Reverse Engineering Techniken

Juri Weber
Universität Stuttgart
Institut für Softwaretechnologie
Universitätsstraße 38
D-70569 Stuttgart
weberji@studi.informatik.uni-stuttgart.de

Zusammenfassung

Produktlinien bieten eine Möglichkeit in kürzeren Ent - wicklungszeiten relativ hochwertige Software zu entwickeln. Software Produktlinien werden nur in seltensten Fällen auf „grüner Wiese" entwickelt, meist existieren bereits Softwaresysteme bzw. Produktvarianten, die in einer Produktlinie verwaltet werden sollen. Es ist dabei sehr lukrativ Teile der Altsoftware oder so genannte Features in der Produktlinie wiederzuverwenden, um auf diese Weise wertvolle Investitionen zu schützen. Dieser Artikel diskutiert die möglichen Einführungsstrategien von Software Produktlinien und erläutert dabei wie einzelne Features in der Altsoftware lokalisiert werden können.

1. Einführung

Die Einführung einer Produktlinie bietet einige Vorteile, wie geringerer Aufwand bei der Verwaltung von Produkten, robuste Produktarchitektur, schnellere Modifikation von Produkten zu niedrigeren Kosten [1]. Allerdings ist die Einführung einer Produktlinie immer mit hohen Kosten verbunden. Diese Kosten lassen sich in vielen Fällen jedoch auf ein bestimmtes Niveau senken. Eine der Möglichkeiten ist die Wiederverwendung von Komponenten der Altsoftware. Für die Entwicklung dieser Komponenten wurde in aller Regel ein hoher Aufwand betrieben und falls die betroffene Komponente in die Produktlinie übernommen wird, kann dieser gespart werden [1]. Eine wichtige Frage ist, welche Teile der Altsoftware wiederverwendet werden können? Nicht nur der Quellcode, sondern auch Teile der Spezifikation, des Entwurfs, darüber hinaus Testfälle, unterschiedliche Werkzeuge und Benutzerschnittstellen können als wieder verwendbare Komponenten angesehen und in die Produktlinie aufgenommen werden[2]. Allerdings müssen bei der Wiederverwendung des Quellcode einige Dinge beachtet werden:

- die Dokumentation des Quellcodes ist in vielen Fällen gar nicht mehr verfügbar und
- die Architektur der Software wurde während eines langen Wartungsprozesses mehrfach verändert.

Somit wird die Altsoftware meistens schlecht strukturiert und weist zwischen den Modulen eine zu hohe Kopplung auf. Es besteht die Möglichkeit, dass keine wiederverwendbaren Komponenten gefunden werden, trotz hoher Reengineeringkosten[10]. Laut Bergey et al. [11] ist das Herauslösen und Wiederverwenden von Komponenten dennoch einer kompletten Neuentwicklung vorzuziehen.

Damit Komponenten aus vorhandenen Altsoftware mit größtmöglichen Nutzen in die Produktlinie aufgenommen werden können, sollte bereits zum Start der Entwicklung der Produktlinie die Wiederverwendung der Komponenten der Altsoftware berücksichtigt werden. Bei der Entwicklung einer Produktlinie muss die Entscheidung getroffen werden, auf welche Art die Produktlinie entstehen soll. Bei der revolutionären Entwicklung wird die Wartung der Altsysteme gestoppt und die Produktlinie neu entwickelt. Die Altsoftware wird nach wiederverwendbaren Komponenten untersucht, die anschließend in die Produktlinie integriert werden können. Bei der evolutionären Entwicklung wird die Altsoftware Schritt für Schritt in Produktlinie überführt. Der Rest des Artikels ist wie folgt strukturiert. In Kapitel 2 und 4 werden der revolutionäre und der evolutionäre Prozess beispielhaft beschrieben. In Kapitel 3 werden mit der Suche auf dem Abhängigkeitsgraphen, der Software Reconnaissance und der Feature Location drei Techniken zur Lokalisierung von Features vorgestellt. Im letzten Kapitel schließt ein Fazit die Ausarbeitung ab.

2. Revolutionäre Entwicklung von Produktlinien

Als Antwort auf die Frage nach systematischen Vor gehen bei der Entwicklung von Produktlinien stellte das SEI die Methode Mining Architectures for Productline (MAP) vor[7]. Die MAP-Methode erlaubt es den Ent-

wicklern wiederverwendbare Architekturen der Altsoftware zu identifizieren. MAP-Methode besteht aus vordefinierten Phasen, die unter der Leitung eines MAP-Teams nacheinander ausgeführt werden. Eine Phase umfasst dabei Eingabedaten, Aktivitäten auf diesen und erzeugte Ausgabedaten. Nachfolgend die

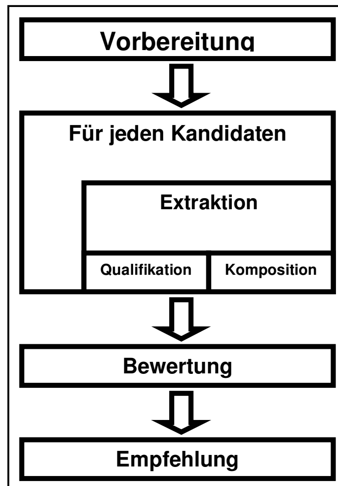

Abbildung 1. Phasen der MAP-Methode

Phasen des MAP (Abbildung 1):

1. Vorbereitung

Als Eingabe für die Vorbereitungsphase dienen Kenntnisse über die PL und die Organisation. Die Aktivitäten sind auf die Bestimmung von Produkten-Kandidaten gerichtet. Diese sollten die vorhandenen Produkte z. B. bezüglich unterschiedlichen Kunden, Plattformen, Protokollen oder Eigenschaften repräsentieren. Die identifizierten Produkte bilden die Ausgabedaten dieser Phase und repräsentieren die Eingabedaten für nachfolgende Phasen. Die nachfolgenden drei Phasen werden auf jedes Produkt angewandt.

2. Extraktion

Die Aktivitäten der Extraktionsphase dienen der Ausbildung eines Implementierungsmodells. Zu Beginn dieser Phase präsentiert der Systemarchitekt dem MAP-Team das System, Software und Entwicklungsumgebung wie Werkzeuge und Compiler. Der Architekt stellt den Quellcode und die vorhandene Dokumentation wie Spezifikation, Schnittstellenbeschreibung, usw. bereit. Das Implementierungsmodell ist eine Menge an Relationen unter den Elementen wie Funktionen, Dateien und Verzeichnisse. Die Relationen werden von Lexern und Parser generiert und beschreiben wie die Elemente zueinander in Beziehung stehen. Dieses Implementierungsmodell dient als Eingabe für die Qualifikations- und Kompositionsphase.

3. Komposition

Das Implementierungsmodell repräsentiert die Eingabedaten dieser Phase. Die Kompositionsphase liefert die Komponenten-Sicht auf das System. Sie bildet die Schlüsselphase für die Erfassung von Strukturen, die in der Bewertungsphase bewertet werden. Die Hauptaktivität der Komposition ist die Zuordnung von Elementen und Komponenten. Das Resultat dieser Phase sind Komponenten-Sichten, die die Funktionalität der Komponenten und ihre Beziehungen untereinander spiegeln; es bildet die Eingabe für die Bewertungsphase.

4. Qualifikation

Auch in dieser Phase ist das Implementierungsmodell die Eingabe. Nach dem Ablauf der Qualifikation liegen als Ausgabe Architektur, Attributen und Designmuster vor. Bekannte Architektur- und Qualitätseigenschaften werden auf das System und seine Komponenten abgebildet. Es entstehen weitere Sichten wie Nebenläufigkeit und Ausführung. Die Kompositions- und Qualifikationsphasen bestimmen die Architektur der Software. Die in den beiden Phasen erzeugte Ausgabedaten fließen als Eingabe in die nächste Phase.

5. Bewertung

Architektur-Kandidaten werden hinsichtlich ihren Gemeinsamkeiten, Variabilitäten und anderen Eigenschaften bewertet. Die Bewertung der Gemeinsamkeiten und Variabilitäten der Produkt-Kandidaten liegt nach dem Phasenablauf als Ergebnis vor.

6. Empfehlung

In der Empfehlung wird vom MAP-Team durch Auswertung der Ergebnisse der Bewertungsphase ein Gutachten erstellt. Dieses Gutachten wird anschließend vom MAP-Team repräsentiert. Es werden Vorschläge bezüglich der Aufnahme von Kandidaten in die PL gemacht.

Nach der Anwendung des MAP liegt Information vor, welche Architekturen in die Produktlinie übernommen werden können.

Nicht immer ist man aber an der Wiederverwendung auf der Architekturebene interessiert. Viel interessanter ist in diesem Kontext sicherlich die Komponentenebene. Auch dafür hat die SEI eine Methode entwickelt, die Options Analysis for Reengineering (OAR)[1]. Die OAR berücksichtigt, dass bei der Entwicklung einer Produktlinie unterschiedliche Interessengruppen wie das Management oder Entwickler beteiligt sind und dass jede dieser Gruppen eigene Anforderungen an die Produktlinie hat. Das Management zum Beispiel strebt die größtmögliche Kostenersparnis an, währen die Entwickler die Entwicklung möglichst einfach gestalten wollen. Es ist wichtig Ziele und Erwartungen dieser Gruppen aufeinander abzustimmen, was mit dem Einsatz von OAR unterstützt wird.

Die OAR-Methode besteht aus 5 Aktivitäten (siehe Abbildung 2):

1. Überblick verschaffen

Die erste Aktivität von OAR dient dazu, alle teilnehmenden Interessengruppen in den Prozess und in die Entscheidungsfindung einzubeziehen. Die verschiede-

nen Interessengruppen werden vom OAR-Team befragt. Die Ziele und Erwartungen werden festgestellt und aufeinander abgestimmt. Zusätzlich werden die Altsysteme auf mögliche Komponenten untersucht.

2. Aufnahme der Kandidaten

Während dieser Aktivität identifizieren die Mitglieder des OAR-Teams die Anforderungen der Produktlinie an die Komponenten der Altsoftware. Alle Komponenten der Altsysteme, die die Anforderungen erfüllen, werden notiert. Das Resultat dieser Aktivität ist somit eine Menge an potentiellen Komponentenkandidaten.

3. Analyse der Kandidaten

Im dritten Schritt werden die Kandidaten daraufhin analysiert, wie sie in die Produktlinie eingearbeitet werden können. Die Kandidaten werden dabei in zwei Kategorien eingeteilt. Black-Box Komponenten müssen entweder nur gekapselt werden, oder können unverändert in der Produktlinie eingesetzt werden. Testfälle, Teile der Spezifikation, des Entwurfs, Werkzeuge können ebenso zu dieser Kategorie gehören wie einzelne Komponenten des Quellcodes. White-Box Komponenten müssen durch Reengineering-Maßnahmen bearbeitet werden um in der Produktlinie wieder verwendet werden zu können. Die Kosten, die Risiken und der Aufwand der Bearbeitung der Kandidaten werden ebenfalls in diesem Schritt abgeschätzt und die Kosten einer Neuentwicklung erhoben.

4. Planung der Alternativen

Zu diesem Zeitpunkt stehen für jede benötigte Komponente mehrere Alternativen zur Auswahl. Im vierten Schritt werden mögliche Kombinationen von Kandidaten erarbeitet. Diese Kombinationen enthalten für alle zukünftigen Komponenten der Produktlinie jeweils einen Kandidaten. Die unterschiedlichen Kombinationen werden bezüglich Ihrer Kosten und Risiken bewertet.

5. Wahl einer Alternative

Abschließend werden die Entscheidungskriterien für die Wahl der Kandidatenkombination festgelegt und schließlich eine Kombination ausgewählt. Die Entscheidung wird protokolliert und den verschiedenen Interessengruppen präsentiert. In jedem Schritt werden zusätzlich die getroffenen Entscheidungen geprüft und die Zeitplanung der Produktlinienentwicklung falls notwendig aktualisiert. Die Anwendung von OAR muss nur noch um Reengineering-Techniken ergänzt werden, deren Aufgabe es ist die genaue Position von wiederverwendbaren Features im Quellcode zu bestimmen.

Im nächsten Kapitel werden drei Techniken der Feature-Lokalisierung vorgestellt.

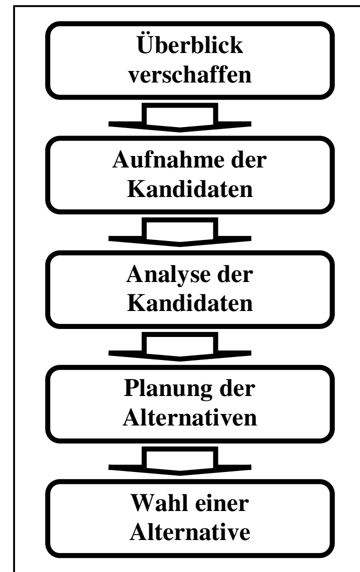

Abbildung 2. OAR Aktivitäten

3. Feature Lokalisierung

Bisher haben wir für die Wiederverwendung neben dem eigentlichen Quellcode auch Dokumente wie Spezifikation oder Testdaten betrachtet. Im folgenden werden wir uns auf den Quellcode beschränken. Der Begriff Feature steht hier für eine Funktionalität eines Programms, die vom Benutzer manipuliert werden kann. Die Hauptaufgabe bei der Beschaffung wieder verwendbarer Komponenten besteht darin, die gewünschten Features im Code der Altsoftware wiederzufinden und so zu restrukturieren, dass die Features innerhalb der Produktlinienarchitektur wiederverwendet werden können. Für die Lösung dieser Aufgabe können Reverse-Engineering-Techniken eingesetzt werden. Die Schwierigkeit bei der Herleitung der Feature-Komponenten-Korrespondenz liegt unter anderem an den unterschiedlichen Abstraktionsebenen. Die geforderte Funktionalität an eine Komponente wird auf der Anforderungsebene festgelegt. (Die neue Produktlinie soll das Feature XY beinhalten). Die zugehörige Komponente muss aber auf der Implementierungsebene gefunden und eventuell abgeändert werden. Ein weiteres Problem besteht darin, dass es in größeren System unmöglich ist, wirtschaftlich, d.h. mit akzeptablen Kosten ein gesamtes Softwaresystem zu verstehen[8] [10]. Es muss also nach Techniken gesucht werden, mit Hilfe derer einzelne Features im Code lokalisiert werden, ohne den Rest des Programms zu verstehen.

3.1 Suche auf Abhängigkeitsgraphen

Eine Möglichkeit zur Lokalisierung von Features wurde vom Vaćlav Rajlich entwickelt [3]. Es handelt sich hier um eine statische, rechnergestützte Analyse auf dem PDG (Program Dependency Graph) [9]. Von Anwender dieser Methode werden Kenntnisse des zu un-

tersuchenden Systems erwartet. Der PDG ist eine Abstraktion einer Routine (Funktion, Methode, Prozedur), bei der die einzelnen Anweisungen oder Anweisungsblöcke durch Knoten repräsentiert werden. Datenflusskanten stellen den möglichen Datenfluss zwischen den Knoten dar. Kontrollflusskanten zeigen die möglichen Pfade der Routine. Die Verknüpfung aller PDGs eines Programms bilden den SDG (System Dependency Graph). Für die Featuresuche wird eine Variante des SDG, der so genannte Abstract System Dependency Graph (ASDG, Abhängigkeitsgraph) eingesetzt. Der ASDG stellt eine Teilmenge des SDG dar. Beim ASDG repräsentieren die Knoten ganze Programmeinheiten, wie zum Beispiel Routinen und globale Variablen. In Abbildung 3 ist ein Beispiel für einen ASDG mit dem zugehörigen C-Quellcode dargestellt. Abbildung 4 stellt den Ablauf der Featuresuche dar. Während der Suche nach einem Feature wird ein Suchgraph aufgebaut. Am Ende der Suche sind alle Knoten der Komponenten enthalten, die das gesuchte Feature implementieren. Der Suchgraph stellt somit immer eine Teilmenge des ASDG dar. Zu Beginn der Suche muss der Programmierer einen passenden Startknoten aus dem ASDG wählen. Falls der Programmierer nicht mit dem System vertraut ist, kann standardmäßig die Routine main() (in C-Programmen) gewählt werden. Als nächstes wählt der Programmierer einen Knoten aus der Menge aller Nachbarn des Suchgraphen im ASDG aus. Dabei kann er verschiedene Strategien verfolgen:

- Kontrollfluss: Bei dieser Strategie betrachtet der Programmierer rekursiv alle Knoten, die entweder durch den betrachteten Knoten aufgerufen werden (top-down) oder den Knoten aufrufen (bottom-up)

Datenfluss: Falls das Feature von dem Inhalt bestimmter Variablen abhängt, kann die Auswahl von Knoten anhand der Datenflusskanten im ASDG abhängig gemacht werden. Die Entscheidung, welche Komponente als nächstes besucht werden soll, trifft der Programmierer bei jeder dieser Strategien aufgrund des Quellcodes des Knotens, seiner Erfahrung mit dem System und der eventuell vorhandenen Dokumentation. Für jeden besuchten Knoten entscheidet der Programmierer, ob der vom Knoten repräsentierte Code an der Realisierung des Features beteiligt ist. Ist die Komponente relevant, wird der Suchgraph um den Knoten erweitert. Dieser Vorgang wird solange wiederholt, bis alle Komponenten gefunden wurden, die das gesuchte Feature realisieren. Eine Unterstützung durch Werkzeuge ist auf die Erstellung des ASDG, sowie auf die Darstellung und Anordnung des Suchgraphen begrenzt. Für die effiziente Durchführung des Verfahrens müssen einige Bedingungen erfüllt werden:

- Zum einem sollte der Benutzer mit dem System vertraut sein
- Und zum anderem sollten Architekturbeschreibungen des Systems oder sonstige Dokumente über das System vorhanden sein.

```
int g;              void foo2(){
main(){               int a = 2;
int a = 2;            foo4(a);
a = foo1(a);          a = foo5();
foo2();             }
foo3();
}                   void foo4(int a){
                      g += a;
int foo1(int a){ }
  g = 3;
return g + a;       int foo5(){
}                     return g;
                    }
void foo3(){}
```

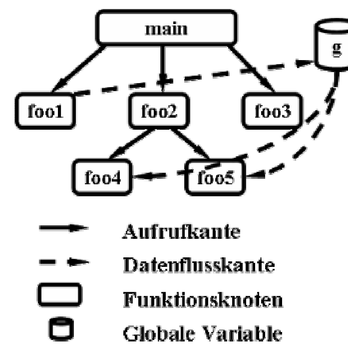

Abbildung 3. Der ASDG mit zugehörigem C-Quellcode

3.2 Software Reconnaissance

Mit seiner Software Reconnaissance hat Norman Wilde eine weitere Methode für die Lokalisierung der Features im Quellcode entwickelt [8]. Diese Methode verwendet im Gegensatz zu Suche auf dem Abhängigkeitsgraphen einen dynamischen Ansatz. Für die Lokalisierung der Features werden zwei Klassen von Testfällen entwickelt:

- Testfälle, die das gesuchte Feature *f* auf alle möglichen Arten ausführen und
- Testfälle, die das gesuchte Feature *f* nicht ausführen.

Nach der Vorbereitung der Testfälle wird der Quellcode auf eine spezielle Weise instrumentiert. Diese Instrumentierung gibt später Aufschluss darüber, welche Komponenten des Quellcodes bei den Testfällen ausgeführt wurden. Diese Komponenten werden wie folgt in Mengen organisiert:

CCOMPS *common components* sind Komponenten, die in allen Testfällen aufgerufen wurde, unabhängig ob *f* ausgeführt wurde oder nicht.

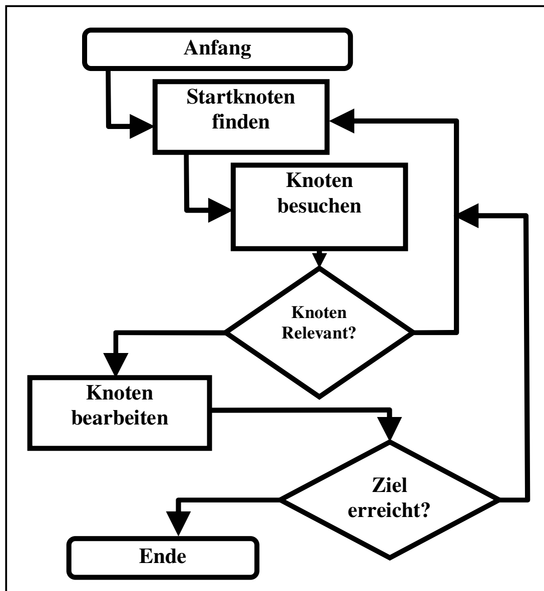

Abbildung 4. Ablauf einer Featuresuche auf dem Abhängigkeitsgraph

ICOMPS *potentially involved components* sind die Komponenten, die mindestens einmal innerhalb der Testfälle aufgerufen wurden, die *f* ausführen. In der Menge ICOMP sind in jedem Fall alle Komponenten des Features enthalten. Jedoch sind darin mit hoher Wahrscheinlichkeit noch Komponenten enthalten, die nicht an der Realisierung des Features beteiligt sind.

IICOMPS *indispensably involved components* sind die Komponenten, die in allen Testfällen aufgerufen wurden, die *f* ausführen. Diese Menge beschreibt das gesuchte Feature schon recht genau. Es handelt sich jedoch meist noch um einen recht großen Ausschnitt des Programms.

UCOMPS *uniquely involved components* ist eine noch exaktere Eingrenzung der beteiligten Komponenten. UCOMPS wird gebildet indem man von der Menge ICOMPS alle Komponenten entfernt, die in Testfällen aufgerufen werden, die *f* nicht beinhalten. Die Menge UCOMPS stellt in aller Regel eine leicht überschaubare Menge dar und bietet daher einen Einstieg in die Suche nach dem Feature.

3.3 Feature Location

Diese Technik basiert auf der Herleitung der Feature-Komponenten-Korrespondenz mit Hilfe der Begriffsanalyse [4] und vereint die Ansätze der beiden bereits erwähnten Techniken, den statischen der Suche auf dem Abhängigkeitsgraph und den dynamischen der Software Reconnaissance [5]. Die Begriffsanalyse wird in vielen Bereichen der Informatik eingesetzt. Feature Location ermöglicht die Suche nach mehreren Features gleichzeitig und bietet Möglichkeiten zur Bestimmung der Abhängigkeiten dieser Features untereinander. Wie bei der Software Reconnaissance werden durch Testfälle, die gleichzeitig mehrere Features abdecken, die aufgerufenen Komponenten im Quellcode bestimmt. Danach werden die Features und die zugehörigen Komponenten in einer Relationentabelle (siehe Tabelle 1) zusammengeführt. Das Paar (Routine *R* und Feature *F*) ist in der Relation, falls *R* bei der Ausführung von *F* verwendet

Tabelle 1. Relationentabelle

	R1	R2	R3	R4	R5	R6	R7	R8
F1	X	X						
F2			X	X	X			
F3			X	X		X	X	X
F4			X	X	X	X	X	X

Tabelle 2. Konzepttabelle

C1	{F1, F2, F3, F4}, ∅
C2	{F2, F3, F4}, {R3, R4}
C3	{F1}, {R1, R2}
C4	{F2, F4}, {R3, R4, R5}
C5	{F3, F4}, {R3, R4, R6, R7, R8}
C6	{F4}, {R3, R4, R5, R6, R7, R8}
C7	∅ {R1, R2, R3, R4, R5, R6, R7, R8}

wird. Ein Paar (O, A) zwischen einer Menge von Routinen O und einer Menge von Features A, bei der alle Features aus A alle Routinen aus O aufrufen, wird als Konzept bezeichnet, Tabelle 2 enthält alle Konzepte aus der Relationentabelle. Somit ist ein Konzept in der Tabelle ein möglichst großes Rechteck um eine Menge von Kreuzen. Die Menge aller in ihrer Halbordnung angeordneten Konzepte definieren einen Konzeptverband. Der Konzeptgraph der Beispieltabelle ist in Abbildung 5 dargestellt. Das Konzept links enthält also alle Routinen, das an der rechten Seite alle Features. Ein Konzept enthält somit alle Routinen der kleineren Konzepte als Teilmenge. Durch das unterschiedliche Vorkommen der Routinen im Graph können Aussagen getroffen werden, welche Routinen gemeinsam für mehrere Features genutzt werden und welche für ein Feature spezifisch sind.

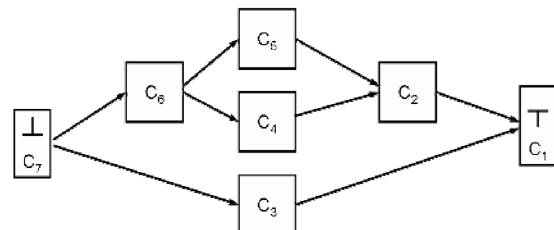

Abbildung 5. Konzeptgraph der Tabelle 1

4. Evolutionäre Einführung

Mit den beiden Techniken MAP und OAR wurde die revolutionäre Einführung vorgestellt. Die Wartung der Altsoftware wird eingestellt und die Produktlinie wird aufgebaut, wobei Teile der Altsoftware wiederverwendet werden können. Diese Art der Einführung bietet große Entwurfsfreiheit. Aber auch Nachteile müssen bei dieser Entwicklungsstrategie beachtet werden. Die Entwickler müssen bereits von Projektbeginn Kenntnisse über Produktlinien aufweisen, des Weiteren ist das Risiko des Scheiterns erhöht, da eventuell erst bei weit fortgeschrittener Entwicklung festgestellt werden kann, ob alle Anforderungen erfüllt werden können. Diese Überlegungen sprechen oft gegen eine revolutionäre Entwicklung von Software Produktlinien. Als Alternative der revolutionären Einführung wurde an der Universität Stuttgart ein Prozessmodell entwickelt, welches erlaubt, Altsysteme Schritt für Schritt durch Restrukturierungsmaßnahmen in eine Produktlinie zu überführen [6] [10]. Diese Strategie bietet einige Vorteile:

- Die Entwickler lernen die Konzepte der Produktlinie während der Entwicklung besser kennen und bekommen so ein Gesamtbild
- Das Risiko des Scheiterns wird minimiert, da die Entwicklung zu jeder Zeit gestoppt werden kann
- Die Entwickler kennen die Produkte, so können sie bereits zu Beginn der Einführung eine genaue Machbarkeitsanalyse der Anforderungen durchführen.

Zusätzlich kann die Wartung der Altsoftware in diesen Prozess integriert werden. Der Evolutionärer Prozess besteht aus mehreren Aktivitäten. Die erste Aktivität ist die Identifizierung der im Produkt enthaltenen Features. Unter Umständen können in den Produkten sehr viele Features enthalten sein. In diesem Fall wird es notwendig diese nach ihrer Dringlichkeit zu priorisieren. In der darauf folgenden Aktivität schätzt der Systemarchitekt den Aufwand und die Kosten der Änderung der untersuchten Features. Falls die geschätzten Werte innerhalb vorgegebenen Grenzen liegen, wird die Änderung in der nächsten Aktivität durchgeführt. Der gesamte Prozess ist iterativ angelegt, d. h. die Features können beliebig oft geändert werden. Zumindest solange, bis sie den Anforderungen an die Produktlinie entsprechen. In jeder Iteration ist die Möglichkeit gegeben den Prozess anzuhalten und die Produkte traditioell weiter zu entwickeln. Die Vorteile durch bereits durchgeführte Änderungen bleiben erhalten.

5. Fazit

Produktlinien bieten Möglichkeiten zur Wiederverwendung von Komponenten der Altsoftware und somit effektiven Schutz der Investitionen. Allerdings will eine Einführungsstrategie sorgfältig überlegt werden. Es muss durch Einbeziehung von allen beteiligten Interessengruppen entschieden werden, welche Strategie unter gegebenen Umständen die richtige ist. Dabei gilt es die Vor- und Nachteile der beiden Methoden zu berücksichtigen. Der Erfolg des gewählten Einführungsvorhabens kann durch die Auswahl geeigneter Reengineering-Techniken begünstigt werden. Je nach dem, ob die Analyse statisch oder dynamisch durchgeführt werden soll, werden entweder die Sucher auf dem PDG oder die Software Reconnaissance eingesetzt. Eine viel versprechende Methode ist aber auch die Feature Location, da sie die Vorteile der beiden vereint und darüber hinaus weitere Vorteile bietet.

Literatur

[1] J. Bergey, L. O'Brien, und D. Smith. Options Analysis for Reengineering (OAR): A Method for Mining Legacy Assets. Technical Report CMU/SEI-2001-TN-013, Jun. 2001.

[2] J. Bergey, L. O'Brien, und D. Smith. Mining Existing Software Assets for Software Product Lines. Technical Report CMU/SEI-2000-TN-008, Mai 2000.

[3] K. Chen und V. Rajlich. Case Study of Feature Location Using Dependence Graph. Aus *Proceedings of the 8th International Workshop on Program Comprehension*, Seiten 241–249, Limerick, Ireland, Jun. 2000. IEEE Computer Society Press.

[4] T. Eisenbarth, R. Koschke, und D. Simon. Herleitung der Feature-Komponenten-Korrespondenz mittels Begriffsanalyse. Aus *Proceedings of the 1. Deutscher Software-Produktlinien Workshop*, Seiten 63–68, Kaiserslautern, Germany, Nov. 2000.

[5] T. Eisenbarth, R. Koschke, und D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. Aus *Proceedings of the International Conference on Software Maintenance*, Seiten 602–611, Florence, Italy, Nov. 2001. IEEE Computer Society Press.

[6] T. Eisenbarth und D. Simon. Guiding Feature Asset Mining for Software Product Line Development. Aus *Proceedingsof the International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing*, Seiten 1–4, Erfurt, Germany, Sept. 2001. Fraunhofer IESE.

[7] L. O'Brien, D. Smith. MAP and OAR Methods: Techniques for Developing Core Assets for Software Product Lines from Existing Assets. Technical Note CMU/SEI-2002-TN-007

[8] N. Wilde und M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7:49–62, Jan. 1995.

[9] K. Ottenstein und L. Ottenstein. The Program Dependency Graph in a Software Development Environment. Aus *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium in Practical Software Developement Environments*, 1984.

[10] D. Simon und T. Eisenbarth. Evolutionary Introduction of Software Product Lines. Aus *Proceedings of the 2nd Software Product Line Conference*, San Diego, CA, USA, Aug. 2002.

[11] N. Weiderman, J. Bergey, D. Smith, und S. Tilley. Can Legacy Systems Beget Product Lines? *Lecture Notes in Computer Science*, 1429, 1998.