

Herleitung der Feature-Komponenten-Korrespondenz mittels Begriffsanalyse

Thomas Eisenbarth, Rainer Koschke, Daniel Simon

Bauhaus-Gruppe Stuttgart

Universität Stuttgart, Breitwiesenstr. 20-22, 70565 Stuttgart, Deutschland

{eisenbts, koschke, simondl}@informatik.uni-stuttgart.de

Zusammenfassung

1. Einführung

In den meisten Fällen entsteht eine Produktlinie aus einem erfolgreichen einzelnen System, das mehrere Einsatzgebiete erschließt. Es existiert also Code, somit ist Reengineering notwendig.

Einsatz von Reengineering in Produkt-Linien: Architekturwiedergewinnung, Komponenten-Reengineering,

Schnittstelle Produkt-Linien/Reengineering:

wichtiger Bestandteil: Feature-Komponent-Map;

Zweck: ...

Schnelles Feedback notwendig, weil Info an Produkt-Linien-Ingenieur zurückgegeben werden muss. Reengineering aber üblicherweise aufwendig. Technik notwendig, die schnelles Feedback ermöglicht. Weitere Analysen später notwendig (Aussagen über Qualität und Aufwand).

2. Übersicht

Feature sind realisierte Anforderungen. Entsprechend gibt es funktionale und nicht-funktionale Anforderungen. Viele nicht-funktionale Anforderungen, z.B. zeitliche Anforderungen, können nicht unbedingt einzelnen Komponenten zugeordnet werden. In manchen Fällen jedoch, können nicht-funktionale Anforderungen, wie z.B. Sicherheit, aus dem Code herausfaktoriert werden und durch einzelne Komponenten realisiert werden. Dabei müssen sich alle anderen Komponenten aber dieser und nur dieser Komponente bedienen.

Komponenten möglicherweise im Voraus nicht bekannt. Triviale Komponente: Funktion oder existierendes Modul. Herleitung logischer Module mittels unserer Techniken.

Prozess zum feature-basierten Reengineering für Produktlinien:

1. Feature-Lokation: Feature-Komponenten-Korrespondenz
2. Komponenten-Assessment
3. Komponenten-Extraktion bzw. Komponenten-Neuentwicklung (-einkauf)
4. Komponenten-Integration (Wrapping, Anpassung)

Feature-Komponenten-Korrespondenz gibt Übersicht, welche Features durch welche Komponenten realisiert werden. Herleitung: dynamische Analyse (Profiler) und Begriffsanalyse. Begriffsanalyse gibt auch Aufschluß über Abhängigkeiten zwischen Features und auch zwischen Komponenten (allerdings nur für die gegebene Implementierung).

3. Begriffsanalyse

Erläuterung Begriffsanalyse

Interpretation des Verbands

Die Begriffsanalyse basiert auf einer Relation R zwischen einer Menge von Objekten O und einer Menge von Attributen A , d.h. $R \subseteq O \times A$. Das Tripel $C = (O, A, R)$ wird **formaler Kontext** genannt. Für eine Menge von Objekten $O \subseteq O$ ist die Menge **gemeinsamer Attribute** definiert als:

$$\sigma(O) = \{a \in A \mid \forall (o \in O)(o, a) \in R\}$$

Entsprechend wird für eine Menge von Attributen $A \subseteq A$ die Menge ihrer **gemeinsamen Objekten** definiert als:

$$\tau(A) = \{o \in O \mid \forall (a \in A)(o, a) \in R\}$$

Als Beispiel diene die binäre Relation zwischen Komponenten (Objekte) und Features (Attribute) wie sie in Tabelle 1 angegeben ist. Eine Komponente (Objekt) O_i hat das Feature (Attribut) A_j , wenn O_i für Feature A_j ausgeführt werden muss. In diesem Falle steht ein **X** in Zeile i und Spalte j der Tabelle 1. Für diese Tabelle gelten zum Beispiel die beiden folgenden Gleichungen:

$$\sigma(\{O_1\}) = \{A_1, A_2\} \text{ und } \tau(\{A_7, A_8\}) = \{O_3, O_4\}$$

Tabelle 1. Beispielsrelation.

	A ₁	A ₂	A ₃	A ₄	A ₅	A ₆	A ₇	A ₈
O ₁	X	X						
O ₂			X	X	X			
O ₃			X	X		X	X	X
O ₄			X	X	X	X	X	X

Die zwei Funktionen σ und τ bilden eine **Galois-Verbindung**, d.h. ein Paar antimonotoner Funktionen:

$$O_1 \subseteq O_2 \Rightarrow \sigma(O_2) \subseteq \sigma(O_1) \text{ und } A_1 \subseteq A_2 \Rightarrow \tau(A_2) \subseteq \tau(A_1)$$

and both $\sigma \circ \tau$ and $\tau \circ \sigma$ are closure operators: e.g., $\sigma \circ \tau(O)$ determines the biggest set of objects that have the same attributes as O .

A pair (O, A) is called a **concept**, if $A = \sigma(O) \wedge O = \tau(A)$, i.e., all objects share all attributes. For a concept $c = (O, A)$, O is the **extent** of c , denoted by $extent(c)$, and A is the **intent** of c , denoted by $intent(c)$.

Informally, a concept corresponds to a maximal rectangle of filled table cells modulo row and column permutations. For example, Tabelle 2 contains the concepts for the relation in Tabelle 1.

Tabelle 2. Concepts for Tabelle 1.

C_1	$\{O_1, O_2, O_3, O_4\}, \emptyset$
C_2	$\{O_2, O_3, O_4\}, \{A_3, A_4\}$
C_3	$\{O_1\}, \{A_1, A_2\}$
C_4	$\{O_2, O_4\}, \{A_3, A_4, A_5\}$
C_5	$\{O_3, O_4\}, \{A_3, A_4, A_6, A_7, A_8\}$
C_6	$\{O_4\}, \{A_3, A_4, A_5, A_6, A_7, A_8\}$
C_7	$\emptyset, \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$

The set of all concepts of a given relation forms a partial order via $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2$ or, equivalently $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_1 \supseteq A_2$.

If $c_1 \leq c_2$, then c_1 is said to be a **subconcept** of c_2 and c_2 is a **superconcept** of c_1 . For example, $(\{O_2, O_4\}, \{A_3, A_4, A_5\}) \leq (\{O_2, O_3, O_4\}, \{A_3, A_4\})$ in Tabelle 1.

The sets of concepts and the partial order form a complete lattice, called **concept lattice**:

$$L(C) = \{(O, A) \in 2^O \times 2^A \mid A = \sigma(O) \wedge O = \tau(A)\}$$

In this lattice, the **infimum** (or **join**) of two concepts is computed by intersecting their extents:

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

Note that $A_1 \cup A_2 \subseteq \sigma(O_1 \cap O_2)$, as $O_1 \cap O_2$ has at least common attributes $A_1 \cup A_2$. Thus, an infimum describes the set of attributes common to two sets of objects.

Similarly, the **supremum** (or **meet**) is computed by intersecting the intents:

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$$

Again, $O_1 \cup O_2 \subseteq \tau(A_1 \cap A_2)$. Thus, a supremum describes a set of common objects which fit to two sets of attributes.

Graphically, the concept lattice for the example relation in Tabelle 1 can be represented as a graph whose nodes are the concepts in Tabelle 2 and whose edges denote the $<$ -

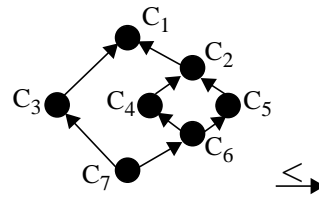


Abbildung 1. Example lattice.

The graph for a concept lattice would be difficult to read when each node showed its complete concepts, i.e., if the nodes C_i were replaced by their contents according to Tabelle 2. Fortunately, there is a better strategy for labelling nodes. A graph node in Abbildung 1(b) is labelled with attribute $a \in A$ if it is the largest concept having a in its intent; it is labelled with an object $o \in O$ if it is the smallest concept having o in its extent. The (unique) lattice element labelled with a is thus:

$$\mu(a) = \bigvee \{c \in L(C) \mid a \in intent(c)\}$$

The element labelled with o is

$$\gamma(o) = \bigwedge \{c \in L(C) \mid o \in extent(c)\} \quad (1)$$

The equivalent graph for Abbildung 1(a) using this labelling strategy is shown in Abbildung 1(b). A concept represented by a node N in this graph consists of all objects at and below N and of all attributes at and above N . For example, the concept labelled with O_2 and A_5 in Abbildung 1(b) is $(\{O_2, O_4\}, \{A_3, A_4, A_5\})$.

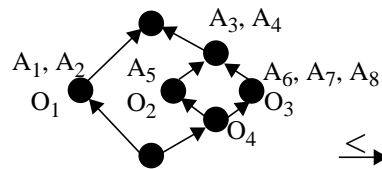


Abbildung 2. Example lattice.

4. Implementierung

Nahezu trivial: gcc, gprof, perl-Script, GraphLet

5. Fallstudie

Xfig-Beispiel, interessante Erkenntnisse

6. Verwandte Gebiete (Related Research)

Statische Feature-Lokation: Chen und Rajlich [2]

Dynamisch: Deprez und Lakhotia [3]

7. Schlussfolgerungen

Referenzen

- [1] Bayer, J., Girard, J.-F., Würthner, M., Apel, M., and DeBaud, J.-M., 'Transitioning Legacy Assets - a Product Line Approach', *Proceedings of the SIGSOFT Foundations of Software Engineering*, Toulouse, pp. 446-463, Association of Computing Machinery, 1999.
- [2] Chen, K. und Rajlich, V., 'Case Study of Feature Location Using Dependence Graph', Proc. of the 8th Int. Workshop on Program Comprehension, pp. 241-249, June 10-11, Limerick, Irland, IEEE Computer Society Press, 2000.
- [3] Deprez, J.-C. und Lakhota, A., 'A Formalism to Automate Mapping from Program Features to Code', Proc. of the 8th Int. Workshop on Program Comprehension, pp. 69-78, June 10-11, Limerick, Irland, IEEE Computer Society Press, 2000.
- [4] Lindig, C. and Snelting, G., 'Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis', Proc. of the Int. Conference on Software Engineering, pp. 349-359, Boston, 1997.
- [5] Lindig, C., Concepts, <ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/misc>
- [6] Koschke, R., 'Atomic Architectural Component Recovery for Program Understanding and Evolution', Dissertation, Institut für Informatik, Universität Stuttgart, <http://www.informatik.uni-stuttgart.de/ifi/ps/rainer/thesis>, 2000.