Package architecture optimization in software application design

VASILE CRĂCIUNEAN, RALF FABIAN, EMIL MARIN POPA
Department for Computer Sciences and Economic Informatics
"Lucian Blaga" University of Sibiu
Ion Rațiu Street, no. 5-7, Sibiu
ROMANIA

Abstract: - One of the most important characteristics of a software application is the fidelity to the object modeled. Generally speaking a software application is developed with the purpose to model, with minimal adjustments, a class of often heterogenic objects. The high complexity of the real objects modeled to be modeled, determines a similar complexity for the software applications they are modeled by.

To achieve our goal of accurate modeling for real systems, we start with 0 fidelity degree objects, and admitting on this way any stepwise adjunction of the specific properties for a real system. Thus the concept has as purpose to define a kind of generalization due property adjunction, increasing the degree of heterogeneity of the initial objects and therefore maximizing the fidelity of the system.

In object oriented design these stepwise adjunction of properties consists in a class hierarchy, starting from abstract and finishing at concrete. The key for controlling the complexity of large software systems is to group classes in a set of cohesive modules with minimal interaction. This paper shows a model for establishing an optimal package architecture in object oriented programming of complex software systems.

Key-Words: - package architecture optimization, cohesive modules, cohesion and coupling principles, complex software systems architecture management.

1 Concepts and notations

If *X* is a set then we will use the following notations:

$|X|$ - for the cardinal of *X*.

C^X or $\overline{X}$ - for the complementary of *X* regarding the total set *U*, which will be understood from the context.

$X \setminus Y = X \cap C_Y$ - for the difference of two sets *X* and *Y*.

Definition 1.1 [3], [7]. A *graph* is a pair $G = (\Gamma, X)$, where X is a set and Γ is a function $\Gamma : X \to 2^X$. The elements of *X* are called *edges* of the graph *G*. If $y \in \Gamma x$ then $(x, y) \in \Gamma$ and the pair (x, y) is called *vertex* of graph *G*. Thus $\Gamma = \{(x, y) \mid y \in \Gamma x\}$ and $\Gamma x = \{y \mid (x, y) \in \Gamma\}$.

Definition 1.2 [3], [7]. A graph $H = (A, \Gamma_A)$ is a *subgraph* of graph $G = (\Gamma, X)$ where $A \subset X$ and Γ_A is defined as $\Gamma_A x = \Gamma x \cap A$. A *partial graph* of $G = (\Gamma, X)$ is a graph $G' = (\Gamma', X)$ where $\Gamma' \subset \Gamma$. A partial graph of a subgraph is called *partial subgraph*.

For a vertex $U = (a, b)$ the edge a is called *initial edge* of the vertex *U* and the edge *b final edge* of vertex *U*. Also *a* is a *immediate predecessor* or *direct ancestor* of *b*, and *b* is *immediate successor* or *direct descendent* of *a*.

Definition 1.3 [3], [7]. A path in graph $G = (\Gamma, X)$ is a sequence of vertices $(u_1, u_2, ..., u_n)$, $u_i \in \Gamma$, $\forall\, i = \underline{1, n}$ such that $\forall\, i = \underline{1, n-1}$ the final edge of u_i is the initial edge of u_{i+1}.

A path is called *simple* if $\forall\, i, j \in \{1, 2, ...n\}$, $i \neq j \Rightarrow u_i \neq u_j$.

A path together with his edges is denoted by $\mu = [x_1, x_2, ..., x_k]$.

A *cycle* ore *circuit* is a patch $\mu = [x_1, x_2, ..., x_k]$ where $x_1 = x_k$. A *cycle* $\mu = [x_1, x_2, ..., x_k]$ is *simple* if the path $\mu' = [x_1, x_2, ..., x_{k-1}]$ is simple.

We denote $\Gamma^{-1} x = \{y \in X \mid (y, x) \in \Gamma\}$. The *inner degree* of an edge x is denoted by $id(x) = |\Gamma^{-1} x|$ and the *outer degree* by $ed(x) = |\Gamma x|$.

Definition 1.4 [3], [7]. A tree is a graph $G = (\Gamma, X)$ verifying the properties:

a) *G* is acyclic;
b) $\exists!\, x \in X$ such that $id(x) = 0$ and is called the *root* of the tree;
c) $\forall\, x \in X$, other then the root, has $id(x) = 1$.

We extend now de concepts form above and define Γ^2, Γ^3,... as follows:

$\Gamma^2 x = \Gamma(\Gamma x)$

$\Gamma^3 x = \Gamma(\Gamma(\Gamma x))$

…

$\Gamma^n x = \Gamma(\Gamma(...(\Gamma x))...)$

The *transitive closure* of Γ is the function

$\hat{\Gamma} : X \to 2^X$ defined as

$\hat{\Gamma}x = \{X\} \cup \Gamma x \cup \Gamma^2 x \cup ... \cup \Gamma^k x \cup ...$

Analogously we extend Γ^{-1}

$\Gamma^{-2} x = \Gamma^{-1}(\Gamma^{-1} x)$

$\Gamma^{-3} x = \Gamma^{-1}(\Gamma^{-1}(\Gamma^{-1} x))$

…

$\Gamma^{-n} x = \Gamma^{-1}(\Gamma^{-1}(...(\Gamma^{-1} x))...)$

The *transitive closure* for Γ^{-1} is the function

$\hat{\Gamma}^{-1} : X \to 2^X$ defined as

$\hat{\Gamma}^{-1}x = \{X\} \cup \Gamma^{-1} x \cup \Gamma^{-2} x \cup ... \cup \Gamma^{-k} x \cup ...$

2 Cohesion and coupling principles

The main principle in software package design is the maximal cohesion in any package and minimal coupling between packages.

For maximal cohesion, two classes are grouped together in a package if and only if they will be reused and modified together. Also we will take in mind the principle of abstract package stability and therefore abstract packages have to be more stable. Thus if we denote

$$x_{ij} = \begin{cases} 1 & \text{if they are reused together} \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

$$y_{ij} = \begin{cases} 1 & \text{if they are modified together} \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

$$a_{ij} = \begin{cases} 1 & \text{if the have similar abstraction} \\ 0 & \text{otherwise} \end{cases} \qquad (3)$$

$$z_{ij} = \begin{cases} 1 & \text{in the same package} \\ 0 & \text{otherwise} \end{cases} \qquad (4)$$

then $z_{ij} = a_{ij} x_{ij} y_{ij}$.

We denote: *Na* – the number of abstract classes; *Nc* – the number of concrete classes.

Under these conditions the degree of abstraction α_p of the package *p* is: $\alpha_p = Na/(Na + Nc)$.

We notice that under the above circumstances α_p is either 0 or 1. Thus after this kind of grouping we get homogeneous packages in respect to the abstraction degree. We call these packages *initial packages*.

Starting from a class hierarchy appropriate to the software system we would like to build, after a first grouping of these classes in packages, according to the matrix $u = (z_{ij})_{ij}$ from above, we obtain a graph. The edges of the graph will be these initial packages.

3 Packages in graphs

Definition 3.1 A *package whit main edge h* is a maximal subgraph (Γ', I) of (Γ, X) verifying the properties:

a) a) $h \in I$;

b) if $x \in I \Rightarrow h \in \hat{\Gamma}x$;

c) c) $I \setminus \{h\}$ is acyclic;

d) if $x \in I \setminus \{h\}$ then $\hat{\Gamma}x \subset I$.

Example 3.1 For the graph in Fig. 1. we have the packages $I(e) = \{e\}$, $I(f) = \{f, h, i, k\}$, $I(g) = \{g, j, t\}$ and $I(l) = \{l, m, n, o, s\}$.

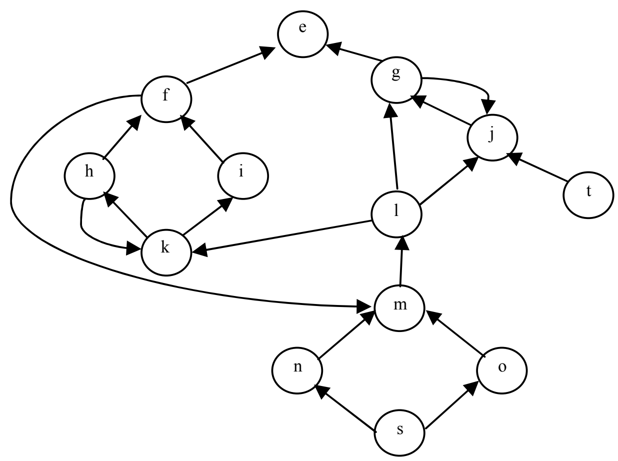

Fig. 1. Graph representation for example 3.1.

The following algorithm determines a package family of a given connected graph.

Algorithm 3.1

Input: A connected graph $G = (\Gamma, X)$

Output: The family of packages I_j, $j = \overline{1, p}$ who partitions graph *G*.

Method: The algorithm is founded on the conditions from the package definition.

01 **START** (*G*)

02 $i := 1$; $j := 1$; $l := 1$

03 **DO UNTIL** ($\Gamma^{-1\S}(I_1 \cup I_2 \cup ... \cup I_j) = \varnothing$)

04 **IF** ($j > 1$) **THEN**

```
05          l := min{m | x_m ∈ Γ^{-1§}(I_1 ∪ I_2 ∪ ... ∪ I_j)}
06          y_j := x_e
07             ELSE
08                   y_i := x_i
09             ENDIF
10             I_{j_1} := {y_i}
11             k := 1
12             DO UNTIL (I_{j_{k+1}} ≠ I_{j_k})
13               I_{j_{k+1}} := I_{j_k} ∪ {x ∈ Γ^{-1§} I_{j_k} | Γ̂x ⊆ I_{j_k}}
14               k := k + 1
15             ENDDO
16             I_j := I_{j_k}
17             j := j + 1; i := i + 1
18       ENDDO
19       p := j − 1
20 END
```

Lemma 3.1 The algorithm 3.1 determines a partitioning of the graph in packages.

Proof. First we show that $I_1, I_2, ..., I_p$ produced by the algorithm verify the conditions a)-d). Let (Γ', I_h) a subgraph. Since $I_{h_1} \subseteq I_{h_2} \subseteq ... \subseteq I_{h_n} \Rightarrow I_h = \bigcup_k I_{h_k}$. The first edge included in I_h is x_l or y_i chosen in step 04, 05 and 06. This will be the main edge of the package I_h. From step 13
$(\forall x \in I_{h_{k+1}} \Rightarrow x \in \Gamma^{-1} h_k) \Rightarrow (\forall x \in I_h \Rightarrow x \in \hat{\Gamma}^{-1} h)$
and so condition b). Let now $C = [x_1, x_2, ..., x_n x_{n-1}]$ be a cycle such that $x_i \in I_h$, $\forall i = \overline{1,n}$ and $h \notin C$. Let x_{i_0} be the first edge included in I_h at step 13. Since x_{i_0} was included $\Gamma x_{i_0} \subseteq I_{j_k}$. But from the condition $x_{i_0} \in C \Rightarrow x_{i_j} \in \Gamma_{x_{i_0}}$ and $x_{i_j} \notin I_{h_k}$ since x_{i_0} was the first node of the cycle included in I_h at step 13, so we have a contradiction and then $I \setminus \{h\}$ is acyclic, that means condition c). Also in step 13 a edge is included in $I_{h_{k+1}}$ with condition $\Gamma x \subseteq I_{h_k}$ ad for here condition d).

We shell notice now that there don't exist two subgraphs that satisfy the conditions a)-d), having different ends and common edges. Hence, let h_1 and h_2 be that two ends and $x \in I_{h_1} \cap I_{h_2}$, then there exists the path $[x, ..., h_1]$ and $[x, ..., h_2]$. But this means that there $\exists y_1, y_2$ so that $y_2 \in I_{h_2}$ and $y_1 \notin I_{h_2}$ such that $y_1 \in \Gamma y_2$. From here $\Gamma y_2 \not\subset I_{h_2}$ which means contradiction to condition d).

Let I_h be a packet produced by above algorithm and $I_h^{'}$ a package with the same initial node *h*. From the above we have $I_h \subseteq I_h^{'}$ und must show that $I_h = I_h^{'}$.

Assuming that $I_h \subseteq I_h^{'}$, then there exists a $x \in I_h^{'} \setminus I_h$, a edge so that $\Gamma x \subseteq I_h^{'}$, because the number of edges is finite and $\Gamma_X \subseteq I_h$. Since $I_h^{'}$ is produced by the algorithm, there exists a natural number n so that $I_h^{'} = I_{h_n}^{'} = I_{h_{n+1}}^{'}$. But $I_{h_{n+1}}^{'} = I_h^{'} \cup \{x \in \Gamma^{-1} I_{h_n}^{'} \mid \Gamma x \subseteq I_{h_n}\} \Rightarrow x \in I_{h_{n+1}}^{'} \Rightarrow I_{h_{n+1}}^{'} \neq I_{h_n}^{'}$ contradiction, so $I_h^{'} = I_h$.

The maximal condition results from the fact that $I_h = \bigcup_k I_{h_k}$, that means I_h is the union of all graphs who respect the conditions 1)-d).

We show now that $(I_1, I_2, ..., I_n)$ makes up a partition for *X*.

We assume $\exists x \in X$ such that $x \notin \bigcup_{k=1}^{n} I_k$, so $\Gamma^{-1}(I_1 \cup I_2 \cup ... \cup I_n) \neq \varnothing$ because the graph is connected and it exists a path from the initial edge to any edge of the graph, results that the algorithm never ends. Thus $X = \bigcup_{k=1}^{n} I_k$.

Lets show that $I_j \cap I_k = \varnothing$. Whit respect to the above, it is enough to show that there aren't two distinct packages with the same main edge *h*.

We assume by contradiction that exists I_h and $I_h^{'}$ so that $I_h \setminus I_h^{'} = \varnothing$, thus $I_h^{'} \subset I_h \cup I_h^{'}$. Since $I_h \cup I_h^{'}$ respects the conditions a)-d), $I_h^{'}$ is not the maximum that satisfies a)-d). Thus we have a contradiction and so $I_h = I_h^{'}$.

From the proof of the lemma results, in addition, that the partitioning of a graph in packages is unique.

Definition 3.2 Let $F = (\Gamma, X)$ be a graph and a package partition determined be algorithm 3.1. A *derivate graph* of *F* is a graph $I(F)$ defined as follows:

a) a) $I(F)$ has an edge for each package;
b) the main edge of $I(F)$ is the edge corresponding to the package that contains the main edge of *F*;
c) a vertex is drawn from package *I* to package *J* if and only if $I \neq J$ and it exists a vertex from an edge of package *I* to the end of package *J*.

The derivate graph of $I(F)$ is $I(I(F)) = I^2(F)$. Hence the number of edges of *F* is finite there exists an *n* such that $I^n(F) = I^{n+1}(F)$.

Definition 3.3 The *limit* of *F* is a graph $I^n(F)$ with the property $I^n(F) = I^{n+1}(F)$, denoted by F_n. From de uniqueness of graph partitioning in packages, results that this F_n exists and is unique.

Definition 3.4 Let $F = (\Gamma, X)$ be a graph, then if F_n has a single edge the graph *F* is called *reductive*, and if F_n has $N > 1$ edges the graph *F* is called *irreducible*.

A graph $G = (\Gamma, X)$ with *n* edges can be represented by a quadric matrix of *n* rows and *n* columns with elements of 0 and 1, called *adjacency matrix* of the graph. If the edges of the graph are $x_1, x_2, ..., x_n$ the adjacency matrix denoted $M = (m_{ij})$, $i, j = \overline{1, n}$ is defined as follows:

$$m_{ij} = \begin{cases} 1 & if \quad (x, y) \in \Gamma \\ 0 & otherwise \end{cases} \tag{5}$$

Thus we start from the initial graph $G = (\Gamma, X)$ whose edges correspond to the initial packages. These one we call *level* 0 *packages*. After applying algorithm 3.1 we obtain a graph $I(G)$ whose edges are *level* 1 *packages*. Generally speaking, the edges of graph $I^n(G)$ are called *level n packages*.

4 Conclusions

Let's see what's about module stability. Generally, if a class doesn't depend on any other class the class is sad to be *independent*. Also, a class is *responsible* if another class depends on her. Any change in such class produces a chain of changes in the dependent classes. Roughly speaking, in a software system there exists neither total independent classes nor classes without any responsibility.

The independency of a package is given by the number of classes of a package depending on classes from the inside of the package. Obviously our partitioning in packages is optimal since, in a level 1 package there exist a single dependent class, and generally, a single level *n* package in a level $n+1$ module depending on another package. Apparently the responsibility seems to be uncontrolled, but a closer analyze shows that any class of a level 1 package can depend on other classes, however the number of classes he depends on, is minimized due the fact that any class can depend only on classes that build the main edge of another package.

If for a given package *p* we denote:

C_a = the number of classes outside the package who depends on classes from his inside,
C_e = the number of classes outside the package on witch depend classes from the package,
I = the degree of instability of the package,
then $I = C_e / (C_a + C_e)$.
In our case

$$C_a = id(x) = |\Gamma^{-1} x| = \sum_{i \notin p} \sum_{j \in p} m_{i,j} \tag{6}$$

$$C_e = ed(x) = |\Gamma x| \sum_{i \in p} \sum_{j \notin p} m_{i,j} = \sum_{j \notin p} m_{h_p, j} \tag{7}$$

where is the main edge of *p*.

The main benefit of these partitioning in packages is the control of cyclical dependences between packages. We notice that the inner cycles of the packages and the cycles between packages are closed by the main edges of the packages, what makes it possible to simultaneously eliminate these cycles.

References:

[1] Emil M. Popa, *Inginerie software*, Editura „Alma Mater”, Sibiu, 2003, vol.I, vol.II.
[2] Vasile Crăciunean, *Proiectarea Translatoarelor*, Sibiu, Editura Alma Mater, 2002.
[3] Ioan Tomescu, *Combinatorica si teoria grafurilor*, Tipografia Universitatii din Bucuresti, 1978.
[4] Ioan Tomescu, *Graphs and Operations Research*, Bucharest University, Bucharest, 1978.
[5] Marius Iosifescu, *Lanțuri Markov finite şi aplicații*, Editura tehnică, 1977.
[6] Ernest G. Manes, Michael A. Arbib. *Algebraic Approaches to program semantics*, Springer – Verlag New York Berlin Heidelberg London Paris Tokyo,1986.
[7] Marvin Scheaffer, *A Mathematical Theory of Global Program Optimisation*, Prentice-Hall, 1973.
[8] Niklaus Wirth, Jurg Gutknecht, Project Oberon: *The Design of an Operating System and Compilers*, Hardcover, 1992.
[9] Gh. Păun, *Mecanisme generative ale proceselor economice*, Ed Tehnică, Bucureşti, 1988.
[10] Robert Morgan, C. Robert Morgan, *Building an Optimizing Compiler*, Paperback, 1997.
[11] Robert C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall, 2003.
[12] Daniel H. Steinberg , Daniel W. Palmer, *Extreme Software Engineering A Hands-On Approach*, Prentice Hall, 2004.