Virtuelle Trennung von Belangen (Präprozessor 2.0)

Christian Kästner, Sven Apel, Gunter Saake

{kaestner,saake}@iti.cs.uni-magdeburg.de, apel@uni-passau.de

Abstract: Bedingte Kompilierung mit Präprozessoren wie *cpp* ist ein einfaches, aber wirksames Mittel zur Implementierung von Variabilität in Softwareproduktlinien. Durch das Annotieren von Code-Fragmenten mit *#ifdef* und *#endif* können verschiedene Programmvarianten mit oder ohne diesen Fragmenten generiert werden. Obwohl Präprozessoren häufig in der Praxis verwendet werden, werden sie oft für ihre negativen Auswirkungen auf Codequalität und Wartbarkeit kritisiert. Im Gegensatz zu modularen Implementierungen, etwa mit Komponenten oder Aspekte, vernachlässigen Präprozessoren die Trennung von Belangen im Quelltext, sind anfällig für subtile Fehler und verschlechtern die Lesbarkeit des Quellcodes. Wir zeigen, wie einfache Werkzeugunterstützung diese Probleme adressieren und zum Teil beheben bzw. die Vorteile einer modularen Implementierung emulieren kann. Gleichzeitig zeigen wir Vorteile von Präprozessoren wie Einfachheit und Sprachunabhängigkeit auf.

1 Einleitung

Der C-Präprozessor *cpp* und ähnliche Werkzeuge[1] werden in der Praxis häufig verwendet, um Variabilität zu implementieren. Quelltextfragmente, die mit *#ifdef* und *#endif* annotiert werden, können später beim Übersetzungsvorgang ausgeschlossen werden. Durch verschiedene Übersetzungsoptionen oder Konfigurationsdateien können so verschiedene Programmvarianten, mit oder ohne diese Quelltextfragmente, erstellt werden.

Präprozessoranweisungen sind zum Implementieren von *Softwareproduktlinien* sehr gebräuchlich. Eine Softwareproduktlinie ist dabei eine Menge von verwandten Anwendungen in einer Domäne, die alle aus einer gemeinsamen Quelltextbasis generiert werden können [BCK98, BKPS04]. Ein Beispiel ist eine Produktlinie für Datenbanksysteme, aus der man Produktvarianten entsprechend des benötigten Szenarios generieren kann [RALS09, Sel08], etwa ein Datenbanksystem mit oder ohne Transaktionen, mit oder ohne Replikation, usw. Die einzelnen Produktvarianten werden durch Features (oder Merkmale) unterschieden, welche die Gemeinsamkeiten und Unterschiede in der Domäne beschreiben [K$^+$90, AK09] – im Datenbankbeispiel etwa Transaktionen oder Replikation.

[1] Ursprünglich wurde *cpp* für Metaprogrammierung entworfen. Von seinen drei Funktionen (a) Einfügen von Dateiinhalten *(#include)*, (b) Makros *(#define)* und (c) bedingte Kompilierung *(#ifdef)* ist hier nur die bedingte Kompilierung relevant, die üblicherweise zur Implementierung von Variabilität verwendet wird. Neben *cpp* gibt es viele weitere Präprozessoren mit ähnlicher Funktionsweise. Zum Beispiel wird für Java-ME-Anwendungen der Präprozessor *Antenna* häufig verwendet, die Entwickler von Java's Swing Bibliothek implementierten einen eigenen Präprozessor *Munge*, die Programmiersprachen Fortran und Erlang haben ihren eigenen Präprozessor, und bedingte Kompilierung ist Bestandteil von Sprachen wie C#, Visual Basic, D, PL/SQL und Adobe Flex.

Eine Produktvariante wird durch eine Feature-Auswahl spezifiziert, z. B. „Die Datenbankvariante mit Transaktionen, aber ohne Replikation und Flash“. Kommerzielle Produktlinienwerkzeuge, etwa jene von *pure-systems* und *BigLever*, unterstützen Präprozessoren explizit.

Obwohl Präprozessoren in der Praxis sehr gebräuchlich sind, gibt es erhebliche Bedenken gegen ihren Einsatz. In der Literatur werden Präprozessoren sehr kritisch betrachtet. Eine Vielzahl von Studien zeigt dabei den negativen Einfluss der Präprozessornutzung auf Codequalität und Wartbarkeit, u.a. [SC92, KS94, Fav97, EBN02]. Präprozessoranweisungen wie *#ifdef* stehen dem fundamentalen Konzept der Trennung von Belangen entgegen und sind sehr anfällig für Fehler. Viele Forscher empfehlen daher, die Nutzung von Präprozessoren einzuschränken oder komplett abzuschaffen, und Produktlinien stattdessen mit ‚modernen‘ Implementierungsansätzen wie Komponenten und Frameworks [BKPS04], Feature-Modulen [Pre97], Aspekten [K+97] oder anderen zu implementieren, welche den Quelltext eines Features modularisieren.

Trotz allem stellen wir uns in diesem Beitrag auf die Seite der Präprozessoren und zeigen, wie man diese verbessern kann. Bereits einfache Erweiterungen an Konzepten und Werkzeugen und ein diziplinliertes Vorgehen können viele Fallstricke beseitigen. Daneben darf auch nicht vergessen werden, dass auch Präprozessoren neben den genannten Schwächen diverse Vorteile für die Produktlinienentwicklung besitzen. Wir zeigen, wie eigene und fremde Fortschritte an verschiedenen Fronten zu einer gemeinsamen Vision der virtuellen Trennung von Belangen zusammenwirken. Die Bezeichnung „Virtuelle Trennung von Belangen“ ergibt sich aus einer Erweiterung, die eine Trennung von Belangen emuliert, ohne den Quelltext wirklich in physisch getrennte Module zu teilen.

Schlussendlich muss noch erwähnt werden, dass auch die vorgestellten Erweiterungen nicht das letzte Wort zum Thema Produktlinienimplementierung sind. In unseren Arbeiten untersuchen wir sowohl modulare als auch präprozessorbasierte Implementierungsmöglichkeiten. Das Ziel dieses Beitrags ist, die in der Forschung unterrepräsentierten Präprozessoren und ihr noch nicht ausgeschöpftes Potential ins Bewusstsein der Forschergemeinde zu rücken.

2 Kritik an Präprozessoren

Im Folgenden werden die drei häufigsten Argumente gegen Präprozessoren vorgestellt: unzureichende Trennung von Belangen, Fehleranfälligkeit und unlesbarer Quelltext.

Trennung von Belangen. Die unzureichende Trennung von Belangen und die verwandten Probleme fehlender Modularität und erschwerter Auffindbarkeit von Quelltext eines Features sind in der Regel die größten Kritikpunkte an Präprozessoren. Anstatt den Quelltext eines Features in einem Modul (oder eine Datei, eine Klasse, ein Package, o.ä.) zusammenzufassen, ist Feature-relevanter Code in präprozessorbasierten Implementierungen in der gesamten Codebasis verstreut und mit dem Basisquelltext sowie dem Quelltext anderer Features vermischt. Im Datenbankbeispiel wäre etwa der gesamte Transaktionsquelltext (z. B. Halten und Freigabe von Sperren) über die gesamte Codebasis der Datenbank verteilt und vermischt mit dem Quelltext für Replikation und andere Features.

Die mangelnde Trennung von Belangen wird für eine Vielzahl von Problemen verantwortlich gemacht. Um das Verhalten eines Features zu verstehen, ist es zunächst nötig, den entsprechenden Quelltext zu finden. Dies bedeutet, dass die gesamte Codebasis durchsucht werden muss; es reicht nicht, ein einzelnes Modul zu durchsuchen. Man kann einem Feature nicht direkt zu seiner Implementierung folgen. Vermischter Quelltext lenkt zudem beim Verstehen des Quelltextes ab, weil man sich ständig mit Quelltext beschäftigen muss, der für die aktuelle Aufgabe nicht relevant ist. Die erschwerte Verständlichkeit des Quelltextes durch Verteilung und Vermischung des Quelltextes erhöht somit die Wartungskosten und widerspricht jahrzehntelanger Erfahrung im Software-Engineering.

Fehleranfälligkeit. Wenn Präprozessoren zur Implementierung von optionalen Features benutzt werden, können dabei sehr leicht subtile Fehler auftreten, die sehr schwer zu finden sind. Das beginnt schon mit einfachen Syntaxfehlern, da Präprozessoren wie *cpp* auf Basis von Textzeilen arbeiten, ohne den zugrundeliegenden Quelltext zu verstehen. Damit ist es ein leichtes, etwa nur eine öffnende Klammer, aber nicht die entsprechende schließende Klammer zu annotieren, wie in Abbildung 1 illustriert (die Klammer in Zeile 4 wird nur in Zeile 17 geschlossen, wenn das Feature *HAVE_QUEUE* ausgewählt ist; falls Feature *HAVE_QUEUE* nicht ausgewählt ist, fehlt dem resultierendem Program eine schließende Klammer). In diesem Fall haben wir den Fehler zu Anschauungszwecken selber eingebaut, aber ähnliche Fehler können leicht auftreten und sind schwer zu erkennen (wie uns mehrfach ausdrücklich von verschiedenen Produktlinienentwicklern bestätigt wurde). Die Verteilung des Featurecodes macht das Problem noch schwieriger.

Das größte Problem ist jedoch, dass ein Compiler solche Probleme bei der Entwicklung nicht erkennen kann, solange nicht der Entwickler (oder ein Kunde) irgendwann eine Produktvariante mit einer problematischen Featurekombination erstellt und übersetzt. Da es aber in einer Produktlinie sehr viele Produktvarianten geben kann (2^n für n unabhängige, optionale Features; industrielle Produktlinien haben hunderte bis tausende Features, beispielsweise hat der Linux Kernel über 8000 Konfigurationsoptionen [TSSPL09]), ist es unrealistisch, bei der Entwicklung immer alle Produktvarianten zu prüfen. Somit können selbst einfache Syntaxfehler, die sich hinter bestimmten Featurekombinationen verstecken, über lange Zeit unentdeckt bleiben und im Nachhinein (wenn ein bestimmtes Produkt generiert werden soll) hohe Wartungskosten verursachen.

Syntaxfehler sind eine einfache Kategorie von Fehlern. Darüber hinaus können natürlich genauso auch Typfehler und Verhaltensfehler auftreten, im schlimmsten Fall wieder nur in wenigen spezifischen Featurekombinationen. Beispielsweise muss beachtet werden, in welchem Kontext eine annotierte Methode aufgerufen wird. In Abbildung 2 ist die Methode *set* so annotiert, dass sie nur enthalten ist, wenn das Feature *Write* ausgewählt ist; ist es dagegen nicht ausgewählt, wird die Methode entfernt und es kommt zu einem Typfehler in Zeile 3, wo die Methode dennoch aufgerufen wird. Obwohl Compiler in statisch getypten Sprachen solche Fehler erkennen können, würde so ein Fehler wieder nur erkannt, wenn die problematische Featurekombination kompiliert wird.

```
1  static int __rep_queue_filedone(
       dbenv, rep, rfp)
2    DB_ENV *dbenv;
3    REP *rep;
4    __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6    COMPQUIET(rep, NULL);
7    COMPQUIET(rfp, NULL);
8    return (__db_no_queue_am(dbenv));
9  #else
10   db_pgno_t first, last;
11   u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
16   // weitere 100 Zeilen C Code
17 }
18 #endif
```

Abbildung 1: Modifizierter Quelltextauszug aus Oracle's Berkeley DB mit Syntaxfehler, wenn *HAVE_QUEUE* nicht ausgewählt ist.

```
1  class Database {
2    Storage storage;
3    void insert(Object key, Object data) {
4      storage.set(key, data);
5    }
6  }
7  class Storage {
8  #ifdef WRITE
9    boolean set(Object key, Object data) {
10     ...
11   }
12 #endif
13 }
```

Abbildung 2: Quelltextauszug mit Typfehler, wenn *WRITE* nicht ausgewählt ist.

Unlesbarer Quelltext. Beim Implementieren von Features mit *cpp* und ähnlichen Präprozessoren wird nicht nur der Quelltext verschiedener Features vermischt, sondern auch die Präprozessoranweisungen mit den Anweisungen der eigentlichen Programmiersprache. Beim Lesen des entsprechenden Quelltexts können eine Vielzahl von Präprozessoranweisungen vom eigentlichen Quelltext ablenken und zudem das gesamte Quelltextlayout zerstören (*cpp* erfordert, dass jede Anweisung in einer eigenen Zeile steht). Es gibt viele Beispiele, in denen Präprozessoranweisungen den Quelltext komplett zerstückeln – wie in Abbildung 3 gezeigt – und damit die Lesbarkeit und Wartbarkeit einschränken.

In Abbildung 3 wird der Präprozessor feingranular eingesetzt, um nicht nur Statements, sondern auch Parameter oder Teile von Ausdrücken zu annotieren [KAK08]. Durch Präprozessoranweisungen und zusätzliche notwendige Zeilenumbrüche werden insgesamt 21 statt neun Zeilen benötigt. Über das einfache Beispiel hinaus sind auch lange und geschachtelte Präprozessoranweisungen (siehe Abbildung 1) mitverantwortlich für schlechte Lesbarkeit.

Auch wenn das Beispiel in Abbildung 3 konstruiert wirkt, findet man ähnliche Beispiele in der Praxis. In Abbildung 4 sieht man etwa den Anteil an Präprozessoranweisungen im Quelltext des Echtzeitbetriebssystems *Femto OS*.

3 Virtuelle Trennung von Belangen

Nach einem Überblick über die wichtigsten Kritikpunkte von Präprozessoren diskutieren wir Lösungsansätze, die wir in ihrer Gesamtheit virtuelle Trennung von Belangen nennen. Diese Ansätze lösen nicht alle Probleme, können diese aber meist abschwächen. Zusammen mit den Vorteilen der Präprozessornutzung, die anschließend diskutiert wird, halten wir Präprozessoren für eine echte Alternative für Variabilitätsimplementierung.

```
1  class Stack {
2    void push(Object o
3  #ifdef TXN
4    , Transaction txn
5  #endif
6    ) {
7      if (o==null
8  #ifdef TXN
9        || txn==null
10 #endif
11       ) return;
12 #ifdef TXN
13     Lock l=txn.lock(o);
14 #endif
15     elementData[size++] = o;
16 #ifdef TXN
17     l.unlock();
18 #endif
19     fireStackChanged();
20   }
21 }
```

Abbildung 3: Java Quelltext zerstückelt durch feingranulare Annotationen mit *cpp*.

Abbildung 4: Präprozessoranweisungen in Femto OS (rote Linie = Präprozessoranweisung, weisse Linien = C-Code).

Trennung von Belangen. Eine der wichtigsten Motivationen für die Trennung von Belangen ist Auffindbarkeit, so dass ein Entwickler den gesamten Quelltext eines Features an einer einzigen Stelle finden und verstehen kann, ohne von anderen Quelltextfragmenten abgelenkt zu sein. Eine verteilte und vermischte präprozessorbasierte Implementierung kann dies nicht leisten, aber die Kernfrage „welcher Quelltext gehört zu diesem Feature“ kann mit *Sichten* trotzdem beantwortet werden [JDV04, SGC07, HKW08, KTA08].

Mit verhältnismäßig einfachen Werkzeugen ist es möglich, (editierbare) Sichten auf Quelltext zu erzeugen, die den Quelltext aller irrelevanten Features ausblenden. Technisch kann das analog zum Einklappen von Quelltext in modernen Entwicklungsumgebungen wie Eclipse implementiert werden.[2] Abbildung 5 zeigt beispielhaft ein Quelltextfragment und eine Sicht auf das darin enthaltende Feature *TXN*. Im Beispiel wird offensichtlich, dass es nicht ausreicht, nur den Quelltext zwischen *#ifdef*-Anweisungen zu zeigen, sondern dass auch ein entsprechender Kontext erhalten bleiben muss (z. B. in welcher Klasse und welcher Methode ist der Quelltext zu finden). In Abbildung 5 werden diese Kontextinformationen grau und kursiv dargestellt. Interessanterweise sind diese Kontextinformationen ähnlich zu Angaben, die auch bei modularen Implementierungen wiederholt werden müssen; dort be-

[2] Obwohl editierbare Sichten schwieriger zu implementieren sind als nicht-editierbare, sind editierbare Sichten nützlicher, da sie es dem Benutzer erlauben, den Quelltext zu ändern, ohne erst zurück zum Originalquelltext wechseln zu müssen. Lösungen für editierbare Sichten sind sowohl aus dem Bereich der Datenbanken wie auch aus bidirektionalen Modelltransformationen bekannt. Eine einfache, aber effektive Lösung, die auch in unseren Werkzeugen benutzt wird, ist es, Markierungen in der Sicht zu belassen, die ausgeblendeten Quelltext anzeigen. Änderungen am Quelltext vor oder nach der Markierung können so eindeutig in den Originalquelltext zurückpropagiert werden.

```
1  class Stack implements IStack {
2    void push(Object o) {
3  #ifdef TXN
4      Lock l = lock(o);
5  #endif
6  #ifdef UNDO
7      last = elementData[size];
8  #endif
9      elementData[size++] = o;
10 #ifdef TXN
11     l.unlock();
12 #endif
13     fireStackChanged();
14   }
15 #ifdef TXN
16   Lock lock(Object o) {
17     return LockMgr.lockObject(o);
18   }
19 #endif
20   ...
21 }
```

(a) Originalquelltext

```
1  class Stack [] {
2    void push([]) {
3      Lock l = lock(o);
4      []
5      l.unlock();
6      []
7    }
8    Lock lock(Object o) {
9      return LockMgr.lockObject(o);
10   }
11   ...
12 }
```

(b) Sicht auf das Feature TXN (ausgeblendeter Code ist markiert mit '[]'; Kontextinformation ist schrägestellt und grau dargestellt)

Abbildung 5: Sichten emulieren Trennung von Belangen.

finden sich die Kontextinformationen etwa in Schnittstellen von Komponenten und Plugins oder Pointcuts von Aspekten.

Mit Sichten können dementsprechend einige Vorteile der physischen Trennung von Belangen emuliert werden. Damit können auch schwierige Probleme bei der Modularisierung wie das „Expression Problem“ [TOHS99] auf natürliche Weise gelöst werden: entsprechender Quelltext erscheint in mehren Sichten.

Das Konzept von Sichten für präprozessorbasierte Implementierungen kann auch leicht erweitert werden, so dass nicht nur Sichten auf einzelne Features, sondern auch editierbare Sichten auf den gesamten Quelltext einer Produktvariante möglich sind. Eine Sicht kann also genau jenen Quelltext anzeigen, der in einer spezifischen Produktvariante für eine Featureauswahl kompiliert würde, und alle Quelltextfragmente von nicht ausgewählten Features ausblenden. Eine solche Sicht ist sinnvoll, um Fehler in einer bestimmten Variante zu suchen oder um das Verhalten mehrer Feature in Kombination (Stichwort Featureinteraktionen [C$^+$03]) zu studieren. Diese Möglichkeiten von Sichten gehen über das hinaus, was bei modularen Implementierung möglich ist; dort müssen Entwickler das Verhalten von Produktvarianten oder Featurekombinationen im Kopf aus mehreren Komponenten, Plugins oder Aspekten rekonstruieren. Besonders wenn mehrere Features auf feingranularer Ebene interagieren (etwa innerhalb einer Methode), können Sichten eine wesentliche Hilfe sein.

Obwohl Sichten viele Nachteile der fehlenden physischen Trennung von Belangen abmildern können, können zugegebenermaßen nicht alle Nachteile beseitigt werden. Wenn Anforderungen wie separate Kompilierung oder modulare Typprüfung von Features bestehen, helfen auch Sichten nicht weiter. In der Praxis können Sichten aber bereits eine große Hilfe darstellen.

Fehleranfälligkeit. Auch Fehler, die bei Präprozessornutzung entstehen können, können mit Werkzeugunterstützung verhindert werden. Wir stellen insbesondere zwei Gruppen von Ansätze vor: disziplinierte Annotationen gegen Syntaxfehler wie in Abbildung 1 und produktlinienorientierte Typsysteme gegen Typfehler wie in Abbildung 2. Auf Fehler im Laufzeitverhalten (z. B. Deadlocks) gehen wir nicht weiter ein, da diese unserer Meinung nach kein spezifisches Problem von Präprozessoren darstellen, sondern bei modularisierten Implementierungen genauso auftreten können.

Disziplinierte Annotationen. Unter disziplinierten Annotationen versteht man Ansätze, welche die Ausdrucksfähigkeit von Annotationen einschränken, um Syntaxfehler zu vermeiden, ohne aber die Anwendbarkeit in der Praxis zu behindern [KAT+09]. Syntaxfehler entstehen im wesentlichen dadurch, dass Präprozessoren Quelltext als reine Zeichenfolgen sehen und erlauben, dass jedes beliebige Zeichen, einschließlich einzelner Klammern, annotiert werden kann. Disziplinierte Annotationen dagegen berücksichtigen die zugrundeliegende Struktur des Quelltextes und erlauben nur, dass ganze Programmelemente wie Klassen, Methoden oder Statements annotiert (und entfernt) werden können. Die Annotationen in Abbildungen 2 und 5a sind diszipliniert, da nur ganze Statements und Methoden annotiert werden. Syntaxfehler wie in Abbildung 1 sind nicht mehr möglich, wenn disziplinierte Annotationen durchgesetzt werden.

Durch die eingeschränkten Ausdrucksmöglichkeiten mag es für bestimmte Aufgaben schwieriger sein, entsprechende Implementierungen zu finden. In einigen Fällen muss dazu der Quelltext umgeschrieben werden, um das gleiche Verhalten auch mit disziplinierten Annotationen zu ermöglichen. Allerdings hat sich herausgestellt, dass in der Praxis disziplinierte Annotationen die Regel darstellen und andere Implementierungen als 'hack' betrachtet werden [BM01, Vit03]. Der Übergang von undisziplinierten zu disziplinierten Annotationen ist typischerweise einfach und die nötigen Änderungen folgen offensichtlichen, einfachen Mustern. Trotz Einschränkung der Ausdrucksfähigkeit sind disziplinierte Annotationen aber immer noch deutlich ausdrucksfähiger als das, was Modularisierungsansätze wie Komponenten, Plugins oder Aspekte bieten [KAK08].

Auf technischer Seite erfordern disziplinierte Annotationen aufwendigere Werkzeuge als undisziplinierte, da der Präprozessor die zugrundeliegende Quelltextstruktur analysieren muss. Werkzeuge für disziplinierte Annotationen können entweder für bestehenden Quelltext prüfen, ob dieser in disziplinierter Form vorliegt, oder sie können (wie in CIDE [KAK08]) bereits in der Entwicklungsumgebung alle Annotationen verwalten und überhaupt nur disziplinierte Annotationen erlauben. Wie in [KAT+09] gezeigt, ist auch die Erweiterung auf weitere Programmiersprachen und deren zugrundeliegende Struktur einfach und kann weitgehend automatisiert werden, wenn eine Grammatik für die Zielsprache vorliegt.

Produktlinienorientierte Typsysteme. Mit angepassten Typsystemen für Produktlinien ist es möglich, alle Produktvarianten einer Produktlinie auf Typsicherheit zu prüfen, ohne jede Variante einzeln zu kompilieren [CP06, KA08]. Damit können viele wichtige Probleme erkannt werden, wie beispielsweise Methoden oder Klassen, deren Deklaration in einigen Produktvarianten entfernt, die aber trotzdem noch referenziert werden (siehe Abbildung 2).

Während ein normales Typsystem prüft, ob es zu einem Methodenaufruf eine passende Methodendeklaration gibt, wird dies von produktlinienorientierten Typsystemen erweitert,

so dass zudem auch geprüft wird, dass in jeder möglichen Produktvariante entweder die passende Methodendeklaration existiert oder auch der Methodenaufruf gelöscht wurde. Wenn Aufruf und Deklaration mit dem gleichen Feature annotiert sind, funktioniert der Aufruf in jeder Variante; in allen anderen Fällen muss die Beziehung zwischen den jeweiligen Annotationen geprüft werden. Erlauben die Annotationen eine Produktvariante, in der der Aufruf, aber nicht die Deklaration vorhanden ist, wird ein Fehler gemeldet.[3] Durch diese erweiterten Prüfungen zwischen Aufruf und Deklaration (und vielen ähnlichen Paaren) wird mit einem Durchlauf die gesamte Produktlinie geprüft; es ist nicht nötig, jede Produktvariante einzeln zu prüfen.

Wie bei Sichten emulieren produktlinienorientierte Typsysteme wieder einige Vorteile von modularisierten Implementierungen. Statt Modulen und ihren Abhängigkeiten gibt es verteilte, markierte Codefragmente und Abhängigkeiten zwischen Features. Das Typsystem prüft dann, dass auch im verteilten Quelltext diese Beziehungen zwischen Features beachtet werden.

Durch die Kombination von disziplinierten Annotationen und produktlinienorientierten Typsystemen kann die Fehleranfälligkeit von Präprozessoren reduziert werden, mindestens auf das Niveau von modularisierten Implementierungsansätzen. Insbesondere produktlinienorientierte Typsysteme haben sich dabei als hilfreich erwiesen und wurden auch auf modulbasierte Ansätze übertragen (z. B. [TBKC07]).

Schwer verständlicher Quelltext. Durch viele textuelle Annotationen im Quelltext kann dieser schwer lesbar werden, wie in Abbildungen 3 und 4 gezeigt. Hauptverantwortlich dafür ist, dass Präprozessoren wie *cpp* zwei Extrazeilen für jede Annotation benötigen (*#ifdef* und *#endif* je in einer neuen Zeile) und nicht Bestandteil der Host-Sprache sind.

Es gibt verschiedene Ansätze, wie die Darstellung und damit die Lesbarkeit verbessert werden kann. Ein erster Ansatz ist, textuelle Annotationen in einer Sprache mit kürzerer Syntax zu verwenden, die auch Annotationen innerhalb einer Zeile erlauben. Eine zweite Verbesserung ist die Verwendung von Sichten, wie oben diskutiert, welche jene Annotationen, die für die aktuelle Aufgabe unwichtig sind, ausblenden kann. Eine dritte Möglichkeit ist es, Annotationen gezielt grafisch abzusetzen, so dass man sie leichter identifizieren kann. Beispiele dafür sind einige Entwicklungsumgebungen für PHP, die verschiedene Hintergrundfarben für PHP- und HTML-Quelltext innerhalb der gleichen Datei verwenden. Schlussendlich ist es sogar möglich, auf textuelle Annotationen komplett zu verzichten und stattdessen Annotationen komplett auf die Repräsentationsschicht zu verlegen, wie in unserem Werkzeug CIDE.

In CIDE gibt es keine textuellen Annotationen, stattdessen werden Hintergrundfarben im Editor zur Repräsentation von Annotationen benutzt [KAK08]. Beispielsweise wird der gesamte Quelltext der Transaktionsverwaltung in Abbildung 6 mit roter Hintergrundfarbe dargestellt. Auf diese Weise kann man den Quelltext (ursprünglich aus Abbildung 3) deutlich kürzer und lesbarer darstellen. Hintergrundfarben wurden dabei inspiriert von

[3] Es gibt viele Möglichkeiten, Beziehungen zwischen Features und Produktvarianten zu beschreiben. Feature-Modelle und aussagenlogische Ausdrücke sind dabei übliche Mechanismen, über die man zudem auch automatisiert Schlüsse ziehen kann [Bat05].

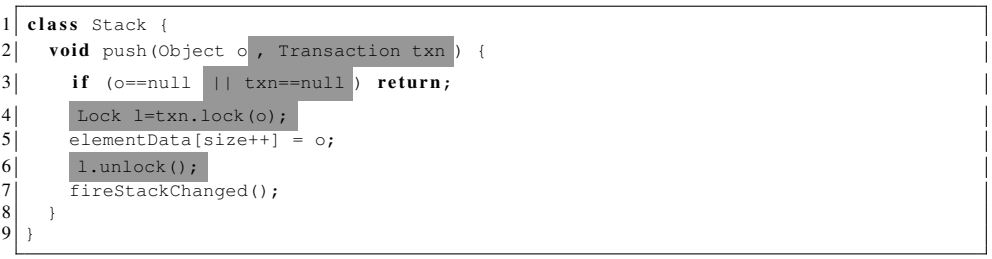

Abbildung 6: Hintergrundfarbe statt textueller Anweisung zum Annotieren von Quelltext.

Quelltextausdrucken auf Papier, die wir zur Analyse mit farbigen Textmarkern (eine Farbe pro Feature) markiert haben. Hintergrundfarben lassen sich leicht in Entwicklungsumgebungen integrieren und sind dort in der Regel noch nicht belegt. Statt Hintergrundfarben gibt es natürlich auch noch viele andere mögliche Darstellungsformen, z. B. farbige Linien neben dem Editor, wie sie in *Spotlight* verwendet [CPR07]. Hintergrundfarben und Linien sind besonders hilfreich bei langen und geschachtelten Annotationen, die bei textuellen Annotationen häufig schwierig nachzuvollziehen sind, besonders wenn das *#endif* einige hundert Zeilen nach dem *#ifdef* folgt wie in Abbildung 1. Uns sind die Begrenzungen von Farben bewusst (z. B. können Menschen nur relativ wenige Farben sicher unterscheiden), aber es gibt ein weites Feld an Darstellungsformen, das noch viel Platz für Verbesserungen lässt.

Trotz aller grafischen Verbesserungen und Werkzeugunterstützung sollte man aber nicht aus dem Auge verlieren, dass der Präprozessor (auch wenn es vielleicht einladend wirkt) nicht als Rechtfertigung dafür dienen darf, Quelltext gar nicht mehr zu modularisieren (mittels Klassen, Paketen, etc.). Sie erlauben den Entwicklern nur mehr Freiheit und zwingen sie nicht mehr, alles um jeden Preis zu modularisieren. Typischerweise wird ein Feature weiterhin in einem Modul oder eine Klasse implementiert, lediglich die Aufrufe verbleiben verteilt und annotiert im Quelltext. Wenn dies der Fall ist, befinden sich auf einer Bildschirmseite Quelltext (nach unseren Erfahrungen mit CIDE) selten Annotationen zu mehr als zwei oder drei Features, so dass man auch mit einfachen grafischen Mitteln viel erreichen kann.

Vorteile von Präprozessoren. Neben allen Problemen haben Präprozessoren auch einige Vorteile, die wir hier nicht unter den Tisch fallen lassen wollen. Der erste und wichtigste ist, dass Präprozessoren ein *sehr einfaches Programmiermodell* haben: Quelltext wird annotiert und entfernt. Präprozessoren sind daher sehr leicht zu erlernen und zu verstehen. Im Gegensatz zu vielen anderen Ansätzen wird keine neue Spracherweiterung, keine besondere Architektur und kein neuer Entwicklungsprozess benötigt. In vielen Sprachen ist der Präprozessor bereits enthalten, in allen anderen kann er leicht hinzugefügt werden. Diese Einfachheit ist der Hauptvorteil des Präprozessors und wahrscheinlich der Hauptgrund dafür, dass er so häufig in der Praxis verwendet wird.

Zweitens sind Präprozessoren *sprachunabhängig* und können für alle Sprachen *gleichförmig* eingesetzt werden. Beispielsweise kann *cpp* nicht nur für C-Quelltext, sondern auch genauso

für Java und HTML verwendet werden. Anstelle eines Tools oder einer Spracherweiterung pro Sprache (etwa AspectJ für Java, AspectC für C, Aspect-UML für UML usw.) funktioniert der Präprozessor für alle Sprachen gleich. Selbst mit disziplinierten Annotationen können Werkzeuge sprachübergreifend verwendet werden [KAT+09].

Drittens, wie bereits angedeutet, verhindern Präprozessoren nicht die traditionellen Möglichkeiten zur Trennung von Belangen. Eine primäre (dominante) Dekomposition etwa mittels Klassen oder Modulen ist weiterhin möglich und sinnvoll. Präprozessoren fügen aber weitere Ausdrucksfähigkeit hinzu, wo traditionelle Modularisierungsansätze an ihre Grenzen stoßen mit querschneidenden Belangen oder mehrdimensionaler Trennung von Belangen [K+97, TOHS99]. Eben solche Probleme können mit verteilten Quelltext und später Sichten auf den Quelltext leicht gelöst werden.

Werkzeuge. Die in diesem Beitrag vorgestellten Verbesserungen für Präprozessoren kommen aus verschiedenen Forschungsarbeiten. Wir haben eigene und verwandte Arbeiten in dem Bereich vorgestellt und zu einer einheitlichen Lösung zusammengeführt. Alle Verbesserungen – Sichten auf Features und Produktvarianten, erzwungene disziplinierte Annotationen, ein Java-Typsystem für Produktlinien und eine visuelle Darstellung von Annotationen – sind in unserem Produktlinienwerkzeug-Prototyp CIDE unter anderem für Java implementiert. In CIDE werden Annotationen statt als textuelle *#ifdef*-Anweisungen direkt in der Entwicklungsumgebung als Mapping zwischen Features und der zugrundeliegenden Quelltextstruktur gespeichert. Es erzwingt daher von vornherein disziplinierte Annotationen und eignet sich (da alle Annotationen leicht zugreifbar verfügbar sind) gut als Testbett für Typsysteme, Sichten und verschiedene Visualisierungen. CIDE steht unter `http://fosd.de/cide` zum Ausprobieren zusammen mit diversen Fallbeispielen zur Verfügung.

4 Zusammenfassung

Unsere Kernmotivation für diesen Beitrag war es zu zeigen, dass Präprozessoren für die Produktlinienentwicklung keine hoffnungslosen Fälle sind. Mit etwas Werkzeugunterstützung können viele der Probleme, für die Präprozessoren kritisiert werden, leicht behoben oder zumindest abgeschwächt werden. Sichten auf den Quelltext können Modularität oder eine Trennung von Belangen emulieren, disziplinierte Annotationen und Typsysteme für Produktlinien können Implementierungsprobleme frühzeitig erkennen und Quelltexteditoren können den Unterschied zwischen Quelltext und Annotationen hervorheben oder Annotationen gar komplett auf die Repräsentationsschicht verlagern. Obwohl wir nicht alle Probleme der Präprozessoren lösen können (beispielsweise ist ein separates Kompilieren der Features weiterhin nicht möglich), haben Präprozessoren auch einige Vorteile, insbesondere das einfache Programmiermodell und die Sprachunabhängigkeit. Zusammen nennen wir diese Verbesserungen „Virtuelle Trennung von Belangen“, da sie, obwohl Features nicht tatsächlich physisch in Module getrennt werden, dennoch diese Trennung durch Werkzeugunterstützung emulieren.

Als Abschluss möchten wir noch einmal betonen, dass wir selber nicht endgültig entschei-

den können, ob eine echte Modularisierung oder eine virtuelle Trennung langfristig der bessere Ansatz ist. In unserer Forschung betrachten wir beide Richtungen und auch deren Integration. Dennoch möchten wir mit diesem Beitrag Forscher ermuntern, die Vorurteile gegenüber Präprozessoren (üblicherweise aus Erfahrung mit *cpp*) abzulegen und einen neuen Blick zu wagen. Entwickler in der Praxis, die zurzeit Präprozessoren verwenden, möchten wir im Gegenzug ermuntern, nach Verbesserungen Ausschau zu halten bzw. diese von den Werkzeugherstellern einzufordern.

Danksagung. Wir danken Jörg Liebig, Marko Rosenmüller, Don Batory und Jan Hoffmann für ihre Unterstützung und die Quelltextbeispiele aus Berkeley DB und Femto OS. Sven Apel wurde durch die DFG unterstützt, Projektnummer AP 206/2-1.

Literatur

[AK09] Sven Apel und Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, Jgg. 3714 of *LNCS*, Seiten 7–20. Springer, 2005.

[BCK98] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl und Klaus Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. Dpunkt Verlag, 2004.

[BM01] Ira Baxter und Michael Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, Seiten 281–290. IEEE, 2001.

[C+03] Muffy Calder et al. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.

[CP06] Krzysztof Czarnecki und Krzysztof Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, Seiten 211–220. ACM, 2006.

[CPR07] David Coppit, Robert Painter und Meghan Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. Software Eng. (ICSE)*, Seiten 754–757. IEEE, 2007.

[EBN02] Michael Ernst, Greg Badros und David Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.

[Fav97] Jean-Marie Favre. Understanding-In-The-Large. In *Proc. Int'l Workshop on Program Comprehension*, Seite 29. IEEE, 1997.

[HKW08] Florian Heidenreich, Jan Kopcsek und Christian Wende. FeatureMapper: Mapping Features to Models. In *Comp. Int'l Conf. Software Engineering (ICSE)*, Seiten 943–944. ACM, 2008.

[JDV04] Doug Janzen und Kris De Volder. Programming with Crosscutting Effective Views. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, Jgg. 3086 of *LNCS*, Seiten 195–218. Springer, 2004.

[K+90] Kyo Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Bericht CMU/SEI-90-TR-21, SEI, 1990.

[K+97] Gregor Kiczales et al. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, Jgg. 1241 of *LNCS*, Seiten 220–242. Springer, 1997.

[KA08] Christian Kästner und Sven Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, Seiten 258–267. IEEE, 2008.

[KAK08] Christian Kästner, Sven Apel und Martin Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Eng. (ICSE)*, Seiten 311–320. ACM, 2008.

[KAT+09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann und Don Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, Jgg. 33 of *LNBIP*, Seiten 175–194. Springer, 2009.

[KS94] Maren Krone und Gregor Snelting. On the Inference of Configuration Structures from Source Code. In *Proc. Int'l Conf. Software Eng. (ICSE)*, Seiten 49–57. IEEE, 1994.

[KTA08] Christian Kästner, Salvador Trujillo und Sven Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*. Lero, 2008.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, Jgg. 1241 of *LNCS*, Seiten 419–443. Springer, 1997.

[RALS09] Marko Rosenmüller, Sven Apel, Thomas Leich und Gunter Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 68(12):1493–1512, 2009.

[SC92] Henry Spencer und Geoff Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, Seiten 185–198, 1992.

[Sel08] Margo Seltzer. Beyond Relational Databases. *Commun. ACM*, 51(7):52–58, 2008.

[SGC07] Nieraj Singh, Celina Gibbs und Yvonne Coady. C-CLR: A Tool for Navigating Highly Configurable System Software. In *Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Seite 9. ACM, 2007.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin und William Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, Seiten 95–104. ACM, 2007.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison und Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. Software Eng. (ICSE)*, Seiten 107–119. IEEE, 1999.

[TSSPL09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat und Daniel Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. GPCE Workshop on Feature-Oriented Software Dev. (FOSD)*, Seiten 81–86, New York, NY, 2009. ACM.

[Vit03] Marian Vittek. Refactoring Browser with Preprocessor. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, Seiten 101–110. IEEE, 2003.