Ramy i ich zastosowanie do enkapsulacji interakcji między procesami *

Agnieszka Gąsiorowska Aleksander Laurentowski
Akademia Górniczo-Hutnicza im. S. Staszica w Krakowie

maj 1994

1 Wstęp

Przełom lat 80–tych i 90–tych w informatyce cechował się dużym wzrostem wymagań wobec oprogramowania użytkowego. Wymagania te obejmowały zarówno stopień funkcjonalności konstruowanych programów, tj. ilość dostarczanych przez nie opcji i funkcji, jak również ich poprawność, niezawodność, wysoką jakość i wygodę kontaktu z użytkownikiem. Pakiety programowe mają dziś rozmiary niewyobrażalne jeszcze kilka lat temu – dziesiątki i setki megabajtów kodu. Tymczasem wydajność pracy programistów przy tradycyjnych metodach programowania jest bardzo ograniczona. Według Jonesa [5] ilość linii kodu źródłowego, którą jest w stanie napisać, przetestować i udokumentować w ciągu jednego dnia programista posługujący się tradycyjnymi metodami programowania jest zaledwie rzędu kilku (w przypadku projektów przekraczających 16 tys. linii, a obecnie większość z nich liczy się w setkach tysięcy). Wynika to z ograniczonych możliwości ludzkiego umysłu – pojedynczy człowiek nie jest w stanie objąć całkowitą kontrolą systemu tej wielkości.

Rozwiązanie zaistniałej sytuacji mogą przynieść nowe metody inżynierii oprogramowania. Jedną z najważniejszych wśród nich jest idea wielokrotnego użycia raz opracowanych składowych programu (ang. *software reuse*).

Celem niniejszego artykułu jest prezentacja jednej z realizacji tej idei: pojęcia ramy (ang. *framework*) jako uniwersalnego komponentu programowego wielokrotnego wykorzystania, cechującego się znaczną elastycznością oraz zdolnością dostosowania do różnych warunków i wymagań. Rama, w zależności od rodzaju, stanowi szkielet bądź część systemu końcowego tworzonego z jej pomocą. Zawsze jednak ramy zawierają i ukrywają (inaczej mówiąc: enkapsulują) pewien schemat funkcjonalny lub strukturalny istniejący w tych systemach. Ich zastosowanie może znacznie uprościć i przyspieszyć konstrukcję oprogramowania, obniżając jednocześnie jej koszt.

Przedstawimy również metodę kostrukcji ram dla schematów interakcji w programach rozproszonych za pomocą mechanizmów programowania obiektowego.

Czujemy się w obowiązku zaznaczyć, że ramy (ang. *frameworks*), o których piszemy, nie mają nic wspólnego z ramami (ang. *frames*), służącymi do reprezentacji wiedzy w systemach ekspertowych [6].

*Pracę wykonano częściowo w ramach grantu KBN nr 8 S503 015 06.

2 Propozycja systematyzacji modeli ram

Istnieje szereg podejść do pojęcia ramy jako jednostki wspomagającej wielokrotne wykorzystanie oprogramowania (*software reuse*). Jednym z przykładów jest *Choices* [2], rama-wzorzec do tworzenia systemów operacyjnych na różne architektury sprzętowe. Składa się ona w zasadniczej części z ok. 300 klas abstrakcyjnych w C++, ukonkretnianych w zależności od wymagań sprzętowych i koncepcyjnych. Tak rozumiana rama jest modelem obrazującym architekturę systemu obiektowo zorientowanego. Z kolei deklaratywny język programowania równoległego PCN [3] umożliwia tworzenie innego typu ram: tzw. komórek (ang. *cells*) i szablonów (ang. *templates*). Komponenty te odwzorowują zadany algorytmem schemat interakcji między procesami na odpowiednią topologię wirtualną procesorów. Jeszcze innym typem ramy jest medium komunikacyjne, zapewniające współdziałanie niezależnych aplikacji. Przykładem jest tu ToolTalk firmy SunSoft, moduł zapewniający aplikacjom automatyczny dostęp do wszelkiego typu danych numerycznych, tekstowych, graficznych, akustycznych, itp. bez konieczności posiadania wiedzy o miejscu i postaci przechowywania tej informacji w systemie, lub bardziej znany mechanizm OLE z Microsoft Windows.

Podejścia te różnią się w wielu aspektach, posiadają jednak naszym zdaniem pewną dozę cech wspólnych. Nie zaproponowano do tej pory uniwersalnej definicji ramy, choć pojęcie to jest często używane przy opisie poszczególnych realizacji projektów z zakresu inżynierii oprogramowania. Nie podano również kryteriów systematyzacji już istniejących implementacji. Wynika to zapewne z faktu, iż problematyka ta jest rozwijana od niedawna i nie osiągneła jeszcze wystarczającej dojrzałości. Jesteśmy przekonani, że próba ogólnego opisu i klasyfikacji ram może nie tylko pomóc w uchwyceniu samej istoty zagadnienia, ale także mieć pozytywny wpływ na rozwój tej dyscypliny.

W punkcie tym przedstawimy zatem propozycję definicji uogólnienia ram oraz kryteria, jakie można stosować do ich klasyfikacji. Pokażemy przy tym, jak za pomocą tych kryteriów można uszeregować wymienione podejścia.

2.1 Ogólna charakterystyka pojęcia rama

Rama jest komponentem programowym, posiadającym następujące podstawowe cechy:

1. Może być **powtórnie użyty** w innym programie.
2. Reprezentuje pewien **schemat** funkcjonalny lub strukturalny (a czasem oba z nich).
3. Jest **uniwersalny i elastyczny**, umożliwiając budowę programów różnorodnych, ale zachowujących ten sam schemat funkcjonalności lub struktury (patrz punkt poprzedni).
4. Jest jednostką **samodzielną** pod względem funkcjonalnym, tzn. posiada ściśle określony zestaw funkcji, który samodzielnie realizuje.
5. Jest **mechanizmem enkapsulacji**, ponieważ sposób realizacji funkcji dostarczanych przez ramę jest z reguły niewidoczny dla ich użytkownika.

Rama powinna być maksymalnie uniwersalna i elastyczna.

Elastyczność ramy oznacza, że możliwe jest dostosowywanie konstruowanych z jej pomocą systemów do wymagań środowiska, ukonkretnianie zawartych w niej założeń abstrakcyjnych, redefiniowanie składników konkretnych, a czasem nawet ingerencja w jej wewnętrzną strukturę. Odróżnia to ramę od procedury bibliotecznej, która jest dobrze znanym i powszechnie stosowanym narzędziem wielokrotnego wykorzystywania oprogramowania. Procedura posiada ściśle określony zestaw parametrów wejściowych i wyjściowych, a implementacji realizowanego przez nią algorytmu nie sposób zmienić.

Uniwersalność rozumiana jako wielofunkcyjność jest trudnym do zapewnienia kryterium i skłaniamy się do poglądu, że rama nie musi być w ten sposób uniwersalna. Stworzenie "Uniwersalnej Ramy Wszystkiego" jest z całą pewnością informatyczną (i filozoficzną) mrzonką. Jeżeli jednak uniwersalność rozumiemy jako możliwość zastosowania dla rodziny pokrewnych potrzeb, to rama powinna z definicji cechować się tą właściwością.

Rama kryjąca projekt (szkielet) całego programu, jak np. *Choices*, różni się od tradycyjnego (czasem istniejącego jedynie w postaci półformalnego zapisu na papierze) projektu tym, że jest **instancją** ogólnego wzoru, a nie zestawem wytycznych oraz tym, że zawsze jest zapisana w konkretnej notacji. Może być to dowolna notacja formalna, najczęściej jednak jest to odpowiedni do zastosowań język programowania, co dodatkowo stwarza możliwość częściowego sprawdzenia poprawności ramy za pomocą procesu kompilacji.

Rama może istnieć zatem w wielu postaciach (reprezentacjach), począwszy od formalnego zapisu w notacji wysokiego poziomu, poprzez zapis w języku programowania, aż po przekształconą w procesie kompilacji postać ładowalną, gotową do dynamicznej konsolidacji.

2.2 Kryteria systematyzacji ram

Bazując na przedstawionych przykładach ram, proponujemy następujące kryteria ich systematyzacji:

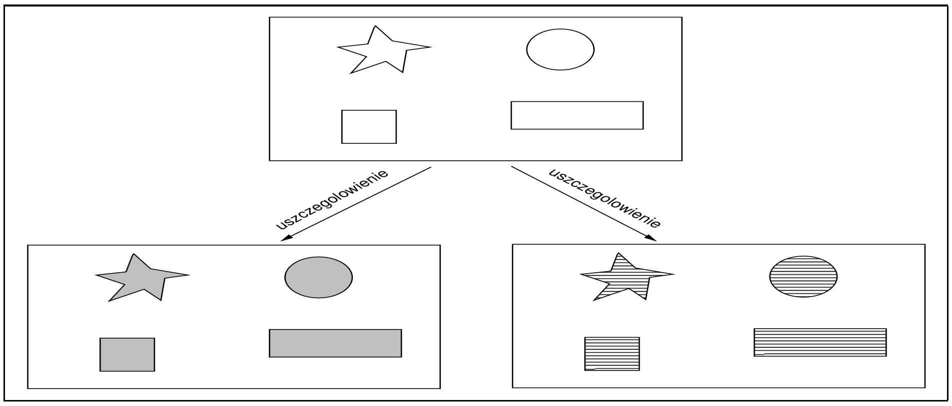
Stopień abstrakcji i komplementarne względem niego pojęcie **stopnia konkretności**. Ramy mogą zawierać w sobie mniej lub więcej elementów abstrakcyjnych, umożliwiających modyfikację i redefinicję ich cech funkcjonalnych. Elementy te określają uniwersalność ramy, jej zdolność dostosowania do zmieniających się zastosowań. I tak np. komórki w PCN są bardziej konkretne niż szablony, ponieważ specyfikują ściśle kod do wykonania w reprezentowanym schemacie interakcji. Właściwość sprawiająca, że do szablonów kod ten można podać jako parametr, czyni je bardziej abstrakcyjnymi i uniwersalnymi.

Zakres funkcjonalny ram. Ramy o dużym zakresie funkcjonalnym, jak np. *ToolTalk* realizują szeroki zestaw funkcji i usług (w tym przypadku protokołów komunikacji). Ramy mniejsze, np. szablony w PCN, realizują węższy ich zestaw, choć jednocześnie mogą cechować się wysokim stopniem abstrakcji (jak jest w tym wypadku).

Sposób reprezentacji ramy określa stopień jej przekształcenia, które może przebiegać od zapisów w formalizmach wysokiego poziomu, poprzez teksty w konkretnym języku programowania aż po postać skompilowaną, dostępną bezpośrednio do dynamicznego łączenia (konsolidacji).

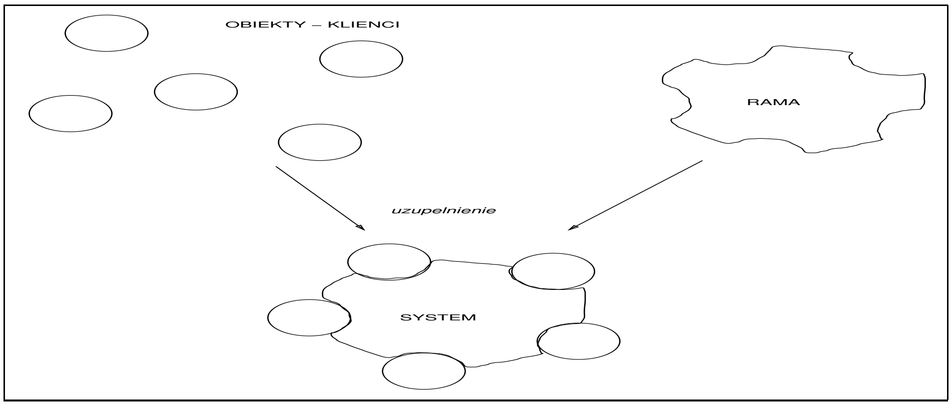
Typ przekształcenia. Zarysowują się dwa zasadnicze sposoby przekształcenia ramy w system użytkowy (czyli jej wykorzystania):

1. **Uszczegółowienie** (ang. *refinement*), polegające na ukonkretnieniu zawartych w ramie abstrakcji, np. kodu, który ma być wykonany w węzłach szablonu w PCN albo metod abstrakcyjnych w *Choices*. Stosując uszczegółowienie na wiele sposobów można utworzyć klasę podobnych systemów konkretnych (patrz rys. 1).



Rysunek 1: Uszczegółowienie

2. **Uzupełnienie,** polegające na dołączeniu do ramy obiektów zewnętrznych, korzystających z jej usług i funkcji. Wyobraźmy sobie pakiet oprogramowania o nazwie *DesktopPublishingSystem*, zawierający m.in. edytory tekstu i grafiki, konwertery formatu dokumentów, przeglądarki, moduły drukarki i naświetlarki, itp.). Programy te, będące odrębnymi aplikacjami, komunikują się ze sobą za pomocą zestawu protokołów, dostarczanych i realizowanych przez *ToolTalk*. Są one w zasadzie samodzielnymi klientami usług ramy, pełniącej w tym wypadku rolę koordynatora. Rama i klienci są samodzielnymi jednostkami, które nawzajem się uzupełniają (patrz rys. 2).



Rysunek 2: Uzupełnienie

Typ relacji między ramą a systemem konkretnym, utworzonym z jej pomocą.

W zależności od typu przekształcenia może to być relacja jednego z dwóch rodzajów:

1. **jest-typu**, oznaczana $\overset{is-a}{\sim}$ dla systemów utworzonych w procesie uszczegóławiania, np.

$$SunChoices \overset{is-a}{\sim} Choices,$$
$$EncoreChoices \overset{is-a}{\sim} Choices$$

2. **jest-częścią**, oznaczana $\xrightarrow{is-part-of}$ oraz odwrotna do niej relacja **posiada**, oznaczana $\xrightarrow{has}$, np.

$$ToolTalk \xrightarrow{is-part-of} DesktopPublishingSystem \equiv$$
$$DesktopPublishingSystem \xrightarrow{has} ToolTalk$$

3 Metoda konstrukcji ram w językach obiektowych

Badaliśmy możliwości stworzenia i zastosowania ram enkapsulujących schematy interakcji między procesami w systemach rozproszonych. Stosując terminologię poprzedniego paragrafu można powiedzieć, że ramy takie charakteryzują się niewielkim zakresem funkcjonalnym. Powinny one jednak cechować się wystarczająco wysokim stopniem abstrakcji i elastyczności, aby mogły być efektywnie i wielokrotnie używane w różnych zastosowaniach. Podjęliśmy próbę opracowania metody konstrukcji takich ram przy użyciu mechanizmów dostępnych w językach obiektowo zorientowanych. Stosując różne jej warianty, stworzyliśmy ramy dla szeregu schematów interakcji, takich jak strumień, pierścień czy gwiazda (zwana także farmą procesorów). Każda z nich została zastosowana w kilku niedużych programach rozproszonych (zaimplementowano m.in. różne rodzaje sortowania, generacji liczb pierwszych, obliczanie fraktala Mandelbrota, itp.). Zarówno ramy, jak i wspomniane programy użytkowe zostały napisane w obiektowo zorientowanym języku programowania równoległego i rozproszonego SR (*Synchronizing Resources*) [1, 7] oraz uruchomione i przetestowane na sieci stacji roboczych SUN.

Opracowaliśmy metodę konstrukcji aplikacji rozproszonych z użyciem ram w klasie języków imperatywnych stosunkowo niewielkim kosztem implementacyjnym, tj. bez tworzenia notacji wysokiego poziomu i jej kompilatora. Zdecydowaliśmy pozostać na średnim poziomie abstrakcji – poziomie języków programowania ogólnego użytku. Ramy tworzone w takich językach są bardziej elastyczne i łatwiejsze do konfiguracji lub zmiany. Wybraliśmy języki obiektowo zorientowane, aby móc wykorzystać mechanizmy, których dostarczają ich kompilatory. Mechanizmy te to wielokrotne dziedziczenie lub import (dla języków nie posiadających własności dziedziczenia kodu). Ten drugi mechanizm jest dostępny między innymi w takich językach jak SR czy DisCo [4]. Polega on na możliwości zadeklarowania klasy *importującej* inną klasę, przy czym klasa importowana nie wchodzi w ten sposób w relację dziedziczności z importującą. Obiekt utworzony z klasy importującej uzyskuje w ten sposób dostęp do interfejsu klasy importowanej i może dzięki temu utworzyć obiekty tej klasy by potem korzystać z ich usług (tj. wywoływać ich metody).

W poniższym paragrafie opiszemy jak konstruować i korzystać z obiektowo zorientowanych ram enkaspulujących schematy interakcji między procesami w programach współbieżnych i rozproszonych. Przedstawimy koncepcję obiektów sklejających, które w proponowanym schemacie konstrukcyjnym pełnią rolę elementów pośrednich (spoiwa) między wykorzystywaną przez programistę ramą a specyficznym dla tworzonej aplikacji algorytmem obliczenia.

3.1 Obiekty sklejające

Konstruktor ram enkaspulujących schematy interakcji między procesami ma do rozwiązania dwa zasadnicze problemy: budowę struktury komunikacyjnej odpowiedniej dla implementowanego schematu interakcji oraz opracowanie sposobu łączenia dwóch oddzielnych i z zasady niezależnych elementów, z których programista składa aplikację: ramy i kodu specyficznego dla obliczenia.

Rozwiązanie problemu pierwszego jest w wysokim stopniu zależne od właściwości języka, w którym konstruujemy ramę. SR, który wykorzystywaliśmy, oferuje bogaty wybór mechanizmów komunikacji i synchronizacji między procesami. Daje to konstruktorowi ramy swobodę przy wyborze najbardziej odpowiednich z nich. My skorzystaliśmy z dwóch: zdalnego wywołania procedury (RPC) i przekazywania komunikatów. Należy jednak zdawać sobie sprawę, że nie wszystkie języki programowania współbieżnego i rozproszonego są tak wygodne pod tym względem. Jesteśmy jednak przekonani, że znalezienie efektywnego rozwiązania zagadnienia komunikacji możliwe jest w każdym z nowoczesnych środowisk programowania rozproszonego.
Bardzo ważny jest sposób reprezentacji wirtualnej topologii, którą tworzą komunikujące się procesy. Proponujemy rozwiązanie w naszym rozumieniu najbliższe istocie metody obiektowej:

- Wszystkie elementy wchodzące w skład działającego programu są obiektami.
- Obiekty komunikują się za pomocą wzajemnych wywołań metod, udostępnionych w ich interfejsach.
- Proces obliczeniowy, wykonywany współbieżnie, jest reprezentowany przez wątek (lekki proces), powstający na skutek wywołania metody obiektu i wykonujący się w jego przestrzeni adresowej.
- Protokół komunikacyjny jest zdefiniowany przez określony zestaw metod zawartych w interfejsie obiektów i odpowiada implementowanemu schematowi interakcji.

Rozwiązanie problemu drugiego powinno być naszym zdaniem niezależne od wybranego języka, pod warunkiem, że jest on obiektowo zorientowany.
Proponujemy zatem wykorzystanie dziedziczenia lub alternatywnego względem niego importu jako mechanizmu konstrukcji elementów pośrednich między ramą a kodem źródłowym właściwym dla implementowanego przez użytkownika-programistę algorytmu. Elementy takie nazwaliśmy obiektami sklejającymi (ang. *glue objects*). Zawierają one zarówno możliwy do równoległego wykonania kod obliczenia, specyfikowany przez użytkownika, jak i interfejs komunikacyjny. Obiekty takie są używane do konstrukcji styku algorytmu obliczeniowego z ramą. Użytkownik definiuje w nich najczęściej jedną prostą metodę "sklejającą". Nazwa pochodzi stąd, iż metoda taka łączy w sobie funkcje obliczeniowe z synchronizacyjnymi w ramach stosowanego schematu interakcji. Metoda ta określa sposób, w jaki proces będzie komunikował się z innymi procesami-uczestnikami obliczenia równoległego. Rama wg. naszej propozycji składa się z dwóch klas: dostarczającej metod komunikacyjnych, zwanej `Comm` i drugiej, strukturalizującej całość modelu obliczenia, której nazwa jest specyficzna dla tego modelu, np. `Pipe, Ring, Mesh`, itp. (patrz też rys. 3). Przyjęliśmy założenie, że stosowany język posiada silny system typów oraz nie ma własności metaprogramowania, tzn. typy wszystkich wyrażeń muszą być określone na etapie kompilacji, a kod nie może być parametrem procedury. Cechy takie posiada większość imperatywnych języków programowania rozproszonego, m.in. SR. Nakłada to dodatkowe ograniczenia na proponowaną metodę. Wspomnimy o nich poniżej. Istota proponowanej metody pozostaje jednak bez zmian. Sposób jej wykorzystania obejmuje następujące etapy:

1. **Dekompozycja algorytmu przez programistę,** polegająca na wydzieleniu z algorytmu równoległego części, mogącej wykonywać się jako równoległy proces. Otrzymany w ten sposób fragment kodu źródłowego można zapisać jako metodę `compute()` klasy `UserCode`. Oba wymienione identyfikatory nie są obligatoryjne, raz wybrane muszą jednak być konsekwentnie stosowane w następnych etapach.

2. **Zdefiniowanie klasy `Glue`, będącej wzorcem dla obiektów sklejających.** Ta nazwa jest obowiązkowa, stanowi bowiem dla kompilatora informację identyfikującą jednoznacznie klasę sklejającą. Definicja klasy `Glue` musi zawierać specyfikację metody sklejającej, właściwej dla realizowanego modelu obliczenia. Metoda ta powinna odwoływać się do wyżej wspomnianej funkcji `compute()` oraz metod komunikacyjnych, dostarczanych przez ramę w specjalnej klasie o ustalonym identyfikatorze `Comm`. Oto przykład takiej metody dla obliczenia w pierścieniu (zapis w abstrakcyjnym języku obiektowo zorientowanym):

   ```
   method compute_in_ring(data)
   {
     send_right(compute(data) + receive_left());
   }
   ```

   gdzie `send_right()` i `receive_left()` są metodami komunikacyjnymi jednokierunkowego pierścienia, dostarczanymi przez klasę `Comm`. Dlatego `Glue` musi dziedziczyć lub importować `Comm`. Ramę reprezentuje klasa dziedziczącą lub importującą `Glue`, o nazwie równoznacznej z implementowaną topologią (np. `Pipe, Ring, Mesh`). Utworzenie jej wystąpienia (instancji) spowoduje powstanie wirtualnej topologii komunikacyjnej, składającej się z obiektów typu `Comm` i `Glue`. Hierarchia klas obiektów tworzących aplikację wg. opisywanej metody wraz z typem ich powiązań jest zilustrowana na rys. 3.

3. **Zapis programu głównego (aplikacji)**, z utworzeniem ramy i zbudowaniem modelu obliczenia za pomocą odpowiedniego równoległego wywołania obiektów sklejających tworzących ramę.

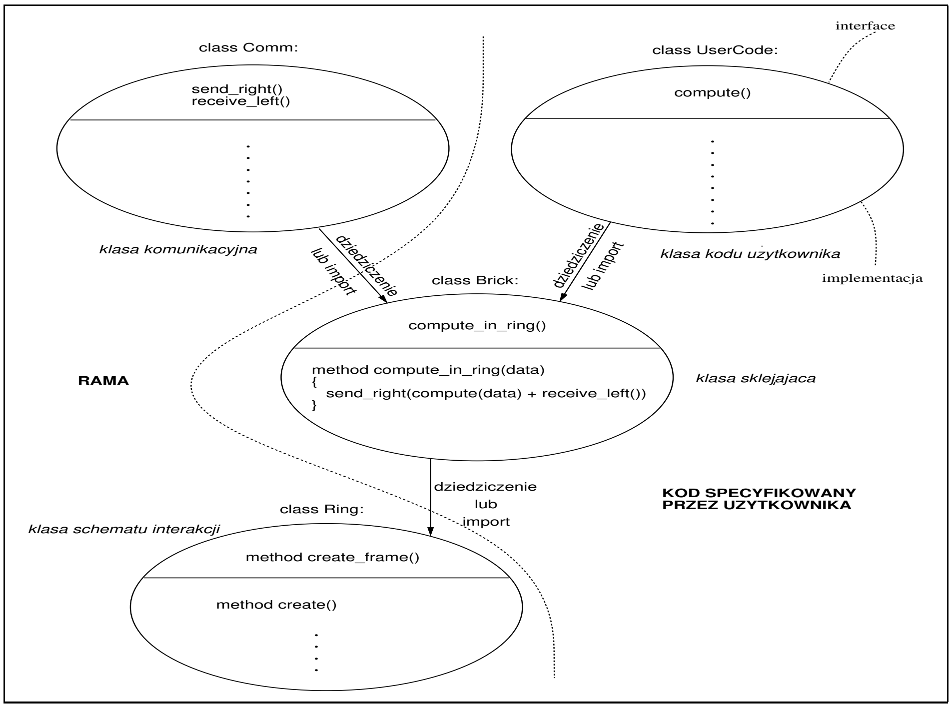
4. **Kompilacji**, która dzięki odpowiednim adnotacjom o wzajemnych relacjach wymienionych klas połączy je mechanizmami dziedziczności i importu, umożliwiając wykonanie programu. Inaczej mówiąc, kod sekwencyjny specyfikowany przez użytkownika ramy uzyska dostęp do funkcji komunikacyjnych, dostarczanych przez ramę i zostanie w momencie jej utworzenia rozproszony w topologii wirtualnej realizowanego przez nią schematu interakcji.

Opiszemy teraz różnice, jakie wynikają z budowy ramy w oparciu o mechanizm dziedziczenia w porównaniu z importem.

Wykorzystanie mechanizmu dziedziczenia

Rama, reprezentowana przez obiekt, w zależności od dostępnego mechanizmu może dziedziczyć lub importować klasę sklejającą. W przypadku dziedziczenia w tworzonej aplikacji występuje w zasadzie tylko jeden obiekt – rama, raz utworzona, do której metod sklejających, odziedziczonych z klasy sklejającej aplikacja bezpośrednio się odwołuje. Na przykład rama `Ring`:

```
class Ring: {
  inherit Glue
.
 /* szczegoly implementacji sa nieistotne dla uzytkownika ramy */
 /* i zalezne od wybranego jezyka        */
.
}
```



Rysunek 3: Hierarchia klas stosowanych do budowy obiektowo zorientowanych aplikacji rozproszonych w schemacie pierścienia z użyciem ramy

Przykładowa aplikacja ze strukturą pierścieniową korzystająca z takiej ramy ma postać:

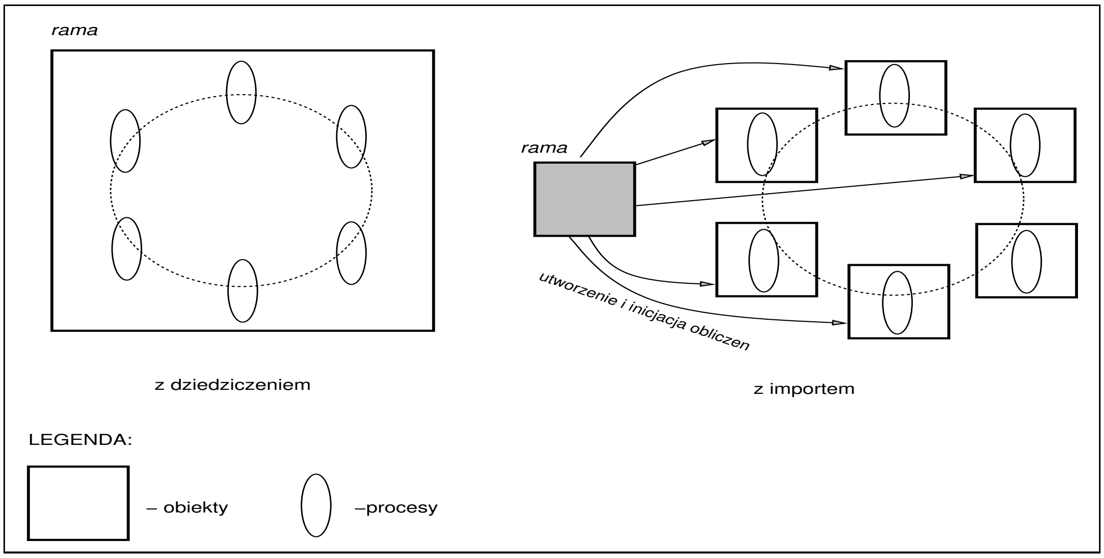
```
main() {
  table Data[N];

  R := create Ring;               /* utworzenie ramy */
  do in parallel for i := 1 to N  /* zasadnicze obliczenie rownolegle */
    R.compute_in_ring(Data[i]);   /* metoda sklejajaca, dziedziczona z klasy sklejajacej */
  R.destroy();   /* zniszczenie obiektu-ramy */
}
```

Rama jest zatem jedynym obiektem, który programista tworzy w aplikacji. Wszelkie inne obiekty tworzone są automatycznie przez ramę. Wywołania zawarte w aplikacji dotyczą wyłącznie metod obiektu-ramy (patrz rys. 4).

Wykorzystanie mechanizmu importu

W przypadku wykorzystywania mechanizmu importu do udostępniania kodu klasy sklejającej dla ramy, tworzy ona z tej klasy osobne obiekty, reprezentujące wchodzące w interakcję procesy (patrz rys. 4). Rama może definiować metody dostępu do tych obiektów (a tym samym do ich metod sklejających), jak w przykładzie:

```
class Ring: {
  import Glue
```



Rysunek 4: Różnice w wewnętrznej strukturze obiektu reprezentującego ramę w zależności od stosowanego mechanizmu udostępniania kodu

```
  table_of_objects glues[N];

  method create_frame(int k) /* tworzy obiekty sklejajace */
  {
    for i := 1 to N
      glues[i] = create Glue;
  }
  method GlueCode(int i, ptr p) /* udostepnia ich metody sklejajace */
  {
    glues[i].compute_in_ring(*p);
  }
}
```

Aplikacja z taką ramą wygląda następująco:

```
main() {
  table Data[N];

  R := create Ring(); /* utworzenie obiektu reprezentujacego rame */
  R.create_frame(N);  /* utworzenie N obiektow obiczeniowych */
  do in parallel for i := 1 to N  /* zasadnicze obliczenie */
    R.GlueCode(i, &Data[i]);     /* z wywolaniem obiektow sklejajacych */
  R.destroy();
}
```

W przypadku korzystania z mechanizmu dziedziczenia rama korzysta jedynie z kodu klasy sklejającej. Używając importu, ramę budujemy z obiektów sklejających w sensie dosłownym, to znaczy obiekt reprezentujący ramę jest odpowiedzialny za ich utworzenie i zainicjowanie w nich obliczeń (wywołanie metod sklejających).

4 Podsumowanie

W historii informatyki można wyróżnić kolejne etapy rozwoju idei wielokrotnego wykorzystania oprogramowania. W latach 40–tych i 50–tych komputery programowano za pomocą pojedynczych instrukcji, początkowo wprowadzanych bezpośrednio w zapisie maszynowym. Przełom lat 50–tych i 60–tych przyniósł pierwszy element wielokrotnego wykorzystania: procedurę (prof. Wheeler z Uniwersytetu w Cambridge). Ciągle udoskonalana (procedury zaczęto łączyć w biblioteki, od niedawna możliwa jest ich dynamiczna konsolidacja), pozostaje ona do dziś powszechnie używanym przez programistów elementem i narzędziem pracy. Wraz z pojawieniem się programowania obiektowego (koniec lat 60–tych) powstał komponent wyższego rzędu: obiekt (klasa), grupujący dane i procedury do ich przetwarzania. Z kolei obiektowo zorientowana rama zawiera obiekty (klasy) powiązane ze sobą siecią relacji różnego typu, podejmujących wspólnie akcje w ramach realizowanego schematu. Rama wydaje się być zatem naturalnym krokiem na drodze rozwoju elementów wielokrotnego wykorzystania.

W powyższym artykule zaproponowaliśmy m.in. ogólną definicję ramy oraz kryteria opisu i klasyfikacji tego pojęcia. Z pewnością już niedługo będzie można je rozszerzyć i wzbogacić. Jesteśmy przekonani, że najbliższe lata przyniosą znaczący wzrost intensywności badań w tej dziedzinie. Poszukiwania te będą zmierzać w kierunku znalezienia jak najdoskonalszych metod tworzenia oraz używania uniwersalnych i elastycznych komponentów programowych. Perspektywa posiadania takich komponentów jest bowiem z pewnością kusząca dla wielu producentów oprogramowania.

Bibliografia

[1] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Trans. on Prog. Languages and Systems*, 10(1):51–86, jan 1988.

[2] R.H. Campbell, N. Islam, R. Johnson, P. Kougiouris, and P. Madany. Choices, Frameworks and Refinement. In *Proceedings IWOOS '91*, Xerox PARC (USA), 1991.

[3] I. Foster, R. Olson, and S. Tuecke. Productive Parallel Programming: The PCN Approach. *Scientific Programming*, 1(1), 1992.

[4] H-M. Järvinen and R. Kurki-Suonio. DisCo Specification Language: Marriage of Actions and Objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, Arlington, Texas, May 1991. IEEE.

[5] C. Jones. *Programming Productivity.* McGraw-Hill, New York, 1986.

[6] M. Minsky. *A Framework for Representing Knowledge.* McGraw-Hill, 1975.

[7] A. Laurentowski K. Zieliński. SR - obiektowo zorientowany język programowania rozproszonego. *Informatyka*, (6), czerwiec 1993.