Wie treibt man die Herde wieder sicher zusammen?

Architekturanalyse für Software-Varianten

Professor Dr. Rainer Koschke, Universität Bremen
Thomas Eisenbarth, Axivion GmbH

Wenn man sehr variantenintensive Software-Produkte zu betreuen hat, bietet sich eine Entwicklung in einer sogenannten *Software-Produktlinie (SPL)* an. Eine Produktlinie ist eine Familie von Programmen, die eine große Anzahl gleicher Anforderungen erfüllt, von denen aber jedes einzelne Programm nicht genau das Gleiche tut wie die anderen – sei es weil es etwas andere funktionale Anforderungen hat oder weil es etwas andere nicht-funktionale Eigenschaften aufweist (z.B. andere Algorithmen mit unterschiedlichem Ressourcenbedarf implementiert). Manche Autoren bezeichnen SPL deshalb auch als *Software-Produktfamilien*.

In der Praxis wird bei der Entwicklung eines neuen Produkts häufig ein Ansatz favorisiert, der ein bestehendes Produkt der Software-Produktlinie komplett kopiert und so abändert und anpasst, dass das konkrete Problem gelöst wird. Diesen Ansatz nennt man "*Clone-and-Own*-Vorgehen".

Der Vorteil des Clone-and-Own-Vorgehens in der Praxis ist die höhere Entwicklungseffizienz. Man hat einen schnelleren Start, mehr Unabhängigkeit und der Ansatz ist auch recht einfach anzuwenden.

Die Nachteile des Clone-and-Own-Vorgehens sind jedoch beträchtlich. Es entsteht zusätzlicher Aufwand bei der Änderung. Weil der Code durch das Kopieren hochgradig redundant ist, müssen für eine konsistente Änderung gleichartige Änderungen an vielen Stellen des Codes gemacht werden. Inkonsistente Änderungen können zu Fehlern und Integrationsproblemen führen. Der logisch gleiche Code muss unter Umständen mehrfach getestet werden.

Der Clone-and-Own-Ansatz stellt ein kurzfristiges Denken dar. Es ist eine mangelhafte Form von Wiederverwendung, die nicht selten aus der Unkenntnis und dem Mangel an Ressourcen für bessere SPL-Praktiken resultiert. Der Clone-and-Own-Ansatz ist nicht selten auch ein Indiz für die mangelhafte Führung von Software-Entwicklungsprojekten. Es gibt keine Infrastruktur für Wiederverwendung und auch keine definierten Prozesse und Rollen dafür. Auch der Grad der Wiederwendung und ihre Wirkung auf Kosten, Entwicklungsgeschwindigkeit und Korrektheit werden nicht gemessen und somit vernachlässigt.

Der Clone-and-Own-Ansatz führt zu einer Menge sehr ähnlicher Implementierungen, die man alle parallel warten muss. Die Nachteile, die daraus erwachsen, müssen kompensiert werden, um noch Herr der Lage bleiben zu können. In dieser Situation bietet sich eine Klonanalyse an, um gleiche Teile, Variationspunkte und optionale Anteile zu identifizieren. Eine Klonanalyse sucht automatisiert nach gleichen Code-Teilen. Sie kann verwendet werden, um kopierte Dateien zueinander in Beziehung zu setzen und deren Unterschiede und Gemeinsamkeiten zu quantifizieren. Folgende Funktionen für jeweils zwei Dateien f_1 und f_2 lassen sich hierzu definieren (dabei sei

tokens eine Funktion, die die Tokens einer Datei wiedergibt und *hash* eine Funktion, die einen Hash-Wert für eine Datei über ihre Tokens berechnet, z.B. mittels MD5; ein *Token* ist die kleinste semantische Einheit einer Programmiersprache, wie zum Beispiel ein Schlüsselwort, ein Operator oder ein Bezeichner; die Funktion *clone* im Folgenden liefert die Menge geklonter Code-Fragmente zwischen zwei Dateien und *parameters* die Menge der Parameter einer Datei oder eines Code-Fragments; als Parameter werden solche Tokens betrachtet, die man eins-zu-eins substituieren kann, zum Beispiel werden im geklonten Code häufig Bezeichner oder Literale ausgetauscht):

$path\text{-}identical(f_1, f_2)$: zwei Dateien haben denselben relativen Pfad und Dateinamen im Quellcode-Verzeichnis

$$t\text{-}identical(f_1, f_2) \Leftrightarrow hash(tokens(f_1)) = hash(tokens(f_2))$$

$$tsim(f_1, f_2) = |\{t | t \in f_1 \wedge f \in clone(f_1, f_2)\}| \,/\, |f_1|$$

$$psim(f_1, f_2) = |parameters(f_1) \cap parameters(f_2)| \,/\, |parameters(f_1) \cup parameters(f_2)|$$

$$similar(f_1, f_2) \Leftrightarrow (tsim(f_1, f_2) \geq 0.7 \vee tsim(f_2, f_1) \geq 0.7) \wedge psim(f_1, f_2) \geq 0.75$$

Die für die Funktion *similar* gewählten Schwellenwerte wurden von uns in einer empirischen Untersuchung ermittelt.

Anhand dieser Funktionen lässt sich der Code im Stammbaum der Produktlinie dateiweise in folgende Kategorien einteilen (siehe auch Abb. 1):

- different
- moved and varied
- re-written
- varied
- moved
- identical

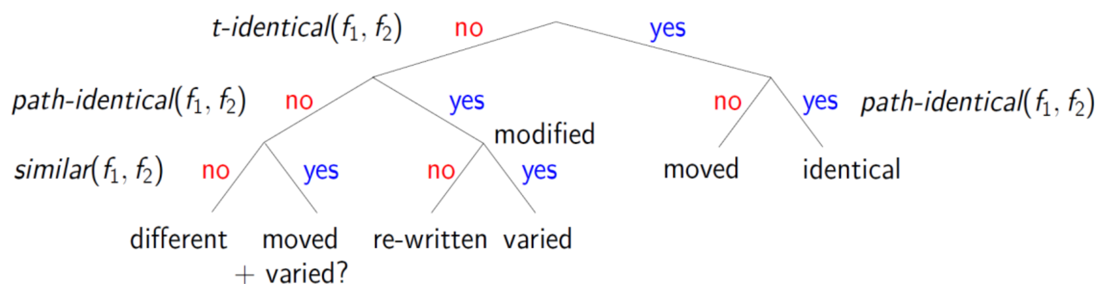

Abb. 1: Zusammenhang zwischen den Bewertungsfunktionen und den Kategorien.

Je nachdem, wie sich die geklonten Produkte entwickelt haben, kann es zu sehr ähnlichen oder auch sehr unähnlichen Variationen gekommen sein. Dadurch wird eine schrittweise Transformation in eine Produktlinie mehr oder weniger erschwert.

Die Analyse dieses Gesichtspunkts kann ebenfalls durch die Klonanalyse unterstützt werden.

Der zweite Schritt nach der Identifikation der Gemeinsamkeiten und Unterschiede ist die Konsolidierung der Software-Varianten. Das dabei zu lösende Problem ist die Beseitigung der Redundanz in den Softwarevarianten. Dazu braucht man einen Katalog an Vorgehensweisen zur Konsolidierung der Varianten, beispielsweise durch eine oder mehrere der folgenden Formen von Refactorings:

- Parametrisierung
- Einsatz von Templates/Generizität
- Einsatz von Entwurfsmustern
- Einsatz von Code-Generierung

Die Grundvoraussetzung für alle diese Konsolidierungsmaßnahmen genauso wie für die Sicherstellung der konsistenten Änderung, wenn die Klone nicht beseitigt werden sollen, ist, dass Gemeinsamkeiten und Variabilitäten zwischen Varianten auf allen Abstraktionsebenen der Software bekannt sind:

- Quellcode-Ebene: dies kann durch den Einsatz von Werkzeugen zum textuellen Vergleich, wie zum Beispiel *diff*, oder durch dedizierte Klonerkennungswerkzeuge erreicht werden.
- Architektur: mit Hilfe von Techniken der Architekturrekonstruktion [1, 2] kann durch ein Reverse-Engineering des Codes die Architektur wiedergewonnen und die verschiedenen Architekturen der einzelnen Produktvarianten miteinander verglichen werden. Dies verschafft eine höhere Abstraktion als die Code-Ebene.
- Funktionalität: auch die implementierten Features werden zwischen den Varianten variieren. Dabei kann es nicht nur gleiche und ganz verschiedene, sondern auch ähnliche Features geben. Dies betrifft sowohl funktionale als auch nicht-funktionale Anforderungen (wie z.B. Ressourcen-Verbrauch, Timing, Robustheit etc.).

Auf Quellcode-Ebene können ähnliche Funktionen zweier Varianten durch folgendes Vorgehen identifiziert werden:

1. Identifikation von Funktionspaaren mit Hilfe der Klonerkennung

2. Messung der Ähnlichkeit (Levenshtein-Distanz) entweder auf Basis einer textuelle, lexikalischen oder syntaktischen Darstellung des Programms

3. Sortierung nach Ähnlichkeit und Validierung der Ähnlichkeit; letzteres ist notwendig, weil eine automatisierter Algorithmus die Ähnlichkeit nur auf Basis syntaktischer Merkmale bestimmt, aber keinen Zugang zur Semantik der Programme hat

Auf der Architekurebene kann die Architektur einer Variante mit Hilfe des hypothesengetriebenen Ansatzes zur Architekturwiedergewinnung ermittelt werden, siehe [2]:

1. Stelle eine Hypothese für ein Architekturmodell auf
2. Extrahiere das Implementierungsmodell aus dem Code
3. Bilde die beiden Modelle aufeinander ab
4. Berechne das Reflexionsmodell, das Übereinstimmungen und Unterschiede zwischen den Architekturen automatisiert aufdeckt
5. Verfeinere/korrigiere die Hypothese, ermittle Kandidaten für das Refactoring

Für die Produktlinienarchitektur muss dieses Vorgehen erweitert werden, um aus den individuellen Produktarchitekturen eine übergeordnete Produktlinienarchitektur zu ermitteln. Dazu wird eine Hypothese für die Produktlinienarchitektur aufgestellt, die dann in der Folge mit den verschiedenen Produkten abgeglichen wird; siehe Abb. 2.

Im Wesentlichen werden bei diesem Prozess den Komponenten und ggf. ihren Abhängigkeiten in der Produktlinienarchitektur über UML-Stereotypen Attribute wie <<optional>>, <<kernel>> und <<variant>> mitgegeben. Das Attribut <<kernel>> beschreibt Komponenten und Abhängigkeiten, die in allen Produkten identisch enthalten sind, <<optional>> solche, die nicht in allen auftauchen, und <<variant>> solche, die zwar in allen Produktarchitekturen auftreten, jedoch in unterschiedlicher Form.

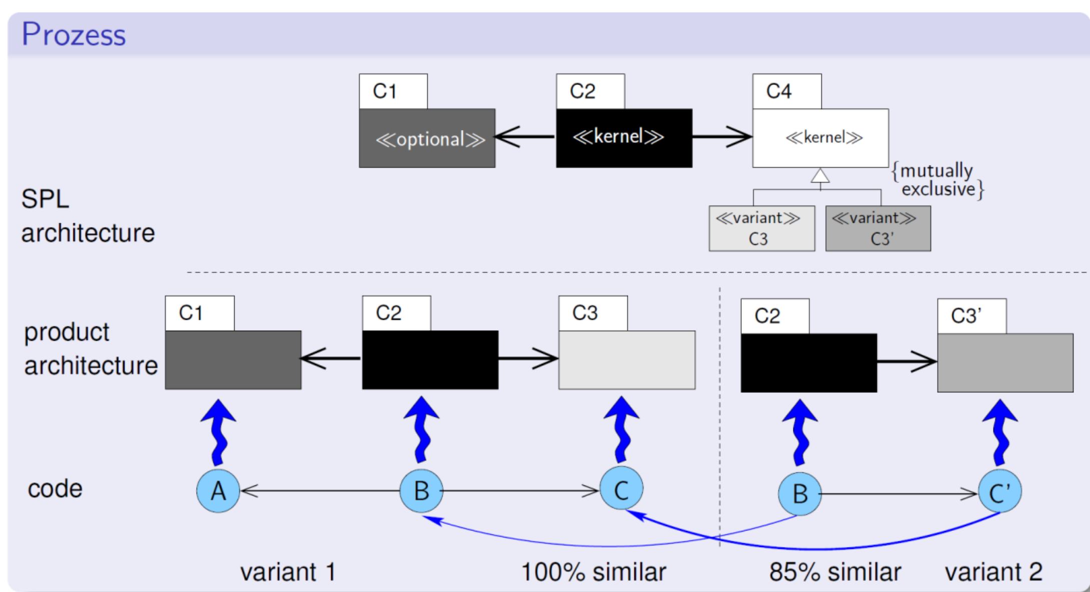

Abb. 2: Architekturwiedergewinnung, zwei Varianten und die daraus abgeleitete SPL-Architektur.

In der Praxis haben wir mit dieser Herangehensweise eine Reihe von Systemen im Rahmen industrieller Fallstudien analysiert. Dabei stellten sich folgende Erkenntnisse ein:

- Unsere Methode konnte identische und ähnliche Funktionspaare zwischen Varianten zuverlässig identifizieren.
- Wir fanden hohe Gemeinsamkeiten insbesondere für Code-Funktionen der als Kernel-Module identifizierten Einheiten, aber auch Variabilitäten.
- Es gab Ähnlichkeiten auch zwischen produktspezifischen Code-Funktionen. Hier wurde also Code nur zwischen einzelnen Produkten kopiert und angepasst.
- Es gab eine hohe Ähnlichkeit auf Architekturebene, die es erlaubte, den Blick für das Ganze zu bekommen.

Fazit

Gerade im Bereich eingebetteter Systeme, die sich nicht selten durch eine Vielfalt der zugrunde liegenden Hardware auszeichnen, wird häufig Wiederverwendung durch Copy & Paste im großen Stil betrieben. Wird dies wiederholt praktiziert, führt dies zu einer enormen Redundanz im Code. Änderungen werden damit erschwert. In diesen Fällen kann die Redundanz wieder durch geeignete Refactorings oder andere Maßnahmen wie die Code-Generierung beseitigt werden. Ist dies nicht möglich oder gewünscht, dann müssen die negativen Folgen der Redundanz zumindest durch eine Analyse der Redundanz kompensiert werden. Mit unserem Ansatz lassen sich Redundanzen und somit Gemeinsamkeiten und Unterschiede sowohl auf Code- als auch Architekturebene identifizieren. Das damit gewonnene Wissen kann dann benutzt werden, um geeignete Maßnahmen zu ergreifen, seien dies die Konsolidierung oder die geplante konsistente Wartung mehrerer ähnlicher Code-Teile. Dies senkt die Kosten in der Entwicklung und erhöht die Planbarkeit und die Fehlerfreiheit.

Quellen

[1] Koschke, Rainer; Simon, Daniel: *Hierarchical Reflexion Models*. In: Working Conference on Reverse Engineering, IEEE Computer Society Press, November 2003, S. 36–45

[2] R. Koschke, P. Frenzel, A. Breu, K. Angstmann. *Extending the reflexion method for consolidating software variants into product lines*. Software Quality Journal December 2009; 17(4):331–366.