Podstawy UML 2.0

Wstęp

Ten krótki artykuł jest wstępem do UML 2.0 – metodologii[1] stosowanej na co dzień w fazie projektowej podczas produkcji oprogramowania. Tekst porusza te zagadnienia związane z UMLem, które są szczególnie istotne i przydatne w fazie tworzenia dokumentu technicznego gier komputerowych. Stąd wiele jest w tekście uproszczeń, nagięć języka, tak aby jak najlepiej służył celom programistów gier. Ponadto z bogatej kolekcji 13 diagramów zdefiniowanych w standardzie UML 2.0 [1] tekst porusza tylko te, które znajdują szersze zastosowanie w projektowaniu poszczególnych części gry bądź jej rdzenia.

Chciałbym także zaznaczyć, że często będę używał terminu obiekt jako polskiego odpowiednika słowa 'item'. W naszym rodzimym języku jest to najlepiej pasujący termin, który niestety koliduje z innym określeniem związanym z programowaniem. Jeśli nie będzie to oczywiste, postaram się to zastosowanie podkreślić. Ponadto trochę z przyzwyczajenia w znacznej części przykładów stosuję angielskie nazwy dla klas (a także ich składowych, obiektów, aktorów itd.). Są to jednak słowa na tyle elementarne, że ich znalezienie w słowniku nie będzie stanowić problemu dla osób, które ich nie znają.

Czym jest UML?

UML (Unified Modeling Language[2]) jest to sposób formalnego opisu modeli reprezentujących projekty informatyczne (zazwyczaj te o charakterze programistycznym, choć i od tej reguły zdarzają się wyjątki). Wysoki poziom abstrakcji gwarantuje całkowitą niezależność od platformy i języka. Obecnie obowiązujący standard (2.1), będący twórczym rozwinięciem UML 1.x (a także stosunkowo młodego 2.0) definiuje 13 różnych diagramów (podzielonych na dwie kategorie[3]: strukturalne i opisujące zachowania) o różnym przeznaczeniu. Każdy z nich opisuje dany system pod innym kątem, z innej perspektywy, a nawet na różnym poziomie abstrakcji. Dla nas z praktycznego punktu widzenia znaczenie będą miały jedynie następujące spośród nich wszystkich:

- Diagram klas (jeden z najważniejszych diagramów strukturalnych, zwany też czasem po prostu diagramem strukturalnym)
- Diagram obiektów
- Diagram przypadków użycia[4]
- Diagram aktywności
- Diagram Automatu Stanów
- Diagram sekwencyjny

---

[1] Użycie słowa metodologia w tym miejscu jest sprzeczne ze stanowiskiem Object Management Group (w skrócie OMG) – twórców standardu UML. Według nich UML jest językiem opisu projektów, natomiast faktyczne metodologie są stosowanym w fazie projektu i fazach ją poprzedzających sposobem/podejściem do gromadzenia, analizowania wymagań stawianych przed systemem informatycznym. Choć rzeczywiście jest to prawda, taka definicja jest bardziej związana z informatyką powiązaną z wielkim biznesem niż z tworzeniem gier.

[2] Nazwa ta wbrew pozorom nie zawiera błędu – pochodzi po prostu z US English, gdzie końcowe litery wyrazów nie ulegają podwojeniu.

[3] Zwyczajowo mówi się o dwóch wymienionych kategoriach, ale standard języka wyraźnie wydziela jeszcze trzecią: diagramy interakcji. Z omawianych w tym artykule należy do nich jedynie diagram sekwencyjny.

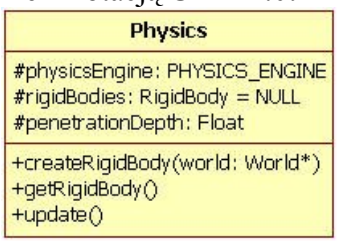
[4] Z ang. use-case. Angielski termin występuje też w polskiej literaturze i terminologii i dlatego go tutaj podaję. Przypadkiem użycia jest dobrze określona bądź nazwana interakcja pomiędzy aktorem i systemem.

Największą zaletą UMLa i w ogóle koncepcji projektowania, jest olbrzymia oszczędność czasu w późniejszej fazie produkcji oraz zmniejszenie ryzyka konieczności przepisywania kodu źródłowego na nowo. Choć to zdanie wielu osobom z pewnością wyda się być truizmem, rzesze programistów dzień w dzień zdają się o nim zapominać. Trzeba więc o tym przy każdej okazji przypominać.
Przewagą UMLa nad innymi formalnymi językami opisu projektów informatycznych jest to, że stworzony model[5] jest niezwykle przyjazny dla programisty – nie dość, że model zwykle ukazuje pewien program w prosty i czytelny sposób, to na dodatek nie jest wymagana duża dodatkowa wiedza, żeby diagramy modelu zrozumieć czy nawet rozszerzyć (w przypadku metodologii takich jak np. PRINCE2 czy COBIT[6] tak łatwo już nie jest). Oczywiście nie dotyczy to wszystkich modeli. UML choć z pozoru prosty, może się bowiem znacznie skomplikować, gdy zaczniemy się zagłębiać w jego szczegóły. Jednak z punktu widzenia programisty (a nie typowego architekta czy projektanta) taka wiedza nie zawsze jest potrzebna.
W dalszej części znajduje się dokładniejszy opis każdego z wymienionych diagramów wraz ze stosownymi przykładami [3]. Chciałbym zwrócić uwagę na fakt, że zamieszczony tu opis to wielki skrót, który jednak powinien całkowicie wystarczyć dla naszych celów.

Opisy Diagramów

Diagram klas (diagram strukturalny)

Diagram ten będący podstawowym i chyba najczęściej wykorzystywanym diagramem strukturalnym, ukazuje wzajemne powiązania między klasami tworzącymi dany system. Nie ukazuje on jednak żadnych relacji pomiędzy samymi obiektami (za to odpowiada diagram obiektów).
Zatrzymajmy się na chwilę w tym miejscu i powiedzmy sobie kilka słów o podstawowej notacji obowiązującej dla tego diagramu. Klasy w zapisie UML 2.0 reprezentowane są przez prostokąty podzielone w poziomie na 3 części. W górnej z nich, po środku znajduje się nazwa klasy[7], w środkowej z wyrównaniem do lewej – składowe zmienne klasy, a w dolnej - również z wyrównaniem do lewej strony – składowe funkcje klasy. Poniżej znajduje się przykładowa klasa, zapisana zgodnie z notacją UML 2.0:



Omówimy teraz pokrótce szczegóły dotyczące powyższego zapisu.
Składowe zmienne i funkcje zapisywane są w następujący sposób:

- zmienne: [zasięg][8] nazwa [: typ] [= wartość początkowa]
- funkcje: [zasięg] nazwa([typ argumentu]) [: typ zwracany]

---

[5] Terminu model używam w tym tekście z dużą dowolnością, nie wprowadzając żadnej formalnej definicji. Należy być jednak świadomym, że pojęcie to jest dość precyzyjnie określone przez standard UMLa. Poziom komplikacji tej definicji jest jednak na tyle duży, że zdecydowałem się nie zagłębiać w szczegóły i pozostać przy definicji intuicyjnej.
[6] Metodologie te zajmują się jednak bardziej sferą zarządzania projektem niż jego faktyczną implementacją.
[7] Po nazwie klasy może się w górnej części pojawić dodatkowo tzw. stereotyp zapisywany w następujący sposób: << nazwa stereotypu >>. Stereotypy to wysokopoziomowe kategorie, typy lub znaczenia konkretnego obiektu (stereotypy nie dotyczą wyłącznie klas) [2]
[8] Nawiasy kwadratowe sygnalizują, że ich zawartość jest opcjonalna.

Gdzie:

- zasięg określany jest w jeden z następujących sposobów:
  - + - publiczny
  - # - chroniony
  - - - prywatny

  Należy w tym miejscu wspomnieć, że czasem zamiast powyższych symboli stosuje się zapis za pomocą odpowiednich ikon. Jednak, taki zapis można spotkać jedynie w oprogramowaniu wspomagającym projektowanie w UMLu, gdyż rysowanie skomplikowanych kształtów na papierze (od czego zwykle wszystko się zaczyna) jest raczej niepraktyczne (nie mówiąc o tym, że takie rysunki mogą być mniej czytelne od znaków ASCII).
- typ, typ argumentu oraz typ zwracany – nazwa typu (może to być albo typ prosty, albo nazwa innej klasy istniejącej w ramach danego systemu). UML 2.0 definiuje następujące typy proste:
  - Integer – liczby całkowite (odpowiednik z C++: int)
  - Float – liczby zmiennoprzecinkowe (odpowiednik z C++: float, double)
  - String – łańcuchy znaków (odpowiednik z C++: char*)
  - Void – typ nieokreślony (odpowiednik z C++: void)
- wartość początkowa – wartość ustawiana dla zmiennej w fazie jej inicjowania.

Kilka dodatkowych przykładów:

- #age: Integer – prywatna zmienna typu całkowitego
- +text: String = „Hello world” – publiczna zmienna typy String (ciąg znakowy)
- +squareRoot(Float): Float – publiczna funkcja przyjmująca argument typu Float i zwracająca liczbę tego samego typu.

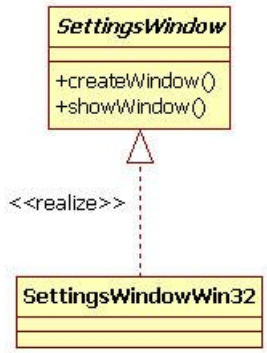
Ponadto chciałbym zwrócić uwagę na fakt, że w diagramach klas zwykło się nie umieszczać następujących elementów:

- Konstruktory i destruktory
- Akcesory (funkcje postaci set...() i get...())
- Elementy bibliotek pochodzących spoza projektu, w tym elementy pochodzące z biblioteki standardowej danego języka (w przypadku STL – dotyczy to np. kontenerów, iteratorów, itd.)
- Wskaźniki

Niemniej czasem istnieją powody na tyle istotne, że od tej niepisanej reguły się odstępuje, co zresztą widać na niektórych zamieszczonych w tym tekście przykładach.

Relacje

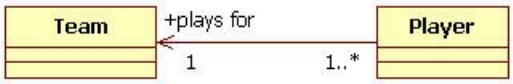
Pojedyncze klasy nie mają praktycznie żadnej wartości. Dopiero współpraca kilku lub większej liczby klas ma rzeczywiste znaczenie. Współpraca klas jest definiowana poprzez tzw. relacje, reprezentowane przez linie (krawędzie) z odpowiednim zakończeniem, które dalej będę nazywał grotami (choć nie jest to nazwa formalna ani poprawna, pozwoli na utożsamienie relacji ze strzałkami). Grot wskazuje zawsze tę klasę, która jest w danej relacji ważniejsza. Można powiedzieć, że klasa 2. świadczy na rzecz klasy B pewne usługi. Poniżej znajdują się 2 klasy będące w relacji:



W tym konkretnym przypadku powiemy, że klasa SettingsWindow (na którą grot wskazuje) jest rodzicem klasy SettingsWindowWin32[9]. Dokładniejszy opis tej i pozostałych relacji znajduje się poniżej, więc tu pominę dokładniejsze wytłumaczenie.

Powiązanie

Dwie klasy są powiązane jeśli są w relacji, do której określenia pozostałe typy relacji UMLa są niewystarczające. Relację tę można sprecyzować za pomocą krótkiego opisu słownego, umieszczanego powyżej linii reprezentującej relację. Opis ten ogranicza się zwykle do kilku wyrazów, z których conajmniej jeden zwykle nazywa czynność. Ważne jest też to, że przy tego typu relacji linia może być pozbawiona „grotu". Wówczas jest to relacja dwustronna.



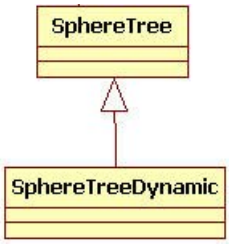
Powyższa relacja oznacza, że obiekty klasy Player grają dla klasy Team. Innymi słowy klasa Player świadczy usługę grania na rzecz klasy Team. W tym konkretnym przypadku relacja mogłaby być pozbawiona grotu, jeśli po drugiej stronie widniałby napis np. „is being paid by". Relacja byłaby wtedy dwukierunkowa. Chciałbym zwrócić jeszcze uwagę na liczby poniżej linii definiującej relację. Ich opisem zajmiemy się dopiero za chwilę, ale muszę się upewnić, że nie przeoczysz żadnego szczegółu.

Generalizacja

Generalizacja reprezentuje dziedziczenie z OOP[10]. Na rzecz tego tekstu należy powiedzieć, że jeśli klasa B jest w relacji generalizacji z klasą Ą klasą podrzędną, to klasa B ma wszystkie składowe klasy A oraz potencjalnie swoje unikalne. Klasa A jest klasą rodzicielską, a klasa B potomną. Przykładem – rzekłbym szkolnym - mogą być klasy: Animal i Bird. Bird będzie oczywiście w relacji generalizacji klasą podrzędną, gdyż zawiera wszystkie cechy zwykłego zwierzęcia (np. zdolność poruszania się, brak umiejętności przeprowadzania procesu fotosyntezy) i dodatkowo definiuje własne, unikalne zachowania i cechy (np. skrzydła, latanie).

[9] W rzeczywistości można też powiedzieć, że klasa SettingsWindow jest super klasą, bądź klasą nadrzędną klasy SettingsWindowWin32. Ponadto w tym konkretnym przypadku należałoby bardziej skupić się na polimorfizmie, ale są to zagadnienia związane z OOP i tłumaczenie ich w tym miejscu mija się z celem.

[10] Z ang. Object Oriented Programming, czyli programowanie zorientowane obiektowo. W OOP istnieje wysoki poziom abstrakcji, który reprezentowany jest przez tzw. obiekty. OOP jest to jeden z najpopularniejszych obecnie sposobów programowania systemów informatycznych i jeden z fundamentów, na którym oparty jest UML.



Jeśli strzałka jest rysowana linią przerywaną a nie ciągłą, zamiast o generalizacji, możemy powiedzieć o realizacji, co jednak z punktu widzenia implementacji, często nie ma aż tak istotnego znaczenia. Sama realizacja jest omówiona w dalszej części.

Zawieranie

Jeśli mówimy, że klasa A jest zawarta w klasie B, to znaczy, że klasa A jest jednym z elementów klasy B, np. klasa Zeszyt może być zawarta w klasie Plecak. Niemniej zawieranie mówi o tym, że zarówno klasa A jak i klasa B mogą istnieć jako oddzielne elementy. W naszym przypadku jest to prawda, zarówno Zeszyt jak i Plecak są elementami nie uzależnionymi od siebie.

Kompozycja

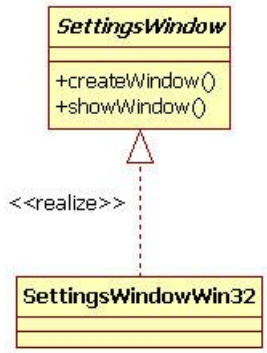
Jest szczególnym (skrajnym) przypadkiem zawierania. Różnica polega na tym, że klasa B, która jest zawarta w A, nie może posiadać samodzielnych instancji. Innymi słowy zawsze jest składową obiektów klasy B, np. Drzwi samochodowe nie mogą istnieć jako samodzielne obiekty. Zawsze są skomponowane z obiektami klasy Samochód. Różnica w notacji w stosunku do zawierania polega na tym, że romb na zakończeniu krawędzi reprezentującej relację jest wypełniony. Rysunek zatem w tym przypadku pominiemy.

Zależność

Ten rodzaj relacji jest zwykle stosowany w początkowej fazie projektowania systemu, ze względu na fakt, że jest to relacja bardzo ogólna, stosowana do opisania wielu rodzajów powiązań. Innymi słowy, jeśli wiadomo, że dwie klasy są w relacji, ale początkowo nie jesteśmy w stanie określić w jakiej dokładnie, wówczas należy skorzystać z relacji zależności, a w późniejszej fazie projektowania zastąpić ją relacją bardziej precyzyjną. Relację tę reprezentuje zwykła linia pozbawiona jakichkolwiek zakończeń.

Realizacja

Klasa B realizuje klasę A, wtedy gdy implementuje jej wszystkie metody. Zazwyczaj klasa A jest interfejsem. W tym przypadku relacja jest dodatkowo opisana przez stereotyp (dość podobnym znaczeniowo stereotypem jest <<implements>>).



Opis relacji

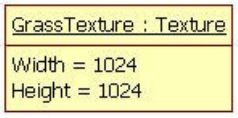
Okazuje się, że rysowanie samych tylko strzałek z różnymi grotami, jest często niewystarczające, po prostu zbyt mało precyzyjne. Aby lepiej opisać relację, można (i często należy) stosować jedną z poniższych technik:

- określić stereotyp relacji – stereotyp relacji jest to jak gdyby zachowanie się relacji, swoiste doprecyzowanie zdefiniowane w UMLu. Stereotypy dotyczą też innych obiektów UMLa – np. klas. O czym już zresztą wspomniałem.
- umieścić krótki opis słowny powyżej linii reprezentującej relację, np. wspomniane już „plays for".
- określić pomiędzy iloma obiektami zachodzi dana relacja. Szczególne znaczenie w modelach mają następujące wartości:
  - 0
  - 1
  - 0..* (od zera do nieskończoności)
  - 1..* (od jednego do nieskończoności)
  - * (dowolna liczba)

  Numer znajdujący się w najbliższym sąsiedztwie danej klasy mówi, ile obiektów tej klasy uczestniczy w relacji. W przykładzie z klasami Team i Player mamy więc, że każdy zawodnik gra dla dokładnie 1 drużyny, a drużyna ma conajmniej 1 zawodnika.

Diagram obiektów

Tak jak diagram klas opisuje klasy i powiązania między nimi, tak diagram obiektów zajmuje się obiektami z OOP. Choć diagram obiektów można często traktować jako uszczegółowienie diagramu klas, należy pamiętać, że nie jest jedno i to samo. Diagram obiektów ukazuje role spełniane przez poszczególne instancje danej klasy (klas). Różna jest także notacja. W diagramie obiektów, obiekt ukazywany jest jako prostokąt (tym razem nie podzielony na żadne części), w którym umieszczona jest jego nazwa pisana z podkreśleniem. Po dwukropku po nazwie, można podać też nazwę klasy, którą dany obiekt reprezentuje.
Dla obiektu możliwe jest także określenie jego stanu działania (run-time state), czyli ustalenie wartości jakie przyjmą składowe klasy obiektu w trakcie działania systemu.



Powyższy przykład to obiekt klasy reprezentującej teksturę trawy, czyli pojedynczą teksturę używaną w świecie gry. Width i Height i wartości im towarzyszące są zaś wspomnianym stanem działania.

Diagram przypadków użycia

Diagram przypadków użycia jest diagramem istotnym we wstępnej fazie projektowania. Prezentuje bowiem wymagania stawiane przed tworzonym systemem (lub dowolnym jego podsystemem). Można go użyć jako pomoc przy specyfikacji wymagań stawianych przed systemem informatycznym. Diagram ten prezentuje interakcje zachodzące pomiędzy systemem a zewnętrznymi jednostkami.
W diagramie użycia jednym z ważniejszych obiektów (tu nie chodzi mi jednak o obiekt w sensie OOP) jest tzw. Aktor. Aktorem nazywamy taki obiekt, który prowadzi interakcję z systemem jednak sam do niego nie należy. Przykładem Aktora może być Klient[11] wchodzący do systemu Sklep i prowadzący z tym systemem pewną interakcję, np. Kup. Aktorzy mogą być z innymi aktorami w takich samych relacjach jak klasy (również notacja nie ulega zmianom). Sposobem reprezentacji Aktorów jest człowieczek złożony z odcinków z dużą głową:

Innym ważnym pojęciem są tak zwane przypadki użycia (w zasadzie jest to sedno tego diagramu). Jest to element reprezentujący jakąś konkretną pracę, interakcję reprezentowaną bez przedstawienia szczegółów. Dla przykładu ze sklepem przypadkiem użycia mogłaby być Zakup. Przypadki użycia są prezentowane przy pomocy owali, w środku których jest wyrażenie odczasownikowe. Podobnie jak klasy i aktorzy Przypadki Użycia mogą być z innymi obiektami w relacjach. Inne właściwości Przypadków Użycia pomijamy. Poniżej znajduje się przykład przypadków użycia systemu Player (który może być utożsamiany z postacią gracza):

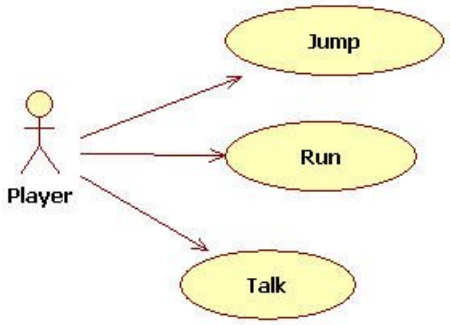
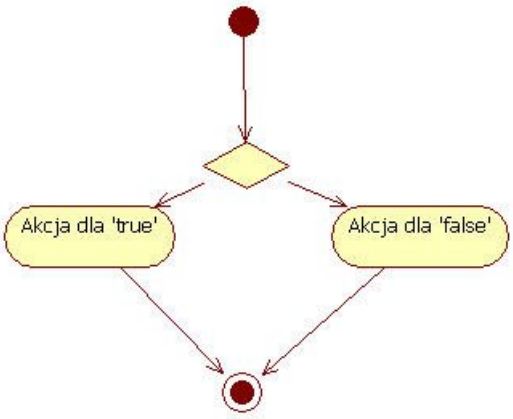


Diagram aktywności

Diagram aktywności to nic innego jak sieć działań danego podsystemu. Jedyna poważniejsza z naszego punktu widzenia różnica między nim a typowym schematem blokowym to inna forma notacji. Istnieją co prawda szczegóły będące jeszcze innymi różnicami, ale na razie je pomijam.

[11] Klient, rozumiany jako Client to w programowaniu nie tylko człowiek, ale także inny system informatyczny, urządzenie bądź nawet klasa. Jest więc to podsystem na rzecz, którego wykonywane są pewne czynności.



Początek schematu oznaczany jest przez wypełnione koło, a koniec przez dwa koła zawarte jedno w drugim. Operacje (tu zwane akcjami) oznaczamy przez prostokąty o mocno zaokrąglonych rogach, a instrukcje warunkowe standardowo przez równoległoboki. Obok strzałek, informujących o kierunku przepływu działania systemu, można oczywiście umieszczać opisy. Oczywiście diagram ten ma dodatkowe możliwości, jednak dla nas nie mają one na razie większego znaczenia.

Diagram Automatu Stanów

Automat Stanów, to schemat blokowy, o ustalonym początku i końcu (nazywanych odpowiednio stanem początkowym i końcowym) oraz pewną skończoną liczbą stanów pośrednich pomiędzy nimi. Diagram Automatu Stanów opisuje zachowanie (czy sekwencję zachowań) konkretnego obiektu, a nie jak w przypadku diagramu aktywności – całego systemu lub pewnego jego podsystemu. Sam Stan to status obiektu w danym momencie, lub aktualnie wykonywana przez niego czynność. Każdy Stan jest połączony przynajmniej z jednym innym w taki sposób, że zawsze istnieje droga ze stanu początkowego do końcowego.
Każdy stan jest prezentowany jako prostokąt z zaokrąglonymi rogami i nazwą stanu w jego górnej części, po środku. Dwa stany są ze sobą połączone za pomocą strzałki, której grot wskazuje kierunek przechodzenia przez stany. Nad nią może się też znajdować krótki opis czynności będącej przyczyną zmiany stanu. Jeśli takiego opisu brakuje, oznacza to, że zmiana zachodzi w sposób samoistny bez względu na status systemu.
Czasem przydatną możliwością jest określenie działań systemu przy wejściu i opuszczeniu danego stanu. Jeśli taka konieczność zachodzi, w dolnej części prostokąta stanu wpisuje się:

- On Entry / [działanie]
- On Exit / [działanie]

Stan może przejść z siebie samego do siebie samego.
Poniżej znajduje się przykład prezentujący stan psychiczny potwora (NPC) w zależności od kilku działań gracza. Chciałbym zwrócić uwagę, że sposób zapisu stanów początkowego i końcowego jest identyczny jak w diagramie aktywności.

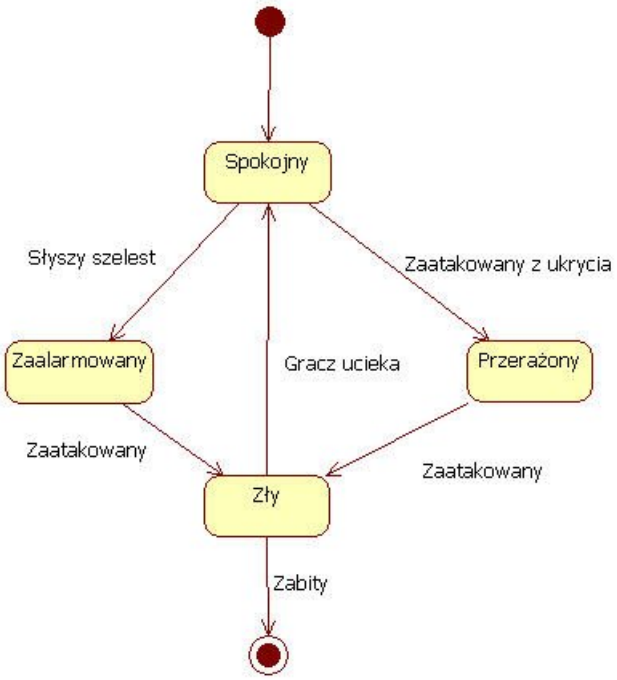
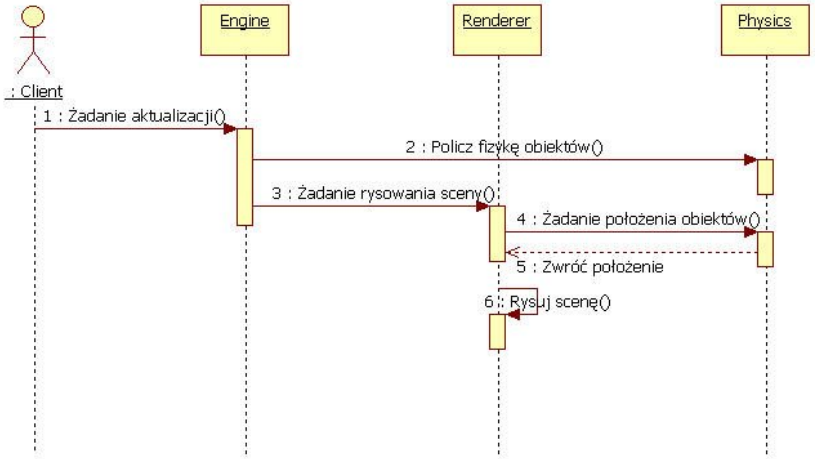


Diagram sekwencyjny

Diagram sekwencyjny ukazuje współpracę między obiektami (czyli tym razem już nie na poziomie klas) i przesyłane między nimi komunikaty. Możliwe jest też użycie w przypadku tego diagramu Aktorów. Obiekt jest prezentowany w sposób standardowy, tj. jako prostokąt z nazwą obiektu i ewentualnie nazwą jego klasy: np. Obiekt: B to obiekt klasy B.
Bez względu na użyte figury, prezentuje się je w sposób tradycyjny, przy czym leżą one w jednej linii (w poziomie) i od każdego z nich odchodzi w dół pionowa przerywana linia.
Te przerywane linie są ze sobą łączone strzałkami, powyżej których umieszcza się nazwę komunikatu przesyłanego między poszczególnymi obiektami. Komunikat można przesyłać między dowolnymi dwoma zdefiniowanymi na diagramie obiektami – nawet do obiektu, który komunikat wysłał. Możliwe jest też wysyłanie komunikatu od obiektu do siebie samego. Przesyłanie komunikatu jest zaznaczane za pomocą strzałki, powyżej której znajduje się krótki opis bądź nazwa komunikatu. Jeśli strzałka jest rysowana za pomocą linii przerywanej, oznacza to że dochodzi do wysłania (send) bądź zwrócenia (return) pewnych danych. Poszczególne komunikaty są numerowane począwszy od indeksu 1 dla komunikatu umieszczonego najwyżej w diagramie. Indeksy rosną wraz ze schodzeniem w dół diagramu.



Oprogramowanie

Zaletą UMLa jest to, że bez trudu da się zaprojektować podsystemy przy pomocy kartki i długopisu (lub ołówka – wówczas łatwiej o poprawianie błędów). Jednak zwykle i tak należy ostatecznie skorzystać z oprogramowania komputerowego. Niestety brak na rynku dobrych darmowych narzędzi zgodnych ze standardem UML 2.0 (nie wspominając o wersji 2.1), a cena komercyjnych pakietów zwykle przekracza możliwości pojedynczego programisty (czy nawet niewielkich zespołów developerskich). Ja osobiście korzystam z freeware'owego StarUML, mimo że nie jest w pełni zgodny z najnowszą specyfikacją i choć ciągle pełen jest drobnych błędów, lepszego darmowego programu do tworzenia modeli w UML jeszcze nie znalazłem. Dobrym narzędziem jest Poseidon – niestety jest to produkt w pełni komercyjny – istnieje jednak wersja Trial pozwalająca na darmowe przetestowanie jego możliwości. Z innych narzędzi należy również wymienić olbrzymi pakiet korporacji Rational, czyli Rose. Jednak jego zaporowa cena i prawdopodobnie brak wsparcia w przyszłości ze strony jego producenta nie czyni z niego najlepszego kandydata.

Zakończenie

Poruszone w tym tekście zagadnienia to jak wielokrotnie podkreślałem jedynie wierzchołek góry lodowej. Zajęliśmy się tylko najistotniejszymi aspektami UMLa 2.0, a nawet je omówiliśmy nad wyraz pobieżnie. Ponadto nie wspomnieliśmy w ogóle o różnych modelach, a nawet nie przedstawiliśmy precyzyjnej definicji samego pojęcia modelu. Jednak jak już mówiłem wiedza ta nie jest niezbędna do tworzenia pełnoprawnych projektów z wykorzystaniem tego języka. Zainteresowani tematem z pewnością zechcą sami rozwinąć swą wiedzę w tym zakresie.

Bibliografia

[1] http://www.uml.org/
[2] http://en.wikipedia.org/wiki/Stereotype_(computing)
[3] http://www.sparxsystems.com.au/resources/uml2_tutorial/index.html