Virtuelle Trennung von Belangen*

Christian Kästner
Philipps Universität Marburg
FB 12, Hans-Meerwein Str., 35032 Marburg

Abstract: Bedingte Kompilierung ist ein einfaches und häufig benutztes Mittel zur Implementierung von Variabilität in Softwareproduktlinien, welches aber aufgrund negativer Auswirkungen auf Codequalität und Wartbarkeit stark kritisiert wird. Wir zeigen wie Werkzeugunterstützung – Sichten, Visualisierung, kontrollierte Annotationen, Produktlinien-Typsystem – die wesentlichen Probleme beheben kann und viele Vorteile einer modularen Entwicklung emuliert. Wir bieten damit eine Alternative zur klassischen Trennung von Belangen mittels Modulen. Statt Quelltext notwendigerweise in Dateien zu separieren erzielen wir eine *virtuelle Trennung von Belangen* durch entsprechender Werkzeugunterstüzung.

1 Einleitung

Der C-Präprozessor *cpp* und ähnliche lexikalische Werkzeuge[1] werden in der Praxis häufig verwendet, um Variabilität zu implementieren die zur Kompilierzeit entschieden wird. 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 annotierten Quelltextfragmente, erstellt werden.

Annotationsbasierte Ansätze wie lexikalische Präprozessoren 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]. Ein Beispiel ist eine Produktlinie für Datenbanksysteme, aus der man Produktvarianten entsprechend des benötigten Szenarios generieren kann, etwa ein Datenbanksystem mit oder ohne Transaktionen, mit oder ohne Replikation, usw. Die einzelnen Produktvarianten werden durch Features (auch Merkmale genannt) unterschieden, welche die Gemeinsamkeiten und Unterschiede in der Domäne beschreiben – im Datenbankbeispiel etwa Transaktionen oder Replikation. Eine Produktvariante wird durch eine Feature-Auswahl spezifiziert, z. B. „Die Datenbankvariante mit

*Dieses Dokument ist eine übersetzte Kurzfassung der Dissertation "Virtual Separation of Concerns" und teilt Textfragmente mit folgender Publikation: C. Kästner, S. Apel, and G. Saake. Virtuelle Trennung von Belangen (Präprozessor 2.0). In *Software Engineering 2010 – Fachtagung des GI-Fachbereichs Softwaretechnik*, number P-159 in Lecture Notes in Informatics, pages 165–176, Gesellschaft für Informatik (GI), 2010.

[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 und für viele weitere Sprachen.

Transaktionen, aber ohne Replikation und Flash“. Kommerzielle Produktlinienwerkzeuge, etwa jene von *pure-systems* und *BigLever*, unterstützen Präprozessoren explizit.

Obwohl annotationsbasierte Ansätze in der Praxis sehr gebräuchlich sind, gibt es erhebliche Bedenken gegen ihren Einsatz. In der Literatur werden insbesondere lexikalische Präprozessoren sehr kritisch betrachtet. Eine Vielzahl von Studien zeigt dabei den negativen Einfluss der Präprozessornutzung auf Codequalität und Wartbarkeit, u.a. [SC92, 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 [BCK98], Feature-Modulen [Pre97], Aspekten [KLM+97] oder anderen zu implementieren, welche den Quelltext eines Features modularisieren.

Wir rehabilitieren annotationsbasierte Implementierungen indem wir zeigen, wie Werkzeugunterstützung die meißten Probleme adressieren und beheben oder zumindest die Vorteile einer modularen Implementierung emulieren kann. Mit Sichten auf die Implementierung emulieren wir Modularität trotz verstreutem Quelltext. Durch visuelle Repräsentation von Annotationen verbessern wir Lesbarkeit und Programmverständniss. Mit disziplinierten Annotationen und einen produktlinienorientiertem Typsystem stellen wir Konsistenz sicher und verhindern bzw. erkennen Syntax- und Typfehler in der gesamten Produktlinie. Neben allen Verbesserungen zeigen und erhalten wir auch Vorteile von annotationsbasierten Implementierungen wie Einfachheit, hohe Ausdruckskraft und Sprachunabhängigkeit. Zusammenfassend verbessern wir annotationsbasierte Implementierung und bieten eine Form der Trennung von Belangen an ohne dass der Quelltext notwendigerweise in Module aufgeteilt wird. Wir sprechen daher von *virtueller Trennung von Belangen*.

Unsere Ergebnisse zeigen dass annotationsbasierte Implementierungen mit entsprechenden Verbesserungen mit modularen Implementierungen konkurrieren können. Modulare und annotationsbasierte Implementierungen haben jeweils ihre Stärken und wir zeigen Möglichkeiten für eine Integration und Migration auf. Wir geben zwar keine endgültige Empfehlung für die einen oder anderen Form der Produktlinienimplementierung, wir zeigen jedoch dass für Präprozessoren trotz ihres schlechten Rufs noch Hoffnung besteht; sie wurden nur in der Forschung weitgehend ignoriert. Durch die vorgestellten Verbesserungen stellt die virtuelle Trennung von Belangen einen konkurrenzfähigen Implementierungsansatz für Softwareproduktlinien dar.

2 Kritik an Präprozessoren

Im Folgenden werden die drei häufigsten Argumente gegen annotationsbasierte Implementierungen 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 ab. 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 lexikalische Präprozessoren wie *cpp* auf Basis von Token arbeiten, ohne die Strukur des zugrundeliegenden Quelltextes zu berücksichtigen. 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 in Zeile 17 geschlossen; falls Feature *HAVE_QUEUE* nicht ausgewählt ist, fehlt dem resultierendem Program eine schließende Klammer). In diesem Fall haben wir den Fehler selber eingebaut, aber ähnliche Fehler können leicht auftreten und sind schwer zu erkennen (wie uns ausdrücklich von mehreren Produktlinienentwicklern bestätigt wurde). Verteilung von Featurecode macht das Problem noch ernster.

Schlimmer noch, ein Compiler kann solche Probleme bei der Entwicklung nicht erkennen, 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 10 000 Konfigurationsoptionen), ist es unrealistisch, bei der Entwicklung immer alle Produktvarianten zu prüfen. Somit können selbst einfache Syntaxfehler über lange Zeit unentdeckt bleiben und im Nachhinein (wenn ein bestimmtes Produkt generiert werden soll) hohe Wartungskosten verursachen.

Syntaxfehler sind nur 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; in allen anderen Varianten kommt es zu einem Typfehler in Zeile 3, wo die Methode dennoch aufgerufen wird. Obwohl Compiler in statisch getypten Sprachen solche Fehler erkennen können, hilft dies nur 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. Viele Präprozessoranweisungen können vom eigentlichen Quelltext ablenken und zudem das gesamte Quelltextlayout zerstören. Es gibt viele Beispiele, in denen Präprozessoranweisungen den Quelltext komplett zerstückeln und damit die Lesbarkeit und Wartbarkeit einschränken. Dies gilt insbesondere, wenn, wie in Abbildung 3, der Präprozessor feingranular eingesetzt wird, um nicht nur Statements, sondern auch Parameter oder Teile von Ausdrücken zu annotieren. 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 zwar 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.

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

```
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*.

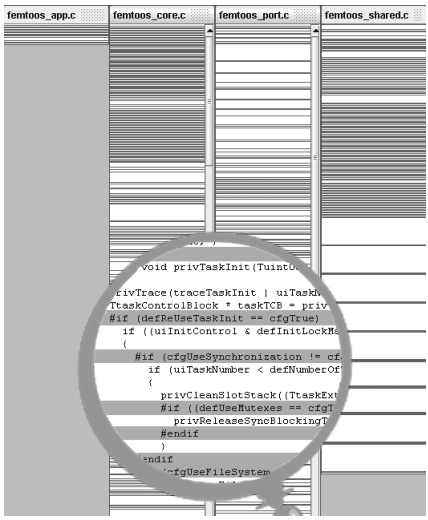

Abbildung 4: Präprozessoranweisungen in Femto OS (rote Linie = Präprozessoranweisung, weisse Linien = C-Code).

abgelenkt zu sein. Die Kernfrage „welcher Quelltext gehört zu diesem Feature“ kann mit *Sichten* auch in verteiltem und vermischten Quelltext beantwortet werden [KAK08].

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. 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 Modulen in Schnittstellen wiederholt werden müssen.

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] oder die Implementierung von Featureinteraktionen [CKMRM03] auf natürliche Weise gelöst werden: entsprechender Quelltext erscheint in mehren Sichten.

Das Konzept von Sichten für präprozessorbasierte Implementierungen kann leicht erweitert werden, so dass nicht nur Sichten auf einzelne Features, sondern auch editierbare Sichten auf den gesamten Quelltext einer Produktvariante möglich sind. 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 Modulen rekonstruieren, was besonders bei feingranularen Interaktionen mühsam sein kann.

```
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.

Sichten können viele Nachteile der fehlenden physischen Trennung von Belangen abmildern, aber zugegebenermaßen nicht alle Nachteile beseitigen. Separate Kompilierung oder modulare Typprüfung von Features sind durch Sichten nicht möglich. In der Praxis können Sichten aber bereits eine große Hilfe darstellen. Atkins et al. haben beispielsweise mit Sichten in einem vergleichbaren Kontext eine Produktivitätssteigerung der Entwickler um 40 % gemessen [ABGM02].

Fehleranfälligkeit. Auch Fehler, die bei Präprozessornutzung entstehen können, können mit Werkzeugunterstützung verhindert oder erkannt 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. Als disziplinierten Annotationen verstehen wir Ansätze, welche die Ausdrucksfähigkeit von Annotationen einschränken und sie auf syntaktische Sprachkonstrukte beschränken. Dies verhindert Syntaxfehler ohne aber die Anwendbarkeit in der Praxis zu behindern [KAT+09]. Syntaxfehler entstehen im wesentlichen dadurch, dass lexicalische Präprozessoren Quelltext als reine Zeichenfolgen sehen und erlauben, dass jedes beliebige Zeichen, einschließlich einzelner Klammern, annotiert werden kann. Disziplinierte Annotationen dagegen berücksichtigen die zugrundeliegende syntaktische 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.

Disziplinierte Annotationen erleichtern zudem die Analyse des Mappings zwischen Features und Quelltextfragmenten, wie Sie etwa für Quelltext-Transformationen und Typprüfungen gebraucht werden. 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, s.u.) bereits in der Entwicklungsumgebung alle Annotationen verwalten und überhaupt nur disziplinierte Annotationen erlauben. Mit entsprechender Infrastruktur können disziplinierte Annotationen auch sprachübergreifend angeboten werden [KAT+09].

Produktlinienorientierte Typsysteme. Mit angepassten Typsystemen für Produktlinien ist es möglich, alle Produktvarianten einer Produktlinie auf Typsicherheit zu prüfen, ohne jede Variante einzeln zu kompilieren. 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). Im Rahmen der Dissertation wurde ein produktlinienorientierte Typsystem für annotationsbasierte Produktlinien entwickelt [KA08, KATS11].

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 dass 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 (wozu effiziente SAT Solver eingesetzt werden). 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.

Durch Informationen über den Variantengenerierungsprozess kann ein produktlinienorientierte Typsystem die gesamte Produktlinie sehr effizient prüfen. Die exponentielle Komplexität in Produktlinien wird in praktischen Fällen weitgehend vermieden; die gesamte Produktlinie kann mit nur geringem, nahezu konstantem Overhead gegenüber der Typprüfung von einer einzelnen Variante geprüft werden [KATS11].

Wie bei Sichten emulieren produktlinienorientierte Typsysteme wieder einige Vorteile von modularisierten Implementierungen. Anstelle von 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.

Schwer verständlicher Quelltext. Wärend Sichten bereits einen Beitragen geleistet haben Quelltext leichter lesbar zu machen, wurden zudem zusätzlich alternative Darstellungsformen exploriert und evaluiert. Etwa können Annotationen statt durch textuelle Anweisungen durch Hintergrundfarben dargestellt werden, wie in Abbildung 6 illustriert.

```
1 class Stack {
2   void push(Object o, Transaction txn) {
3     if (o==null || txn==null) return;
4     Lock l=txn.lock(o);
5     elementData[size++] = o;
6     l.unlock();
7     fireStackChanged();
8   }
9 }
```

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

Damit wird die textuelle Vermischung von Hostsprache und Variabilitätssprache vermieden. Hintergrundfarben und ähnliche visuelle Repräsentationen 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. In einem kontrollierten Experiment mit 43 Studenten konnten wir zeigen dass Hintergrundfarben bei bestimmten Aufgaben die Verarbeitungsgeschwindigkeit gegenüber textuellen Annotationen signifikant beschleunigt.

Trotz aller grafischen Verbesserungen und Werkzeugunterstützung sollte man nicht aus dem Auge verlieren, dass Präprozessoren nicht als Rechtfertigung dafür dienen darf, Quelltext gar nicht mehr zu modularisieren. Sie erlauben nur mehr Freiheit und zwingen Entwickler 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 und Messungen in 30 Millionen Zeilen C Quelltext) selten Annotationen zu mehr als zwei oder drei Features, so dass man auch mit einfachen grafischen Mitteln viel erreichen kann.

Vorteile von Präprozessoren und Integration. Neben allen Problemen haben Präprozessoren auch 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. 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 viele Sprachen *gleichförmig* eingesetzt 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 verhindern Präprozessoren nicht die traditionellen Möglichkeiten zur Trennung von Belangen. Eine primäre (dominante) Dekomposition in Module 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 [KLM+97, TOHS99]. Eben solche Probleme können mit verteilten Quelltext und später Sichten auf den Quelltext leicht gelöst werden.

Schlußendlich ist auch eine Integration von Annotationen mit eher klassichen Kompositionsbasierten Entwicklungsmodellen möglich. Im Rahmen der Arbeit wurden beide Ansätze gegenüberstellt und festgestellt dass beide sich ergänzen können. Unter Anderm haben wir vorgestellt wie Annotationen vollautomatisch und inkrementell in Feature-Module transformiert werden können (und vice versa) [KAK09]. Ein typisches Anwendungsszenario ist es Produktlinien zunächst mit Annotationen zu implementieren und dann, bei Bedarf, schrittweise in eine modularere Implementierungsform zu überführen.

Werkzeuge und Evaluation. Alle vorgestellten Verbesserungen – Sichten auf Features und Produktvarianten, disziplinierte Annotationen, ein produktlinienorientiertes Typsystem und visuelle Darstellung von Annotationen – sind in unserem Produktlinienwerkzeug-Prototyp CIDE implementiert. Die unterschiedlichen Konzepte und ihre Kombinationen wurden in verschiedenen Projekten evaluiert. Unter anderem wurden im Rahmen der Dissertation 13 nicht-triviale Fallstudien (in den Sprachen Java, C, C++, Haskell, Python, XML, HTML und AntLR) mit CIDE implementiert um unter andere die Ausdrucksfähigkeit disziplinierter Annotationen und die Skalierbarkeit des Typsystems zu demonstrieren. Darunter befindet sich auch etwa eine in Features zerlegte Version des Datenbankmanagementsystems Berkeley DB. Die Korrektheit des Typsystems wurde zudem, für ein auf Featherweight Java basierendes Subset von Java, mit dem Theorembeweiser *Coq* formal bewiesen. Den Nutzen von Farben haben wir in einem kontrollierten Experiment mit 43 Studenten empirisch evaluiert und Aussagen zur Verteilung von Annotationen in bestehendem Quelltext mit 40 Open-Source Projekten mit insgesamt 30 Millionen Zeilen C Code evaluiert. CIDE ist steht unter einer Open Source Lizenz und ist unter `http://fosd.de/cide` zusammen mit allen Fallbeispielen zur Verfügung.

4 Zusammenfassung

Unsere Kernmotivation für diesen Beitrag war es zu zeigen, dass Präprozessoren für die Produktlinienentwicklung durchaus Potential haben. Mit Werkzeugunterstützung – Sichten, Typsysteme, Visualisierung, etc. – können viele der Probleme, für die sie kritisiert werden, leicht behoben oder zumindest abgeschwächt werden. Obwohl wir nicht alle Probleme lösen können (beispielsweise ist ein separates Kompilieren von Features nicht möglich), haben Präprozessoren auch einige Vorteile, insbesondere das einfache Programmiermodell und die Sprachunabhängigkeit und stellen damit nach unserer Auffassung eine ernstzunehmende Alternative für die Produktlinienimplementierung.

Als Abschluss möchten wir noch einmal betonen, dass wir selber nicht endgültig entscheiden 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, möchten wir im Gegenzug ermuntern, nach Verbesserungen Ausschau zu halten bzw. diese von den Werkzeugherstellern einzufordern.

Literatur

[ABGM02] David L. Atkins, Thomas Ball, Todd L. Graves und Audris Mockus. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Trans. Softw. Eng. (TSE)*, 28(7):625–637, 2002.

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

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill und Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.

[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, 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, 2008.

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

[KAK09] Christian Kästner, Sven Apel und Martin Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, Seiten 157–166, 2009.

[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)*, Seiten 175–194, 2009.

[KATS11] Christian Kästner, Sven Apel, Thomas Thüm und Gunter Saake. Type Checking Annotation-Based Product Lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2011. accepted for publication.

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

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

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

[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 Engineering (ICSE)*, Seiten 107–119, 1999.

Christian Kästner ist wissenschaftlicher Mitarbeiter in der Gruppe für Programmiersprachen und Softwaretechnik an der Philipps Universität Marburg. Er promovierte 2007 bis 2010 am Datenbanklehrstuhl von Gunter Saake an der Otto-von-Guericke-Universität Magdeburg. Er interessiert sich für Korrektheit und Verständlichkeit von Systemen mit Variabilität, insbesondere Implementierungsmechanismen, Werkzeuge, Typsysteme, Featureinteraktionen, und Refactoring. Er ist Autor und Coautor von über 50 begutachteten wissenschaftlichen Beiträgen.