

Rozdział

Wielowymiarowe rozdzielanie zagadnień, a podejście komponentowe

Andrzej Krzywda*

Instytut Informatyki, Uniwersytet Wrocławski, andrzej@64pola.pl

Tomasz Nazar

Instytut Informatyki, Uniwersytet Wrocławski, tomasz.nazar@ii.uni.wroc.pl

Ewa Gurbiel

Instytut Informatyki, Uniwersytet Wrocławski, ewa.gurbiel@ii.uni.wroc.pl

I have a small mind and can only comprehend one thing at a time

E. Dijkstra

Streszczenie

Wielowymiarowe rozdzielanie zagadnień (WRZ) to podejście do kompozycji oprogramowania oparte na spostrzeżeniu, iż zagadnienia (concerns) systemów można podzielić ze względu na wymiar (rodzaj). Takimi wymiarami są np. klasy, funkcjonalności, aspekty, role itp.

W pracy tej pokazano na przykładzie konkretnego projektu, w jaki sposób można wykorzystać WRZ do implementacji i zarządzania komponentami ponownego użycia.

1. Wprowadzenie

Rozdzielanie zagadnień jest jednym z kluczowych problemów inżynierii oprogramowania. Oznacza ono zdolność do identyfikacji, enkapsulacji i operowania tylko tymi fragmentami systemu, które są odpowiedzialne za konkretny cel, lub problem, nad którym pracujemy. **Zagadnienia** (ang. *concerns*) stanowią podstawę do uporządkowania i dekompozycji oprogramowania na zrozumiałe i łatwe w zarządzaniu moduły.

* Siemens AG, CT SE2 – Architecture, Monachium, Niemcy

Zagadnienia można pogrupować ze względu na wymiary

- klasy
- możliwości (ang. *features*)
 - zdalny dostęp
 - interfejs użytkownika
- trwałość
- role
- wzorce projektowe
- warianty
- konfiguracje

Ostatnie dwa punkty są szczególnie istotne w przypadku **rodzin programów** (ang. *family of software*) lub zarządzania **liniami produkcyjnymi** (ang. *product lines management*).

Osiągnięcie pełnego rozdzielania zagadnień powinno zaowocować:

- redukcją złożoności oprogramowania;
- redukcją wpływu zmian na system;
- umożliwieniem ewolucji systemu informatycznego;
- wspomaganie ponownej używalności modułów;
- uproszczeniem integracji komponentów.

Do tej pory nie udało się osiągnąć wszystkich celów w praktyce [TARR2000]. Główną przyczyną takiego stanu rzeczy jest fakt, że zbiór związanych ze sobą zagadnień zmienia się w czasie i występuje w ściśle określonym kontekście.

Różne działania, etapy tworzenia oprogramowania, programiści często wymagają zagadnień różnych rodzajów, a więc występujących w różnych wymiarach. Rozdzielenie zagadnień często ułatwia pracę nad pewnym zbiorem zagadnień w danym wymiarze, jednocześnie utrudniając pracę w innym.

W rozdziale 2 tego artykułu opisano wielowymiarowe rozdzielanie zagadnień (ang. *multi-dimensional separation of concerns* - MDSoc) oraz język Hyper/J, będący implementacją MDSoc dla języka Java. Rozdział 3 to opis podejścia komponentowego do tworzenia oprogramowania. W rozdziale 4 dyskutujemy o wykorzystaniu MDSoc do implementacji komponentów ponownego użycia, oraz opisujemy nasze doświadczenia z prac nad konkretnym systemem. W rozdziale 5 porównujemy WRZ z innymi metodologiami w zakresie implementacji i zarządzania komponentami ponownego użycia. Rozdział 6 podsumowuje pracę oraz naświetla obszar naszych przyszłych badań związanych z tematem pracy.

2. Wielowymiarowe rozdzielanie zagadnień - Hyper/J

2.1. Tyrania dominującej modularyzacji

Tyrania dominującej modularyzacji (ang. *the tyranny of dominant decomposition*) oznacza wymuszenie przez język programowania jednego sposobu podziału systemu na moduły. Przykładem są klasy w językach obiektowych, funkcje w językach funkcyjnych czy też więzy w językach logicznych. Klasy, funkcje oraz więzy są przykładami tzw. wymiarów dekompozycji systemu. Możliwość modularyzacji ze względu na jeden wymiar (np. klas) powoduje, iż elementy alternatywnego wymiaru (np. funkcji) są rozrzucone (ang. *scattered*) między modułami, oraz wymieszane (ang. *tangled*) z innymi elementami.

Sytuacja taka powoduje, iż niemożliwe jest czerpanie korzyści z różnych modularyzacji systemu w trakcie tworzenia oprogramowania.

Przykład 1.

Implementacja metody *toString()* w języku Java jest przykładem rozrzużenia zagadnienia na przestrzeni wielu klas, a przez to również powoduje ona wymieszanie tej funkcjonalności z innymi zagadnieniami systemu jak np. opis zasad biznesowych.

Definicja 1.

Wielowymiarowe rozdzielanie zagadnień oznacza:

- możliwość istnienia **wielu** wymiarów zagadnień;
- rozdzielanie zagadnień względem tych wymiarów jest **równoległe**, żaden wymiar nie może być dominujący;
- **dynamiczne** zarządzanie nowymi zagadnieniami tzn. w chwili ich pojawiania się w systemie.

Wielowymiarowe rozdzielanie zagadnień jest wsparciem dla **modularyzacji systemu na żądanie** (ang. *on-demand remodularization*) [TARR2000], czyli procesu pozwalającego na wybór dowolnej modularyzacji zagadnień w istniejącym systemie.

Język **Hyper/J** umożliwia korzystanie z wielowymiarowego rozdzielania zagadnień w języku Java.

2.2. Hiperprzestrzenie

Definicja 2.

Hiperprzestrzeń stanowi kontener dla wszystkich zagadnień (ang. *concerns*) związanych z wytwarzaniem systemu (lub rodziną systemów). Można ją traktować jak wielowymiarową macierz.

Każda oś macierzy reprezentuje wymiar, a punkty na osi to zagadnienia w tym wymiarze.

Wymiary pomagają spojrzeć na świat, który modelujemy z różnych perspektyw.

Przykład 2.

Definicja hiperprzestrzeni w języku Hyper/J

hyperspace TopoTree

composable class model.label.;*

composable class model.root.;*

Definicja w przykładzie 2 opisuje hiperprzestrzeń o nazwie **TopoTree**. Pokazuje ona, które pakiety do niej należą, oraz pozwala określić czy elementy w danym pakiecie mogą być komponowane (słowo kluczowe **composable**) z innymi elementami. Oznaczenie słowem **uncomposable** uniemożliwiłoby komponowanie klas danego pakietu z innymi elementami.

2.3. Odwzorowanie zagadnień

Odwzorowanie elementów kodu w zagadnienia podajemy za pomocą reguł przedstawionych w przykładzie 3.

Przez elementy kodu rozumiemy w tym kontekście pakiety, klasy lub metody.

Przykład 3.

Przykład odwzorowań w języku Hyper/J

package model.core : Model.Core

package model.core : Applications.TopoTree

Pierwszy wiersz w przykładzie 3 oznacza, iż cały pakiet *model.core* należy do zagadnienia o nazwie *Core* w wymiarze o nazwie *Model*. Drugi wiersz oznacza, iż ten sam pakiet należy do zagadnienia o nazwie *TopoTree* w wymiarze o nazwie *Application*. Jak widać, w przykładzie 3, nic nie stoi na przeszkodzie, aby te same elementy kodu (w tym przypadku pakiet *model.core*) mogły należeć do różnych zagadnień w różnych wymiarach. Istotne jest przy tym jedno zastrzeżenie, iż względem jednego wymiaru, element kodu może należeć tylko do jednego zagadnienia.

2.4. Hiperwarstwy

Definicja 3.

Hiperwarstwa jest zbiorem zagadnień, który jest **deklaratywnie kompletny**.

Definicja 3 jest jedną z kluczowych definicji dotyczących wielowymiarowego rozdzielania zagadnień. Oznacza ona, iż hiperwarstwa nie musi zawierać implementacji wszystkich wymaganych (ang. *expected*) do jej funkcjonowania metod, wystarczy że zostaną one zadeklarowane i odpowiednio oznaczone.

Wymagane elementy określają zależność między zagadnieniami. Przykładem może być implementacja hiperwarstwy, będącej implementacją graficznej reprezentacji drzewa. Metoda *showTree()* mająca wyświetlić drzewo w interfejsie użytkownika wymaga

dostępu do metody *getComponent()* dostarczającej graficznego komponentu, w którym można to drzewo osadzić.

Każda hiperwarstwa może deklarować wiele wymaganych elementów.

Język **Hyper/J** umożliwia dwa sposoby oznaczania wymaganych elementów:

- Jeśli wymagany elementem jest klasa, deklarujemy ją jako abstrakcyjną.
- W przypadku, gdy nie możemy deklarować klasy jako abstrakcyjnej oznaczamy wszystkie wymagane metody tak jak w przykładzie 4.

Przykład 4.

Deklaratywna kompletność w języku **Hyper/J**.

```
public class Tree {
    public void setRoot(Node node) {throw new UnimplementedError();}
}
```

Elementy wymagane przez hiperwarstwę mogą być dostarczone przez **wiele** różnych modułów.

Deklaratywna kompletność eliminuje powiązania między zagadnieniami, co ułatwia konfigurację i ponowną używalność. Z praktycznego punktu widzenia umożliwia ona pracę nad hiperwarstwami w dowolnym środowisku wspierającym język **Java**.

Definicja 4.

Zbiór hiperwarstw, które wspólnie się uzupełniają (dostarczają sobie nawzajem wszystkie wymagane elementy) nazywamy **hipermodułem**.

Przykład 5.

Hipermoduł **TopoTree**

hypermodule TopoTree

hyperslices:

```
Model.Core,
Model.Hierarchy,
Model.Root,
Model.Label,
Model.State,
Gui.Tree,
Gui.Frame;
```

relationships:

```
mergeByName;
```

end hypermodule;

Przykład 5 jest kompletnym opisem hipermodułu **TopoTree**.

W sekcji **hyperslices** wymienione są wszystkie hiperwarstwy wchodzące w skład hipermodułu. Istotne jest, aby wszystkie elementy wymagane przez wymienione hiperwarstwy były dostarczone przez hiperwarstwy występującej w tej samej definicji hipermodułu.

Sekcja **relationships** pozwala określić ogólną strategię kompozycji hiperwarstw. Strategia **mergeByName** zakłada, iż elementy kodu o tej samej nazwie, mają zostać ze sobą skomponowane. Jeśli w dwóch różnych zagadnieniach występuje klasa *Node*

oznacza to, iż wszystkie znajdujące się w nich metody i pola zostaną wkomponowane do klasy **Node** w hipermodule **TopoTree**.

Alternatywną strategią jest **nonCorrespondingMerge**, która domyślnie nie komponuje klas ze sobą, lecz wymaga dokładnych zasad określających jak odpowiadają sobie klasy w różnych hiperwarstwach.

2.5. Hyper/J

Hyper/J jest narzędziem, które realizuje wielowymiarowe rozdzielanie zagadnień w języku **Java**. Jako parametry podajemy:

- definicję hiperprzestrzeni (patrz przykład 2),
- odwzorowanie zagadnień (patrz przykład 3),
- definicję hipermodułu (patrz przykład 5),
- pliki *.class* (skompilowane klasy w języku **Java**)

W wyniku otrzymujemy wygenerowany hipermoduł, składający się tylko i wyłącznie z hiperwarstw wymienionych w jego definicji.

Język **Hyper/J**, w przeciwieństwie do innych innych języków aspektowych ([AspectJ2001], [Cesar2003]) nie definiuje nowych słów kluczowych dla języka **Java**, dzięki czemu możemy używać go w swoich ulubionych środowiskach tworzenia oprogramowania (ang. *IDE*).

3. Podejście komponentowe

Komponent jest elementem oprogramowania o następujących cechach ([MEYE1997])

- Może być używany przez inne elementy (przez klientów).
- Dostawca komponentu nie musi wiedzieć, kim jest jego klient.
- Klient używa komponentu opierając się wyłącznie na oficjalnych informacjach.

Z punktu widzenia wielowymiarowego rozdzielania zagadnień, hiperwarstwy mogą być traktowane jako komponenty. Spełniają one wszystkie wyżej wymienione założenia:

- Są używane przez hipermoduły w procesie kompozycji.
- Programista hiperwarstwy nie musi być świadomy, w jakim kontekście zostanie ona użyta.
- Oficjalne informacje dostarczane przez hiperwarstwę to istniejące w niej klasy oraz metody, które można użyć w procesie kompozycji.

4. TopoTree

Aby lepiej zrozumieć wielowymiarowe rozdzielanie zagadnień posłużymy się projektem o nazwie **TopoTree**. Został on zaimplementowany¹ w **Siemens AG, Monachium** w celu pokazania możliwości języka **Hyper/J**. Projekt jest częścią badań nad praktycznym zastosowaniem języków zorientowanych aspektowo. Wymagania zostały tak narzucone, aby zaimplementować funkcjonalności typowe dla większości aktualnie prowadzonych w firmie projektów.

Wymagania dotyczące tego projektu są następujące:

- Reprezentacja sieci urządzeń, składającej się z fizycznych jednostek.
- Sieć urządzeń jest hierarchiczna.
- Możliwość grupowania fizycznych jednostek w logiczne grupy (abstrakcyjne węzły).
- Na pierwszym poziomie znajduje się tylko jedna grupa (korzeń).
- Każde urządzenie może znajdować się w określonym stanie alarmowym.
- Określona strategia propagowania stanu w górę drzewa.
- Interfejs użytkownika – wizualizacja sieci w postaci drzewa.
- Struktura urządzeń oraz ich stan mają być zapamiętywane w bazie danych.
- Zdalny, współbieżny dostęp wielu aplikacji – klientów.

4.1. Implementacja

Pierwsza iteracja tworzenia systemu to identyfikacja podstawowych klas programu umieszczonych w pakiecie *model.core*. Wyodrębniono klasy – **TopoTree**, **Node**.

Przykład 6.

Metoda reprezentująca podstawową funkcjonalność systemu **TopoTree**

```
public class TopoTree {
    public void fillUpTopoTree() {
        Node siemens = new Node("Siemens");
        Node poland = new Node("Poland", siemens);
        Node germany = new Node("Germany", siemens);
        (..)
    }
}
```

Wymagane jednostki kodu to:

- Klasa *Node*
 - Konstruktor *Node(String)*
 - Wymaga istnienia metody *setName(String)* w klasie *Node*

¹ Projekty **TopoTree** oraz **FamilyTree** zaimplementował A. Krzywda w czasie pobytu w Siemens AG w Monachium (listopad 2003 – kwiecień 2004).

- Konstruktor *Node(String, Node)*
 - Wymaga istnienia metody *setFather(Node)* w klasie *Node*
- Metoda *setRoot(Node)* w klasie *TopoTree*

W pakiecie *model.core* (odwzorowanym w zagadnienie *Model.Core*) deklarujemy wymagane metody w sposób opisany w przykładzie 4. Uruchomienie aplikacji **TopoTree** w takiej postaci, spowoduje wyrzuceniem wyjątku **UnimplementedError**.

Na podstawie wymaganych elementów można określić główne zagadnienia związane z podstawową funkcjonalnością systemu:

- etykietowanie węzłów – *Model.Label* – dostarczy metodę *setName(String)*,
- hierachia węzłów – *Model.Hierarchy* – dostarczy metodę *setFather(Node)*,
- korzeń drzewa – *Model.Root* – dostarczy metodę *setRoot(Node)*.

Takie odwzorowania oznaczają, iż *Label*, *Hierarchy*, *Root* są zagadnieniami znajdującymi się w wymiarze o nazwie *Model*. Implementacja zagadnienia *Model.Hierarchy* znajduje się w pakiecie *model.hierarchy*.

Przykład 7.

Implementacja zagadnienia *Model.Root*

```
public class RootContainer {
    private Root root;
    public Root getRoot() {return root;}
    public void setRoot(Root root) {this.root = root;}
}
```

```
public abstract class Root {}
```

Przykład 8.

Implementacja zagadnienia *Model.Hierarchy*

```
public class TreeNode {
    private TreeNode father;
    private Vector children;
    public TreeNode getFather() { return this.father; }
    public Vector getChildren() { return this.children;}
    public void setFather(TreeNode father) {
        this.father = father;
        if (!father.getChildren().contains(this)) {father.addChild(this);}
    }
    public void addChild(TreeNode son) {
        getChildren().add(son);
        if (son.getFather() == null) {son.setFather(this);}
    }
}
```

Późniejszej kompozycji złożonej z zagadnień *Model.Root*, *Model.Hierarchy* oraz *Model.Core* nie przeszkadza fakt, iż obie używają innych nazw klas. Zasady kompozycji określają, które klasy oraz metody należy do siebie przyrównać (ang. *equate*).

Przykład 9.

Zasady kompozycji hipermodułu *TopoTree*

```
equate class
    Model.Hierarchy.RootContainer,
    Model.Core.TopoTree;
equate class
    Model.Core.Node,
    Model.Root.Root,
    Model.Hierarchy.TreeNode;
```

W przykładzie 9, klasy *RootContainer* i *TopoTree* z różnych hiperwarstw zostają do siebie przyrównane, co umożliwia im nawzajem uzupełnienie wymaganych metod.

Dzięki możliwości przyrównywania klas jak w przykładzie 9, dla każdej hiperwarstwy można użyć różnych nazw klas i metod. Zwiększa to możliwość ponownego użycia hiperwarstw w innej aplikacji.

Implementacja i kompozycja wszystkich zagadnień pozwoliła nam na uruchomienie i bezbłędne wykonanie aplikacji **TopoTree**. Aby jednak mieć możliwość graficznej wizualizacji systemu, należy zaimplementować interfejs użytkownika. Wyróżniono następujące zagadnienia w wymiarze o nazwie *Gui*:

- *Gui.Frame*
- *Gui.Tree*
- *Gui.StatusBar*

Składowe interfejsu użytkownika implementowane były niezależnie od siebie używając różnych konwencji nazewniczych. Fragment pliku kompozycji (przykład 10) pokazuje, w jaki sposób zostały one zintegrowane.

Przykład 10.

Zasady kompozycji systemu **TopoTree**

```
equate class
    Model.Core.TopoTre,
    Model.Root.RootContainer
    (...)
    Gui.Frame.FrameApplication,
    Gui.StatusBar.Application,
    Gui.Tree.Application;
equate operation
    Gui.Tree.initTree,
    Gui.StatusBar.addStatusBar
```

Przykład 10 pokazuje, w jaki sposób przyrównujemy do siebie klasy oraz metody. Rezultatem takich zasad kompozycji będzie jedna wspólna klasa z metodami i polami klas *TopoTree*, *RootContainer*, *FrameApplication*, *Application*. Przyrównanie metod

oznacza, iż zostaną one wykonane wspólnie, w porządku podanym w zasadach. Tak więc metoda *addStatusBar()* klasy *Application* zostanie wykonana po metodzie *initTree()*.

4.2. Ponowne użycie hiperwarstw

Dzięki temu, iż każdą hiperwarstwę zaimplementowano traktując je niezależnie od aplikacji **TopoTree**, możemy użyć ich w innym kontekście bez wprowadzania w nich żadnych zmian. Zasady kompozycji oparte na przyrównywaniu elementów umożliwiają podmianę nawet podstawowego komponentu systemu *TopoTree* jakim jest *Model.Core*. Umożliwia to zbudowanie aplikacji, o zupełnie różnym modelu biznesowym korzystając z gotowych komponentów – hiperwarstw. W tym przypadku moglibyśmy traktować hiperwarstwy modelu biznesowego jako komponenty, które możemy wymienić na hiperwarstwy innego modelu biznesowego.

4.3. Aplikacja *FamilyTree*

Aplikacja **FamilyTree** umożliwia reprezentowanie drzewa genealogicznego. Została opracowana w celu pokazania, iż korzystając z gotowych hiperwarstw można szybko zbudować system o innym modelu biznesowym. Głównymi założeniami było korzystanie z już istniejącego kodu źródłowego, oraz niewprowadzanie w nim żadnych zmian. Wszelkie zmiany odbywały się tylko w definicji hipermodułu **FamilyTree**.

Podstawowa funkcjonalność wymaga istnienia klas *FamilyTree* oraz *FamilyMember*. Zagadnienie *Model.Hierarchy* w tym przypadku umożliwia nam reprezentowanie powiązań między członkami rodziny. Hiperwarstwy wymiaru *Gui* pozwalają na użycie wcześniej zaimplementowanego interfejsu użytkownika.

Proces tworzenia aplikacji **FamilyTree** sprowadził się tylko do implementacji hiperwarstwy *Applications.FamilyTree*, gdzie umieszczono implementacje klas *FamilyTree* i *FamilyMember* oraz skonstruowania definicji hipermodułu **FamilyTree**.

Przykład 11.

Implementacja klas *FamilyMember* oraz *FamilyTree*

```
public abstract class FamilyMember {
    public abstract Vector getParents();
    public abstract void setFirstName(String string);
    public abstract String getFirstName();
    public abstract void addParent(FamilyMember parent);
    public abstract void setChild(FamilyMember child);
    public abstract FamilyMember getChild();
}
public class FamilyTree {
    private String familyName;

    private FamilyMember getRelativeMember() {throw new UnimplementedError();}
    private void setRelativeMember(FamilyMember f) {
        UnimplementedError();}
        throw new
```

```

    private void setFamilyName(String string) {
        this.familyName = string;
    }
    private String getFamilyName() {
        return this.familyName;
    }
    public static void main(String args[]) {
        FamilyTree familyTree = new FamilyTree();
        familyTree.setFamilyName("Kowalski");
        FamilyMember member = Factory.create("Jan",1980,9,8);
        //(..)
        familyTree.setRelativeMember(member);
        familyTree.show();
    }
    private void show() {}
}

```

Wyodrębniono nowy wymiar o nazwie *Applications*, którego hiperwarstwy reprezentują poszczególne aplikacje.

Przykład 12.

Odzworowanie zagadnień wymiaru *Applications*

```

package applications.hyperjTopoTree : Applications.HyperJTopoTree
package applications.familyTree    : Applications.FamilyTree

```

Przykład 13.

Hipermoduł **FamilyTree**

hypermodule FamilyTree

hyperslices:

```

    Applications.FamilyTree,
    Model.Label,
    Model.Identifiable,
    Model.Hierarchy,
    Model.Root,
    Gui.Frame,
    Gui.FrameLookAndFeel,
    Gui.Tree,
    Gui.SplitPane,
    Gui.StatusBar;

```

//zasady kompozycji systemu FamilyTree

4.4. Porównanie z metodyką obiektową

Implementacja aplikacji **TopoTree** oraz **FamilyTree** za pomocą hiperwarstw daje efekt oszczędności kosztów. Ponieważ wykorzystane hiperwarstwy są porównywalnych rozmiarów, można oszacować stopień ponownego użycia kodu na 90%. Oczywiście należy zauważyć, iż obie aplikacje wykorzystują drzewiastą strukturę danych, co

znacznie ułatwiło wykorzystanie wspólnych komponentów. Analogiczne modele danych są jednak częstą sytuacją podczas zarządzania liniami produkcyjnymi. Dzięki lepszej modularyzacji, testowanie komponentów jest prostsze, niż w przypadku metodyki obiektowej. Do kosztów pracy nad systemem należy również doliczyć pracę nad zasadami kompozycji rozwijanych aplikacji.

Charakterystyczną cechą wielowymiarowego rozdzielania zagadnień jest zmniejszenie powiązań między komponentami. Wzajemne zależności ustalane są w specyficznych dla aplikacji zasadach kompozycji. Rozwiązanie takie nie jest możliwe z użyciem metodyk obiektowych. Powoduje to zmniejszenie możliwości tworzenia komponentów ponownego użycia.

Tworzenie komponentów ponownego użycia wiąże się z posiadaniem repozytorium komponentów. Ma to wpływ na proces tworzenia aplikacji, gdyż podczas analizy nowych wymagań należy wziąć pod uwagę opłacalność i sens wykorzystania gotowych komponentów. Zarządzanie komponentami za pomocą repozytorium umożliwia oddzielenie komponentów ponownego użycia od modułów charakterystycznych dla tylko jednej aplikacji.

4.5. Testowanie aplikacji zbudowanych z gotowych komponentów

Każdy odseparowany moduł oprócz implementacji pewnego zagadnienia wprowadza również zbiór przypadków testowych, możliwych do uruchomienia tylko w wygenerowanej aplikacji. Podejście to jest bliskie idei programowania kontraktowego [MEYE1997]. Po każdej kompozycji programu możemy sprawdzić czy odpowiednie asercje są prawdziwe. Do testów użyto biblioteki **JUnit**. Implementując komponenty za pomocą narzędzia **Hyper/J** można używać techniki **testowania przed kodowaniem** (ang. *Test First Programming*). Komponenty są samotestujące się.

4.6. Korzyści

Lepsza modularyzacja systemu wspiera niezależne implementowanie modułów przez różne zespoły. Wielowymiarowe rozdzielanie zagadnień pozwala na używanie konwencji nazewnictwa charakterystycznych dla danego kontekstu, dzięki czemu poszczególne zespoły implementują moduły używając języka charakterystycznego dla dziedziny problemu, mimo iż komponent może zostać wykorzystany w innych aplikacjach. Rozdzielone zagadnienia wraz z przypadkami testowymi zwiększają ogólną jakość systemu.

Wykorzystanie gotowych komponentów w celu wytworzenia nowej aplikacji (**FamilyTree**) pokazuje, iż łatwo można użyć języka **Hyper/J** do zarządzania linią produkcyjną systemów.

Kolejną zaletą jest fakt, iż implementując nowe hiperwarstwy w jednej aplikacji, niskim kosztem możemy ich użyć w innych aplikacjach. Wszystkie zależności między hiperwarstwami opisane są tylko w specyficznych dla aplikacji zasadach kompozycji.

4.7. Problemy

Kompozycja oprogramowania z gotowych komponentów przynosi oczywiste korzyści, lecz istnieją również wady takiego podejścia.

Utrudnieniem jest zaburzenie możliwości tradycyjnego śledzenia przebiegu wykonywania programu, gdyż przyrównane do siebie elementy kodu, mimo iż wykonywane jeden po drugim, występują osobno w kodzie źródłowym. Rozwiązaniem tego problemu (charakterystycznego dla programowania aspektowego [AspectJ2001], [Caesar2003], [WWW2001]) jest wykorzystanie wsparcia odpowiednich narzędzi.

Wielowymiarowe rozdzielanie zagadnień umożliwia korzystanie z idei otwartych klas (ang. *open classes*), co jest możliwe również w takich językach jak np. MultiJava [CLIF2000]. Problemem takiego podejścia jest fakt, iż cechy klasy rozrzucone są na przestrzeni wielu modułów, przez co niemożliwy jest tradycyjny widok klasy. Problem ten rozwiązywany jest za pomocą idei przecinających spojrzeń na kod [JANZ2004].

Pamiętać również należy, iż w przypadku, gdy z komponentów korzysta więcej niż jedna aplikacja, należy ostrożnie podchodzić do procesu refaktoryzacji ([FOWL1999]) kodu.

5. Porównanie z innymi metodologiami

Hyper/J jest zaliczany do języków zorientowanych aspektowo. Najpopularniejszym obecnie językiem aspektowym jest AspectJ [AspectJ2001]. Język ten umożliwia zarówno modularyzację zagadnień przecinających jak i korzystanie z idei otwartych klas. AspectJ wyróżnia się jednak rozszerzeniem języka Java, co wymusza korzystanie z odpowiednich narzędzi. W porównaniu z Hyper/J, w języku AspectJ nie występuje idea deklaratywnej kompletności. AspectJ wymusza również istnienie komponentu bazowego, co uniemożliwia podmianę komponentu reprezentującego model aplikacji.

CaesarJ [Caesar2003] jest językiem aspektowym wzorującym się na Hyper/J. Wykorzystuje ideę deklaratywnej kompletności, lecz podobnie jak AspectJ wymusza posiadanie komponentu bazowego. Dodatkową zaletą jest możliwość aktywacji i deaktywacji komponentów w trakcie działania aplikacji. Język ten znajduje się ciągle w fazie rozwijania i nie nadaje się do wykorzystania w komercyjnych zastosowaniach.

Kolejnym popularnym językiem aspektowym jest JAsCO [JAsCO2004]. Rozszerza on składnię języka Java. Podobnie jak CaesarJ umożliwia aktywację i deaktywację komponentów w trakcie działania programu. Niestety nie umożliwia on korzystania z idei otwartych klas, co powoduje ograniczone możliwości implementacji komponentów.

6. Podsumowanie oraz planowane prace

W pracy pokazano jak, z użyciem języka **Hyper/J**, zaimplementować aplikację dla ustalonych wymagań. Najważniejsze zagadnienia zostały ujęte w osobnych komponentach, które umożliwiają ponowne użycie. Jednocześnie pokazano jak postępować przy implementacji nowych wymagań.

Aktualnie trwają prace nad wykorzystaniem przedstawionego tu podejścia do rozdzielenia zagadnień związanych z trwałością obiektów (ang. *persistence*) zarówno z użyciem systemu PrevaYler ([WWW2002a]) jak i relacyjnych baz danych.

Bibliografia

- [TARR2000] Tarr Peri, *Hyper/J User Manual*, IBM Research, 2000.
- [AspectJ2001] www.eclipse.org/aspectj/
- [Caesar2003] www.caesarj.org
- [CLIF2000] Clifton C., Leavens G., *MultiJava: modular open classes and symmetric multiple dispatch for Java*, OOPSLA, 2000.
- [JANZ2004] Janzen G., De Volder G., *Programming with Crosscutting Effective Views*, ECOOP'2004.
- [JAsCO2004] <http://ssel.vub.ac.be/jasco/>
- [MEYE1997] Meyer B., *Object Oriented Software Construction*, Prentice Hall PTR, 1997.
- [WWW2002a] PrevaYler, <http://www.prevaYler.org>
- [WWW2001] Aspect Oriented Software Development Community, <http://www.aosd.net>
- [FOWL1999] Fowler M., Beck K., Brant J., Opdyke W., Roberts D., *Refactoring: Improving the design of existing code*, Addison-Wesley, 1999.