Dynamische Klassendiagramme - Nutzung der Metapher vom „Konsumieren und Produzieren“ in BlueJ

Axel Schmolitzky und Chris Stahlhut, Universität Hamburg
{schmolit, 6stahlhu}@informatik.uni-hamburg.de

Zusammenfassung

Konsumieren und Produzieren sind zwei Seiten derselben Medaille. Wir untersuchen seit einigen Jahren die Metapher vom Konsumieren und Produzieren (kurz: K&P-Metapher) im Umfeld der Lehre objektorientierter Programmierung.

In diesem Artikel stellen wir eine Erweiterung der für die Lehre der objektorientierten Programmierung entwickelten Entwicklungsumgebung BlueJ vor. Diese Erweiterung führt die bereits in BlueJ implizit vorhandene Unterstützung der K&P-Metapher konsequent fort, indem sie auch im BlueJ-Klassendiagramm dynamisch die Unterscheidung zwischen dem Konsumieren und dem Produzieren einer Klasse ermöglicht.

Einleitung

Konsumieren und *Produzieren* sind zwei Seiten derselben Medaille. Ein Buch kann gelesen (konsumiert) und geschrieben (produziert) werden, ein Tisch benutzt oder gebaut, eine Software genutzt oder entwickelt werden.

Damit ein Artefakt (ein Gegenstand, ein Text, ein Konzept) konsumiert werden kann, muss es zuerst produziert werden. Bevor ein Buch gelesen werden kann, muss es geschrieben werden. Bevor ein Tisch für eine Mahlzeit genutzt werden kann, muss er gebaut werden. Bevor Software benutzt werden kann, muss sie programmiert werden. Im Allgemeinen gilt dabei, dass das Produzieren deutlich anspruchsvoller ist als das Konsumieren; eine für sich genommen triviale Erkenntnis.

Wir untersuchen seit einigen Jahren die Metapher vom Konsumieren und Produzieren (kurz: *K&P-Metapher*) im Umfeld der Lehre der objektorientierten Programmierung (Späh u. Schmolitzky, 2007). Bei unserer Didaktik machen wir uns dabei zunutze, dass Konsumieren leichter als Produzieren ist: Häufig vermitteln wir unseren Studierenden ein zu verstehendes Konzept nur in einer „abgespeckten“ Version (lassen sie es nur konsumieren), um es erst zu einem späteren Zeitpunkt ausführlich zu behandeln (sie so damit vertraut zu machen, dass sie es produktiv einsetzen können). Dieser simple Kniff ermöglicht es an etlichen Stellen, die gerade bei der Objektorientierung häufig zirkulären Abhängigkeiten von Grundkonzepten etwas aufzubrechen.

In (Schmolitzky u. Züllighoven, 2007) haben wir dazu bereits einige Beispiele aus dem Bereich der Programmierung angeführt. Unter anderem haben wir *Generizität* in Java als ein Konzept aufgeführt, das für den Umgang mit generischen Sammlungen in Java anfänglich nur konsumiert zu werden braucht (sprich: wie gibt man bei der Deklaration einer Sammlung den Typ ihrer Elemente an), und können so in einer Erstsemesterveranstaltung die sehr nützlichen Sammlungen des *Java Collections Framework* (JCF) relativ früh thematisieren. Erst deutlich später (im darauf folgenden Semester) thematisieren wir Generizität so, dass die Studierenden selbst generische Klassen produzieren können.

Zur Verdeutlichung kann im Kontrast dazu eine andere Sicht dargestellt werden, wie sie beispielsweise in einem Lehrbuch von Liang umgesetzt ist (Liang, 2010). Hier wird das JCF sehr spät (in Kapitel 22) angesprochen; vermutlich deshalb, weil die Autoren der Meinung sind, dass vor dem Umgang mit generischen Sammlungen grundsätzlich geklärt sein muss, was Generizität ist (Kap. 21). Ein weiteres Beispiel für dieses Denken findet sich in (Ashok u. a., 2008). Auf diese Weise wird ein zentrales und nützliches Konzept (Sammlungen) aufgrund einer vermeintlichen formalen Abhängigkeit von einem anderen Konzept (Generizität) aus unserer Sicht unnötig „nach hinten geschoben“.

In diesem Artikel liegt der Fokus nicht auf dem Konsumieren und Produzieren von Konzepten in der Lehre, sondern auf dem Konsumieren und Produzieren von Klassen in der objektorientierten Programmierung (OOP). Während ein Dozent die Entscheidung, welcher Anteil eines Konzeptes konsumierend und

welcher produzierend betrachtet werden soll, mehr oder weniger willkürlich treffen kann, gibt es bei den Klassen der OOP klar definierte Eigenschaften, die für Klienten einer Klasse einerseits und für die Entwickler einer Klasse andererseits relevant sind. Dies macht sie einer automatisierten Darstellung zugänglich.

Im nächsten Abschnitt werden wir zuerst einige Begriffe benennen, die für die nachfolgende Diskussion grundlegend sind. Abschnitt 3 stellt dar, auf welche Weise die K&P-Metapher bisher in der Lehrumgebung BlueJ umgesetzt ist. Abschnitt 4 zeigt, wie dies konsequent fortgeführt werden kann, und stellt eine prototypische Implementation vor. Abschnitt 5 diskutiert, wie die bisherigen Erkenntnisse gewinnbringend eingesetzt werden könnten. Abschnitt 6 fasst die Arbeit zusammen.

K&P in der Objektorientierung

Ein zentraler Gedanke der Objektorientierung ist, dass die Zuständigkeiten für die Aufgaben eines Systems auf verschiedene Objekte verteilt werden und diese Objekte sich gegenseitig benutzen. Ein Objekt, das ein anderes benutzt, bezeichnen wir als einen *Klienten*; ein Objekt, das benutzt wird, als einen *Dienstleister*. Jedes Objekt kann sowohl Klient als auch Dienstleister sein.

Ein weiterer zentraler Gedanke der Objektorientierung (und der Softwaretechnik) ist, dass bei einem Objekt zwischen seiner *Schnittstelle* und seiner *Implementation* unterschieden werden kann. Über seine Schnittstelle bietet ein Objekt seine Dienstleistungen an; in der Implementation wird dieses Angebot umgesetzt. Diese Unterscheidung wird deutlicher, wenn der Fokus von den Objekten auf die sie definierenden Klassen gerichtet wird: In Java wird systematisch zwischen der Schnittstelle einer Klasse und ihrer Implementation unterschieden, indem die mit dem Werkzeug *javadoc* erzeugte Dokumentation einer Klasse üblicherweise ihre Schnittstelle darstellt, während der Quelltext der Klasse ihre vollständige Implementation wiedergibt.

Ein Programmierer eines Klienten, der Exemplare einer anderen Klasse ausschließlich *benutzen* will, konsumiert die andere Klasse lediglich. Er muss idealerweise nur ihre Schnittstelle kennen (die auch explizit in Form eines Java-Interfaces beschrieben sein kann), um qualifiziert als Klient auftreten zu können. Erst wenn ein Programmierer die andere Klasse *warten* muss (Fehler beheben muss, ihre Funktionalität erweitern muss, etc.), muss er ihren Quelltext bearbeiten und kann in Bezug auf ihre Dienstleistungen produzierend aktiv werden.

Die Essenz lautet: eine Klasse konsumieren erfordert nur die Kenntnis ihrer Schnittstelle; eine Klasse produzieren erfordert zwingend Einblick in ihre Implementation. Beide Sichten auf eine Klasse finden sich auch in der Entwicklungsumgebung *BlueJ* wieder, allerdings nur unvollständig.

K&P in BlueJ bisher

BlueJ ist eine für die Lehre des objektorientierten Programmierparadigmas entworfene Entwicklungsumgebung (engl. abgekürzt: IDE) für Java. Sie ist wie Eclipse frei verfügbar, hat aber im Gegensatz zu Eclipse nicht den Anspruch, eine vollwertige IDE für die professionelle Softwareentwicklung darzustellen; BlueJ ist gezielt minimalistisch ausgelegt, um Programmieranfängern die Kernkonzepte der objektorientierten Programmierung zu vermitteln und diese nicht mit umfangreicher Funktionalität zu überwältigen (Kölling u. a., 2003). BlueJ wird weltweit inzwischen von fast 1000 Hochschulen eingesetzt (BlueJ, 2010).

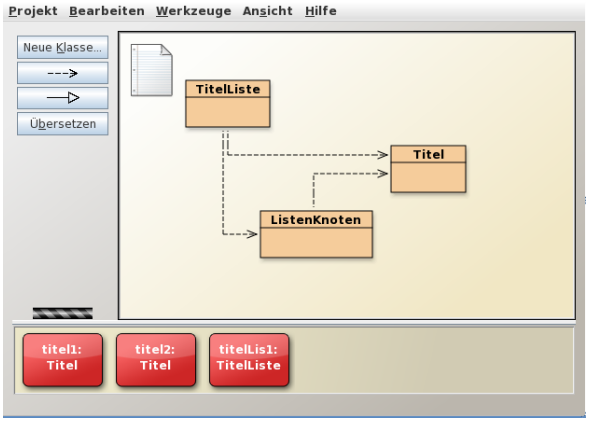

Abbildung 1: Das Hauptfenster von BlueJ

BlueJ stellt die Klassen eines Java-Projektes, anders als übliche IDEs, nicht in Listenform dar, sondern in Form eines einfachen UML-Klassendiagramms (Abbildung 1). BlueJ visualisiert mit diesem Diagramm nicht nur die Struktur des Projektes, sondern bietet auch direkte Interaktionsmöglichkeiten mit den dargestellten Klassen und ihren Exemplaren: Es können interaktiv Exemplare dieser Klassen erzeugt und an diesen Exemplaren dann alle Methoden aufgerufen werden.

Aufgrund der Möglichkeit, Klassen und Objekte interaktiv zu benutzen, ohne Quelltext schreiben zu müssen, kann bei der Benutzung von BlueJ bereits zwischen konsumierenden und produzierenden Personen unterschieden werden. Eine konsumierende Person kann Klassen und Objekte benutzen, ohne wissen zu müssen, wie diese mit Java erstellt werden. Nur eine produzierende Person, also jemand, der Klassen in ihrem Quelltext verfasst, muss mehr über Java wissen (Syntax, Bibliotheksklassen, etc.). Der *Object-First-Ansatz* von Barnes und Kölling (Barnes u. Kölling, 2009) basiert sehr stark auf diesen Möglichkeiten.

Der in BlueJ integrierte Editor zum Bearbeiten von Klassendefinitionen bietet zwei Sichten: eine konsumierende, die die von *javadoc* für die Klasse erstellte (und nicht interaktiv änderbare) Schnittstellenbeschreibung darstellt (*Schnittstellensicht*); und eine

produzierende Sicht, die den vollständigen Quelltext der Klasse darstellt und bearbeitbar macht (*Quelltextsicht*).

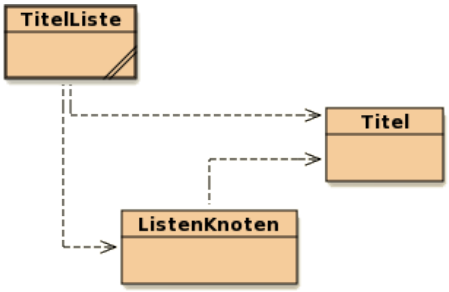

Abbildung 2: Ein Klassendiagramm in BlueJ

Für die nachfolgende Diskussion soll ein Beispiel das Verständnis erleichtern: In einem Lehrprojekt, das fachlich einen MP3-Player modellieren soll, werden neben den abzuspielenden *Musiktiteln* auch *Wiedergabelisten* benötigt. Das BlueJ-Projekt dazu enthält deshalb neben der Klasse `Titel` auch eine Klasse `TitelListe`. Objekte der Klasse `TitelListe` sollen alle Eigenschaften aufweisen, die eine Liste üblicherweise anbietet: Die Reihenfolge der Elemente ist frei manipulierbar und es können beliebig Duplikate eingefügt werden (eine Semantik, die offensichtlich gut zu einer Wiedergabeliste von Musiktiteln passt). Entsprechend bietet ihre Schnittstelle Operationen wie *einfuegenAnPosition*, *entferneAnPosition* etc. an. Die Klasse `TitelListe` sei intern als doppelt verkettete Liste von *Knotenelementen* umgesetzt, die neben den Verkettungsreferenzen jeweils eine Referenz auf einen `Titel` halten sollen (in dem Lehrprojekt geht es um verschiedene Implementationsformen für Listen). Diese Knotenelemente seien in der Klasse `ListenKnoten` realisiert. Abbildung 2 zeigt diese Zusammenhänge in einem BlueJ-Klassendiagramm.

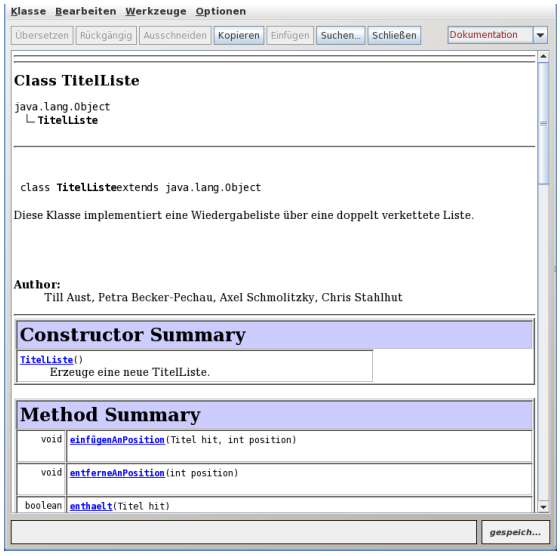

Abbildung 3: Die Schnittstellensicht im BlueJ-Editor

Wenn ein Klient der Klasse `TitelListe` geschrieben werden soll (etwa eine Komponente, die Wiedergabelisten auf dem Bildschirm darstellt), kann der Programmierer diese Klasse im BlueJ-Editor betrachten, um die aufrufbaren Operationen vor sich zu haben. Da den Programmierer somit nur die Schnittstelle interessieren muss, kann dazu die *Schnittstellensicht* des Editors gewählt werden (Abbildung 3).

Diese Sicht verbirgt alle Implementationsdetails, unter anderem auch die Abhängigkeit zu der Klasse `ListenKnoten`; ein Effekt, der im Sinne des Geheimnisprinzips nur erwünscht sein kann.

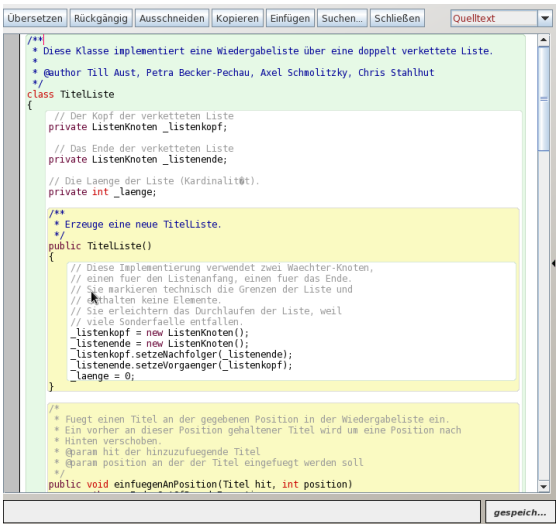

Abbildung 4: Die Quelltextsicht im BlueJ-Editor

Erst wenn die Implementation der Klasse `TitelListe` selbst betrachtet oder bei Wartungsarbeiten verändert werden soll, muss in die *Quelltextsicht* des Editors gewechselt werden. Diese Sicht unterscheidet sich nur geringfügig (primär durch die Blockhervorhebung, Abbildung 4) von der anderer Editoren.

BlueJ unterstützt somit mit den beiden Sichten im Editor direkt die K&P-Metapher: Klienten können eine Klasse nur konsumieren (müssen nur die Informationen sehen, die für eine Benutzung notwendig sind: die Schnittstellensicht), während Wartungsprogrammierer den Zugriff auf den vollen Quelltext benötigen, um produzierend tätig werden zu können.

K&P in BlueJ fortgeführt

Die Klasse `TitelListe` hat eine in ihrer Schnittstelle definierte Benutzt-Beziehung zu der Klasse `Titel`, während die Beziehung zu der Klasse `ListenKnoten` nicht zu ihrer Schnittstelle gehört. Die Klasse `ListenKnoten` benutzt ebenfalls die Klasse `Titel`. Alle drei Abhängigkeiten werden im BlueJ-Klassendiagramm durch Pfeile dargestellt, siehe Abbildung 2.

Da es sich bei der Klasse `Titel` um den Elementtypen der Liste handelt, ist die Abhängigkeit in der

Schnittstelle der Klasse `TitelListe` unumgänglich; erwartungsgemäß muss ein Klient einer `TitelListe` damit rechnen, `Titel` als Parameter zu übergeben (obwohl nicht alle Operationen der Klasse explizit mit Titeln umgehen müssen, siehe oben).

Es ist hingegen für einen Klienten eher nachrangig, wie die Klasse `TitelListe` implementiert ist. Der Umstand, dass dies in einer verketteten Liste passiert, ist für die Benutzung üblicherweise irrelevant. Dies wird auch in den beiden Editor-Darstellungen der Klasse deutlich: In der Schnittstellensicht ist lediglich die Klasse `Titel` aufgeführt, während im Quelltext der Klasse auch die Klasse `ListenKnoten` auftaucht.

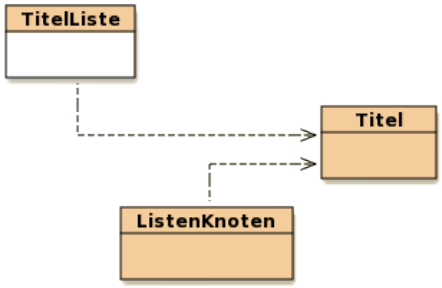

Abbildung 5: Klientensicht der `TitelListe`

Diesen Unterschied könnten wir auch im Klassendiagramm darstellen, indem wir eine *Klientensicht* auf die Klasse `TitelListe` postulieren: in dieser Sicht besteht lediglich eine Abhängigkeit zur Klasse `Titel` (Abbildung 5). Die Darstellung der `TitelListe` in Abbildung 2, in der alle Abhängigkeiten der Klasse gezeigt werden, nennen wir hingegen ihre *Produzentensicht*. Da Produzenten einer Klasse ihren Quelltext erstellen/bearbeiten, sollten für sie alle Abhängigkeiten sichtbar sein.

Die Darstellung in Abbildung 5 stammt aus einer von uns entwickelten Erweiterung von BlueJ, die ein interaktives Umschalten zwischen den beiden Sichten auf *jede Klasse* im Klassendiagramm ermöglicht. Wir nennen diese Erweiterung im Folgenden kurz *BlueJ-CP*.

Die grundsätzliche Möglichkeit zweier Sichten auf jede Klasse führt in BlueJ-CP dazu, dass es nicht nur ein statisches, sondern eine Vielzahl von möglichen Klassendiagrammen für dasselbe Projekt gibt. Durch die interaktive Möglichkeit des Wechsels der Sichten wird das statische Klassendiagramm zu einem dynamischen Klassendiagramm. Bei einem Projekt mit n Klassen ergeben sich bis zu 2^n mögliche Klassendiagramme. Das zu einem bestimmten Zeitpunkt angezeigte Klassendiagramm bezeichnen wir als das *momentane Klassendiagramm*.

Um die Unterscheidung visuell stärker zu verdeutlichen, haben wir für die Darstellung der Klientensicht auch das Klassensymbol leicht verändert: der untere Teil erscheint weiß. Für die Produzentensicht haben wir nichts am Symbol verändert, weil dies der bisherigen Darstellung in BlueJ entspricht.

Ausblenden von Klassen

Da im Kontext unseres Beispiels allein die Klasse `TitelListe` die Klasse `ListenKnoten` benutzt, kann letztere als ein ausgelagertes Implementationsdetail betrachtet werden. Aus Sicht eines Klienten der `TitelListe` ist nicht nur die Benutzt-Beziehung zwischen den beiden Klassen irrelevant, sondern die komplette Klasse `ListenKnoten`; sie muss in der Konsumentensicht der Klasse `TitelListe` nicht angezeigt werden. Abbildung 6 zeigt eine in dieser Hinsicht konsequente Darstellung des Projekts mit der Konsumentensicht der `TitelListe`. Dieses Diagramm ist deutlich einfacher, ein von uns erwünschter Effekt; durch das Umschalten von Klassen in ihre Klientensicht soll die Komplexität des Diagramms reduziert werden können.

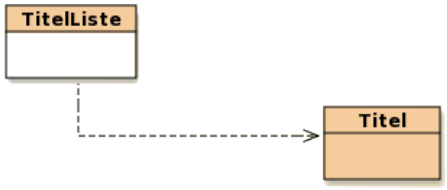

Abbildung 6: Die konsequente Klientensicht

Für unsere Erweiterung ergibt sich somit ein rein formales Kriterium, nach dem Klassen verborgen werden können: Zeigt das momentane Klassendiagramm eine Abhängigkeit zu einer Klasse `A`, so ist diese Klasse anzuzeigen. Dies ist von der Art der Abhängigkeit (basierend auf der Schnittstelle oder nicht) unabhängig. Benutzen zwei Klassen `B` und `C` dieselbe Klasse `A`, so ist `A` anzuzeigen, wenn min. eine eingehende Abhängigkeit angezeigt wird. Eine Klasse wird also nur verborgen, wenn sie im momentanen Klassendiagramm keine eingehenden Abhängigkeiten besitzt.

Wird ein Projekt erstmalig in BlueJ-CP geöffnet, könnte für alle Klassen die Klientensicht gewählt werden, in der Hoffnung, dass dies einen ersten „high-level“ Überblick vermittelt, weil transitiv alle Implementationsdetails ausgeblendet sind. Bei der Wahl der anfangs anzuzeigenden Klassen sind jedoch einige Dinge zu beachten.

Einstiegspunkte für Java klassisch

Einen offensichtlichen Einstiegspunkt in eine allgemeine Java-Anwendung bietet die `main`-Methode. Klassen, die eine Methode mit einer speziellen Signatur anbieten (es kann mehrere in einem Projekt geben), bilden somit einen guten Ausgangspunkt. Im Idealfall verfügt genau eine Klasse über eine `main`-Methode. Diese Klasse wird in der Klientensicht angezeigt und ist somit möglicherweise sogar die einzige Klasse, die im momentanen Klassendiagramm angezeigt werden muss. Erst durch das Umschalten bei dieser Klasse auf die Produzentensicht werden die Klassen sichtbar, die im Rumpf der `main`-Methode benutzt werden. Auf

diese Weise kann ein Klassendiagramm nach und nach „entfaltet“ werden.

Java-Pakete verfügen ebenfalls über eine Schnittstelle, sie besteht aus allen *öffentlichen* (mit dem Modifikator `public` deklarierten) *Klassen*. Nur die auf diese Weise „exportierten“ Klassen können aus anderen Paketen benutzt werden, alle anderen Klassen gehören zur Implementation eines Paketes.

Die öffentlichen Klassen eines Pakets (insbesondere der Pakete, die keine `main`-Methode enthalten) stellen somit weitere sinnvolle Einstiegspunkte dar, die initial bei der Darstellung eines Paketes sichtbar sein sollten. Auch sie sollten anfangs in der Klientensicht dargestellt werden (inklusiv ihre transitiven Schnittstellen-Beziehungen) und können bei Bedarf aufgefaltet werden.

Eine Ausnahme stellen JUnit-Testklassen dar. Diese Klassen müssen nur deshalb öffentlich sein, damit sie vom JUnit-Rahmenwerk gefunden und ausgeführt werden können, und gehören somit nicht zur Schnittstelle eines Pakets. Zur Reduktion der Diagramm-Komplexität können sie gesondert ausgeblendet werden.

Einstiegspunkte für BlueJ

Das Problem mit beiden Einstiegsheuristiken (`main`- und öffentliche Klassen) ist, dass sie nicht gut für echte BlueJ-Projekte funktionieren: Da in BlueJ Objekte interaktiv erzeugt werden und die Benutzer direkt an diesen Objekten Methoden aufrufen können, werden `main`-Methoden nur in Ausnahmefällen implementiert; siehe dazu unter anderem (Kölling u. Rosenberg, 2001). Und die Lehre-Projekte in BlueJ bestehen auch nur sehr selten aus mehreren Paketen (obwohl BlueJ diese auch darstellen kann), sondern meist nur aus einem, typischerweise dem unbenannten Paket.

Für echte BlueJ-Projekte ergibt sich deshalb die auf den ersten Blick paradoxe Schlussfolgerung, dass genau die Klassen initial dargestellt werden sollten, die nicht von anderen Klassen benutzt werden. Denn bei diesen liegt die Vermutung nahe, dass sie für den interaktiven Gebrauch entwickelt wurden und somit gute Einstiegspunkte sind. Auf diese Weise ist auch sichergestellt, dass eine gerade neu erstellte, aber noch unbenutzte Klasse angezeigt wird.

Einsatzmöglichkeiten

Die Erkenntnisse aus unserem anfänglich rein gedanklichen Experiment mit BlueJ und der realen Umsetzung in BlueJ-CP weisen uns zwei interessante Wege: Zum einen sollte untersucht werden, inwieweit BlueJ-CP sinnvoll in der Programmierausbildung eingesetzt werden kann, um die Konzepte Schnittstelle und Implementation anschaulicher zu vermitteln. Zum anderen scheint uns das Konzept der verschiedenen Sichten auf Klassendiagramme übertragbar auf andere Werkzeuge.

In der Programmierausbildung

Idealerweise würden die Teilnehmer an einer einführenden Programmierveranstaltung gar nicht merken, dass sie mit einem veränderten BlueJ arbeiten, und an passender Stelle würden die zusätzlichen Möglichkeiten von BlueJ-CP zum Einsatz kommen. Wir haben in dieser Hinsicht bisher noch keine belastbaren Erfahrungen sammeln können.

Immerhin haben wir mit 23 freiwilligen Teilnehmern einer Erstsemesterveranstaltung eine kleine Studie durchgeführt, in der sie eine Vorversion von BlueJ-CP alternativ zu BlueJ bei der Lösung einer Übungsaufgabe einsetzen konnten (Stahlhut, 2010). Ein Ergebnis dieser Studie war, dass 21 der 23 Probanden der Meinung waren, dass BlueJ-CP beim Verständnis der Konzepte Schnittstelle und Implementation unterstützt, während 2 Probanden es nicht als unterstützend empfunden haben. 18 von 23 empfanden es als angemessen, Klassen benutzungsabhängig verbergen zu können (bei 5 Enthaltungen).

Die Erkenntnisse aus dieser Studie sind primär in die Weiterentwicklung von BlueJ-CP eingeflossen; aber das Feedback aus den geführten Interviews lässt deutlich darauf schließen, dass die interaktive Möglichkeit zweier Sichten auf eine Klasse, intelligent eingesetzt, beim Verständnis von Kapselung und Schnittstellen hilfreich sein kann.

In der Visualisierung von Software

Seit wir BlueJ-CP zur Visualisierung von Java-Projekten zur Verfügung haben, erscheinen uns die aus Quelltexten generierten Klassendiagramme anderer UML-Werkzeuge merkwürdig „flach“: Sie zeigen immer alle Abhängigkeiten, unabhängig von ihrer Einordnung als Schnittstellen- oder als Implementations-Abhängigkeit. Die Kriterien für diese Einordnung sind rein formal und können ohne weiteres auch in anderen Werkzeugen als in BlueJ dafür genutzt werden, automatisiert verschiedene Abstraktionen desselben Quelltextes anzubieten.

Ein interessantes Vorhaben könnte somit sein, die mit BlueJ-CP gesammelten Erkenntnisse auch in anderen Werkzeugen umzusetzen; zumindest in solchen, die quelloffen zur Verfügung stehen. Wir sehen dabei insbesondere die Möglichkeit im Vordergrund, ein bestehendes System mit Hilfe der beschriebenen Mittel nach und nach interaktiv aufzufalten und somit einen objektorientierten Entwurf zu explorieren. Inwieweit dies wirklich beim Einlesen und Verstehen hilfreich ist, gälte es zu untersuchen.

Ein Werkzeug, das explizit zur Untersuchung der K&P-Metapher entwickelt wurde, ist *jMango* (Späh, 2008). Es bietet neben einer filtergestützten Visualisierung von Java-Systemen auch eine Unterstützung für die Modellierung von Konzept-Abhängigkeiten. Die in einer Erstsemesterveranstaltung vermittelten Konzepte wurden mit Hilfe von jMango systematisch modelliert (Albers, 2009).

Da jMango als eine Eclipse-RCP-Anwendung konstruiert ist, wäre es denkbar, gezielt die Visualisierungsmöglichkeiten für allgemeine Java-Projekte als Plugin für Eclipse anzubieten. Auf diese Weise könnte das dynamische Falten von Klassendiagrammen auf breiter Ebene für Java-Eclipse-Projekte zur Verfügung gestellt werden. Bisher fehlt in jMango allerdings noch ein guter Algorithmus für das automatische Layouten von Klassendiagrammen (zumindest als Ausgangspunkt), so dass ein gebrauchsfertiges Plugin noch Forschungs- und Entwicklungsaufwand erfordert.

Zusammenfassung

Das Konsumieren einer Klasse von ihrem Produzieren zu unterscheiden ist ein Kernkonzept der Objektorientierung, dem – softwaretechnisch breiter betrachtet – die Trennung von Schnittstelle und Implementation zugrunde liegt. Die IDE BlueJ macht diese Unterscheidung in ihrem Editor deutlich, indem sie zwei Sichten auf eine Klasse anbietet.

In diesem Artikel haben wir dargestellt, wie BlueJ erweitert werden kann, um diesen Unterschied auch im BlueJ-Klassendiagramm zu visualisieren. Konzeptuell ergab sich daraus, dass es für ein Klassendiagramm mit n Klassen 2^n verschiedene Darstellungen geben kann, die ein System auf mehreren Abstraktionsebenen darstellen können. Diesen Umstand gilt es weiter zu untersuchen, sowohl in seinen Einsatzmöglichkeiten in der Lehre als auch in der Visualisierung von Softwaresystemen.

Literatur

[Albers 2009] ALBERS, T.: *Evaluation eines prototypischen Werkzeuges zur Visualisierung von Konzeptabhängigkeiten unter Berücksichtigung der Metapher „Konsumieren vor Produzieren“*, Universität Hamburg, Bachelorarbeit, 2009

[Ashok u. a. 2008] ASHOK, S ; KIONG, D ; POO, D: *Object-Oriented Programming and Jav*. Springer Verlag, 2008. – 2. Auflage

[Barnes u. Kölling 2009] BARNES, D. J. ; KÖLLING, M.: *Objects First with Java — A Practical Introduction using BlueJ*. Harlow, United Kingdom : Prentice Hall / Pearson Education, 2009. – 4. Auflage

[BlueJ 2010] BLUEJ: *BlueJ — The interactive Java environment*. Version: 2010. `http://www.bluej.org`. – zuletzt besucht: 2010-12-16

[Kölling u. a. 2003] KÖLLING, M. ; QUIG, B. ; PATTERSON, A. ; ROSENBERG, J.: The BlueJ system and its pedagogy. In: *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology* (2003)

[Kölling u. Rosenberg 2001] KÖLLING, M. ; ROSENBERG, J.: Guidelines for Teaching Object Orientation with Java. In: *Information Technology in Computer Science Education (ITiCSE 2001)*, 2001

[Liang 2010] LIANG, Y. D.: *Introduction to Java Programming*. Harlow, United Kingdom : Prentice Hall / Pearson Education, 2010. – 8. Auflage

[Schmolitzky u. Züllighoven 2007] SCHMOLITZKY, A. ; ZÜLLIGHOVEN, H.: Einführung in die Softwareentwicklung - Softwaretechnik trotz Objektorientierung? In: *A. Zeller and M. Deininger (Hrsg.), Software Engineering im Unterricht der Hochschulen (SEUH)*, 2007

[Späh 2008] SPÄH, C.: *Konsumieren und Produzieren als nützliche Metaphern in der Softwaretechnikausbildung und der Exploration von Klassenstrukturen*, Universität Hamburg, Diplomarbeit, 2008

[Späh u. Schmolitzky 2007] SPÄH, C. ; SCHMOLITZKY, A.: „Consuming before Producing“ as a Helpful Metaphor in Teaching Object-Oriented Concepts. In: *Eleventh Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts*, 2007

[Stahlhut 2010] STAHLHUT, C.: *Die Metapher „Konsumieren und Produzieren“ in BlueJ*, Universität Hamburg, Bachelorarbeit, 2010