstawanie wielu problemów związanych z utrzymywaniem systemu. Wynika

to stąd, że sterowniki często działają jak procedury główne i muszą znać wiele informacji o innych obiektach, którymi sterują; kiedy zmienia się jeden z tych obiektów, wtedy trzeba modyfikować sterownik.

Do pełnego opisania przedsięwzięcia biznesowego w metodzie Jacobsona tworzone są dwa rodzaje modeli – model przypadków użycia i model obiektów.

Każdy z twórców poszczególnych szkół projektowania obiektowego wykorzystuje własną notację. Sprzeczności w budowie poszczególnych rodzajów diagramów wynikają przede wszystkim z różnic fizycznych odwzorowań klas i obiektów, relacji, usług i przepływów.

UML zapewnia wspólną i zwięzłą notację przy pomocy której opisywane są obiekty i komponenty systemów informatycznych. Minimalizuje to wysiłek deweloperów software'owych, ponieważ muszą się oni nauczyć tylko jednego języka (przynajmniej w teorii). UML jest po prostu krokiem w dobrym kierunku.

W metodyce Lynx w trakcie modelowania przez cały czas dominującą rolę odgrywają procesy biznesowe. Metodologia Lynx oparta jest na dwóch typach diagramów:
- Diagramy hierarchiczne procesów (*Process Hierarchy Diagram*).
- Diagramy wątków procesów (*Process Thread Diagram*).

Diagramy hierarchiczne procesów obrazują związki statyczne zachodzące między procesami. Służą one do graficznego pokazania powiązań i zależności, zachodzących w organizacji między procesami biznesowymi oraz ich podziałów na różnych stopniach i poziomach hierarchii.

Elementem nadrzędnym jest modelowane przedsiębiorstwo (lub jego część). Składa się ono z określonej liczby kluczowych procesów biznesowych, a te z kolei zbudowane są z poszczególnych grup biznesowych (przy mało skomplikowanym procesie biznesowym poziom ten może być pominięty). Każda grupa biznesowa składa się z określonej liczby elementarnych procesów biznesowych (EPB).

Metodyka The Select Perspective jest jedną z ciekawszych metod obiektowej analizy i projektowania, opracowaną w firmie SELECT Software Tools. W 2002 roku firma analityczna Gartner Group uznała metodykę Select Perspective za jedną z wiodących metodyk wytwarzania systemów na świecie (software 2015). Metodyka Select Perspective łączy w sobie trzy istotne aspekty budowy współczesnych systemów informatycznych:
- modelowanie procesów biznesowych,
- modelowanie systemu,
- modelowanie relacyjnej bazy danych.

Unikalną rzeczą na rynku metodyk jest pełne wsparcie dla podejścia komponentowego, opartego na współpracy dostawców i odbiorców komponentów. Podejście to staje się obowiązującym wzorcem nowoczesnego procesu produkcji oprogramowania i uważane jest za rodzaj „rewolucji przemysłowej". Opiera się na koncepcji budowy systemu niewielkich modułów o dobrze określonych usługach, możliwych do wielokrotnego zastosowania i ła-

twych do testowania. Użycie komponentów pozwala dostarczać rozwiązania szybciej, taniej i lepiej, niż gdyby były tworzone od podstaw.

Czy UML to panaceum na wszystkie problemy programistyczne?

UML definiuje „przemysłową" notację i semantykę wykorzystywaną przy projektowaniu i implementacji software'u używając technologii OO (*object oriented*) lub opartej na komponentach. Trudno jest dzisiaj znaleźć jakąś książkę traktującą o w/w tematach, która nie używałaby UML.

UML nie jest wystarczający

Zasada wielości modeli sformułowana przez Agile Manifesto (AM) mówi, że należy mieć do dyspozycji wiele technik modelowania, traktowanych jako intelektualna skrzynka z narzędziami, jeżeli chce się być efektywnym jako programistyczny deweloper. W artykule pt. *Artifacts for AM* przedstawiono przegląd wielu popularnych artefaktów modelowania, włączając w ten zbiór narzędzi.

Każdy model ma swoje silne i słabe strony, ale żaden z nich nie jest wystarczający i spełniający wszystkie wymagania środowiska programistycznego (analityków, projektantów i programistów). I chociaż UML jest dostatecznie silnym narzędziem, to sam nie spełnia wszystkich wymagań. Na przykład, chcąc zaprojektować GUI dla aplikacji, użyteczne jest stworzenie diagramu modelującego przepływy i sterowania, które mają miejsce w interfejsie. UML nie oferuje jak dotychczas takich diagramów. Podobnie istnieje silna potrzeba tworzenia modeli danych, reprezentujących fizyczne aspekty baz danych. Jak dotychczas, UML nie oferuje takich diagramów w swoim repertuarze.

Jak już wcześniej wspomniano UML nie jest idealne. Chociaż diagramy aktywności można użyć do modelowania procesów biznesowych, to istnieje znacznie lepiej do tego celu przystosowane narzędzie: jest nim *Business Process Modelling Notation* (BPMN 2015). Wszystkie znaki wskazują, że narzędzia BPMN zostaną wkrótce włączone do standardu UML.

Podobnie UML nie dostarcza w swym standardzie narzędzi pomocnych do modelowania usług sieciowych. Nie ma w nim „diagramów sieciowych". Nawet RUP, wywodzący się z tego samego nurtu, który jest organizacją propagującą UML, nie zawiera explicite (w sposób jawny) takich artefaktów, jak: reguły biznesowe, modele danych, diagramy interfejsów użytkownika.

Zapomnij o wykonywalnym UML'u (przynajmniej na razie)

Każdy chciałby zarysować kilka diagramów, nacisnąć przycisk i mieć działający software systemu, który spełnia nasze oczekiwania. Brzmi trochę jak magia, nieprawdaż? Być

może, ale jest to główna część wizji wykonywalnego UMLa (*Executable UML vision*). Podstawą tej idei jest użycie narzędzi CASE to przetworzenia szczegółowych diagramów UML i wzbogacenia ich w specyfikacje napisane w językach formalnych, najlepiej w OCL (*Object Constraint Language*) instytucji OMG (*Object Management Group*).

Główną ideą przyświecającą WUML jest to, że systemy mogą być modelowane na wyższym poziomie abstrakcji niż kod źródłowy, następnie symulacja dostarczałaby narzędzi do sprawdzania poprawności koncepcji, by później procesy translacji doprowadzały do efektywnego kodu docelowego. Ten wyższy poziom abstrakcji powinien pomóc uniknąć przedwczesnego projektowania, pozwalając na wprowadzanie „w biegu" wymaganych zmian i opóźniać implementacyjne decyzje prawie do końca procesu. Idea jest piękna, ale w rzeczywistości jest wiele problemów, które nie pozwalają na realizację tej idei:

- UML nie jest wystarczający.

UML nie jest wystarczająco dobrym narzędziem do modelowania aplikacji biznesowych, jak to już pokazano powyżej. Skutkuje to tym, że firmy software'owe specjalizujące się w wytwarzaniu specjalistycznych narzędzi programistycznych są zmuszone dodawać swoje produkty do narzędzi UML tworząc ich rozszerzenia. To nie jest WUML. To są wykonywalne modele i w takich razach powinno się mówić o wykonywalnym modelowaniu (xModeling).

- Integracja zbioru narzędzi wspomagających WUML jest trudna.

Załóżmy, że jeden lub więcej producentów software'u narzędziowego zdecyduje się implementować ideę WUML, aby wypełnić lukę w UML. Jak to powinno działać? Idealnie by było, gdyby pewne firmy skoncentrowały się ściśle na stworzeniu dobrych narzędzi modelowania, inne skierowałyby swoje wysiłki na stworzeniu dobrych wyjść z tych modeli, wytwarzających wykonywalny kod dla poszczególnych platform. Pewne firmy produkowałyby wtyczki dla J2EE, inne dla .NET, inne dla środowisk mainfrejmowych. Aby to osiągnąć potrzebny byłby wspólny standard dla współdzielenia informacji między tymi producentami.

- Pojedynczy producent sobie nie poradzi.

Inne podejście może polegać na tym, że pojedynczy producent będzie wykonywał zarówno narzędzia modelowania systemów, jak i narzędzia do generacji kodu z modeli. W teorii jest to podejście bardzo racjonalne. Ale ze względu na bardzo szeroki wachlarz platform programistycznych i sprzętowych i szereg opcji wewnątrz tych platform, pojedynczy wytwórca będzie się starał skoncentrować na jednej niszy.

Jeden może specjalizować się w produktach wspierających środowisko J2EE dla JSP (Java Sever Pages), serwetach, EJBs (Enterprise JavaBeans) i np. w relacyjnych bazach danych. Inny może skierować swoją uwagę i wysiłki dla stworzenia WUML dla „grubych" klientów Javy, inny zaś dla „grubych" klientów Win64 itd. Nie ma wątpliwości, że już w najbliższej przyszłości pojawią się narzędzia spełniające wizję WUML. Rzeczywiście, kilku

producentów narzędzi CASE osiągnęło już teraz niezłe rezultaty. Ale, jak twierdzi Scott W. Ambler (Agile 2015; Agile Alliance 2015; dinsboiv 2015), ta wizja „padnie" jak wiele innych z przeszłości. Zawsze będą potrzebni programiści.

Czy takie narzędzie może być „zwinne" (*agile*)? Prawdopodobnie.

Jeżeli jest możliwe wytworzenie oprogramowania narzędziowego, łatwego w użyciu, które generuje software wystarczająco dobry dla „przemysłowego" użycia i zapewnia lepszą „wydajność" zainwestowanych pieniędzy (maksymalizuje ROI – *Return On Investment*) – to autor jest „za". Jednak, czy należy spodziewać się, że jakieś tego typu narzędzie pojawi się wkrótce: autor mówi „nie!".

Modelowanie procesów biznesowych – BPM

Modelowanie procesów biznesowych BPM (*Business Process Modelling*) łączy strategie biznesowe z rozwojem systemów informatycznych w celu uzyskania maksymalnych korzyści operacyjnych. Połączenie spojrzenia procesowego (*workflow*), funkcjonalnego, organizacyjnego i danych/zasobów z odpowiednimi miernikami typu koszty czy czasy żywotności pozwala na zbudowanie podwalin do analizowania wąskich gardeł, krytycznych ścieżek i różnego rodzaju nieefektywności w działaniu. Jeśli BPM jest zrealizowany prawidłowo, to może także służyć jako prekursor modelowania/konstruowania aplikacji i przepływu czynności

Technologia informatyczną, która zdobywa sobie coraz więcej zwolenników/użytkowników jest BPMN (*Business Process Modelling Notation)*. Procesy biznesowe rozpatrywane bardziej z ludzkiego punktu widzenia niż z punktu widzenia informatyka, mogą być rozpatrywane, standaryzowane i modelowane przy użyciu BPMN. BPMN dostarcza wielu modeli, zaprojektowanych aby służyły ludziom, którzy projektują i zarządzają procesami biznesowymi (BPMN 2015; BPMN/2.0 2015). BPMN zapewnia narzędzia do mapowania modeli na wykonywalny kod w systemie BPM Systems (WSBPEL). Ponadto BPMN daje możliwość standaryzacji mechanizmów wizualizacji procesów biznesowych, które są następnie przetwarzane na wykonywalne kody programów komputerowych.

Technologia BPMN zapewnia (podobnie jak UML) zdolność zrozumienia wewnętrznych procedur biznesowych (wewnątrz organizacji) przy pomocy notacji graficznych (diagramów) i daje organizacjom możliwość komunikowania tych diagramów do otoczenia organizacji w sposób standardowy.

Obecnie istnieje mnóstwo metodologii i narzędzi modelowania procesów biznesowych. Zakładając np. że pracownicy będą przechodzić z jednej firmy do drugiej, firmy będą się łączyć czy też dzielić – jest bardzo prawdopodobne, że analitycy biznesowi muszą rozumieć wielorakie formy procesów biznesowych. Procesy te będą miały różne formy w zależności od tego, w którym cyklu życia się znajdują: w projektowaniu, implementacji, wykonaniu, sprawdzaniu czy też analizie.

Zatem standardowa graficzna notacja będzie bardzo pomocna w zrozumieniu działania i współpracy transakcji biznesowych zarówno wewnątrz organizacji, jak i między organizacjami. Zapewni to wzajemne zrozumienie uczestników procesów biznesowych i umożliwi szybkie dostosowanie się do zmian zachodzących w biznesowym środowisku (także B2B). BPMN jest wierne tradycji notacji wykorzystywanej w schematach przepływowych (*flowcharting*) ze względu na ich czytelność i elastyczność. Do modelowania procesów biznesowych organizacja OMG wykorzystuje doświadczenia notacji poprzedzające BPMN ze względu na kombinację ich czytelności, elastyczności i rozszerzalność.

BPMN rozszerza również tradycyjne notacje modelujące procesy biznesowe przez spójne traktowanie koncepcji procesów biznesowych dotyczących transakcji typu B2B, takich jak publiczne i prywatne procesy i choreografie (*Choreographies*). Podobnie uwzględnia on (BPMN) zaawansowane koncepcje dotyczące modelowania, takie jak obsługa wyjątków, transakcji i kompensacji.

Podsumowanie

Inżynieria oprogramowania jest dziedziną, której zadaniem jest stworzenie narzędzi umożliwiających przetworzenie procesów twórczych, jakimi są projektowanie i konstruowanie systemów informatycznych, siłą rzeczy bardzo subiektywnych, w miarę obiektywny ciąg działań prowadzących do stworzenia poprawnego komercyjnego produktu. W artykule autor starał się przedstawić metodologie (narzędzia) historyczne już (choć jest wielu twórców świadczących, że metodologie strukturalne jeszcze nie umarły), aktualne (metodologie obiektowe) i słabo jeszcze rozpropagowane metodologie oparte na modelowaniu procesów biznesowych (BPMN) w sposób jak najbardziej naturalny. Naczelnym celem opracowywanych metodyk jest stworzenie narzędzi, przy pomocy których będzie można doprowadzić do bezpośredniej translacji modelu w wykonywalny kod programu komputerowego.

Bibliografia

AgileAlliance (2015), http://www.AgileAlliance.org [dostęp: 25.12.2015].

Booch G. (1994), *Object Oriented Analysis and Design with Applications* (2nd Edition), Benjamin/Cummings.

BPMN (2015), http://www.omg.org/bpmn/index.htm [dostęp: 25.12.2015].

BPMN 2.0 (2015), http://www.omg.org/spec/BPMN/2.0. [dostęp: 25.12.2015].

Coad P., Yourdon E. (1989), *Object-Oriented Analysis*, Yourdon Press.

Coad P., Yourdon E. (1991), *Object-Oriented Design*, Yourdon Press.

Dahl O., Dijkstra E.W., Hoare C.A. (1972), *Structured Programming*, Academic Press, London.

Dijkstra E.W. (1968), Letters to the editor: go to statement considered harmful, "Communications of the ACM", No. 11(3).

dimsboiv (2015), http://dimsboiv.uqac.ca/8INF851/web/part1/introduction/The_Agile_Manifesto.pdf [dostęp: 25.12.2015].

Knuth D. (1968), *The Art of Computer Programming*, Vol.1, Addison-Wesley, London.

Knuth D., Floyd R. (1971), *Notes on avoiding 'go to' statements*, "Information Processing Letters", No. 1, February.

Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W. (1991), *Object-Oriented Modeling and Design*, Prentice-Hall, New York.

Unold O. (2015), http://www.zio.iiar.pwr.wroc.pl/ [dostęp: 28.12.2015].

Wirfs-Brock R., Wilkerson B., Wiener R. (1990), *Designing Object-Oriented Software*, Prentice-Hall, New York.

http://www.software.com.pl/konferencje/sperspective [dostęp: 28.12.2015].

Software Engineering. Review of Methodologies

Summary

The author made a short review of some software engineering methodologies. Evaluation of strengths, weaknesses, and limitations of these methodologies is made in this paper. The author brings forward new perspectives in this area: the BPMN (Business Process Modelling Notation) methodology. In point 2, there are shown some reasons for creating particular tools in the software engineering area of interest. Points 3-5 contain a short description of three of them: structured approach, object approach and BPMN. In the conclusion part, the author made comparisons and evaluated usefulness of particular methodologies.

Key words: software engineering, methodologies, Business Process Model and Notation (BPMN).

JEL codes: L86

Artykuł nadesłany do redakcji w styczniu 2016 roku.

© All rights reserved

Afiliacja:
dr Benedykt Burgielski
Akademia Finansów i Biznesu Vistula
Wydział Inżynierski
ul. Stokłosy 3
02-787 Warszawa
tel.: 22 457 23 00
e-mail: b.burgielski@vistula.edu.pl