Vererbung bei der Sanierung objektorientierter Systeme

Rainer Neumann [1]
Rainer.Neumann@gmx.net

Benedikt Schulz [2]
bschulz@fzi.de

Wolf Zimmermann [1]
zimmer@info.uni-karlsruhe.de

[1] Institut für Programmstrukturen und Datenorganisation (IPD)
Universität Karlsruhe
Zirkel 2, Postfach 6980, 76128 Karlsruhe

[2] Forschungszentrum Informatik Karlsruhe (FZI)
Forschungsbereich Programmstrukturen
Haid-und-Neu-Straße 10-14
76131 Karlsruhe

Kurzfassung

Vererbung spielt neben Kapselung eine zentrale Rolle innerhalb objektorientierter Programme. Dabei kann jedoch ein eklatanter Bruch zwischen konzeptueller Vererbung bei der Analyse und im Entwurf und deren Implementierung im Programm festgestellt werden. Dieses Papier beleuchtet die Probleme, denen sich Entwickler objektorientierter Systeme auch heute noch ausgeliefert sehen, und illustriert damit einen wesentlichen Teil der zukünftigen Sanierungsprobleme.

Einleitung

Während derzeit im Zusammenhang mit Softwaresanierung vor allem über die Behandlung von COBOL- oder Assembler-Programmen die Rede ist, gewinnt die Betrachtung objektorientierter Altlasten immer mehr an Bedeutung. Der Grund hierfür liegt in den folgenden Punkten:

1. Die objektorientierte Denkweise eignet sich gut zum Entwurf umfangreicher und komplexer Systeme.
2. Objektorientierte Programmiersprachen versprechen einen besseren Übergang vom Entwurf zur Implementierung.

In der Praxis zeigen sich leider einige gravierende Probleme dieses Idealbildes:

1. Objektorientierung ist eine Denkweise, die das Ergebnis eines Lernprozesses ist, und nicht das eines Programmierkurses.
2. Der Übergang vom Entwurf zur Implementierung ist oftmals aufwendig, auf keinen Fall aber trivial.

Im Rahmen dieses Papiers wird beschrieben, welche Konsequenzen sich aus diesem Übergang ergeben. Dies geschieht am Beispiel der Vererbung in objektorientierten Systemen.

Das zentrale Problem beim Übergang vom Modell zur Implementierung zeigt sich darin, daß im Modell explizit vorhandenes Wissen in der Implementierung oftmals nur implizit vorhanden ist – dies wird am Beispiel der spezialisierenden Vererbung aufgezeigt. Diese impliziten Annahmen können für die Stabilität eines Systems jedoch essentiell sein. Durch Wartung und Erweiterung kommen im Laufe der Zeit oftmals noch neue, ebenfalls implizite Annahmen hinzu, die teilweise den im System verborgenen Annahmen widersprechen können.

Die zentrale Aufgabe bei der Sanierung alter Systeme – und dies beschränkt sich nicht auf objektorientierte Programme – besteht aus der Rückgewinnung und Dokumentation des impliziten Wissens, das im Code verborgen ist. Die dann folgenden Transformationen sind meist einfache Operationen, die nicht weiter problematisch sind.

Der Rest dieses Papiers ist folgendermaßen strukturiert: Zunächst wird der konzeptionelle Bruch zwischen Vererbung im Modell und im Programm aufgezeigt. An einem einfachen Beispiel wird dann gezeigt, wie der Übergang vom Modell zum Code erfolgt. An dem entstehenden Programm wird das Einbringen einer neuen Anforderung demonstriert. Der so entstehende Code verdeutlicht die Probleme, die sich bei der Sanierung ergeben können.

Vererbung in Analyse und Entwurf

Bei der Analyse einer Problemdomäne und beim anschließenden Entwurf treten verschiedene Arten der Vererbung auf. Die Vererbungsbeziehungen lassen sich dabei nach Ziel und Art unterscheiden.

1. Ziel – zu welchem Zweck wird eine Vererbungsbeziehung eingeführt? Eine erbende Klasse kann eine geerbte Klasse z.B. erweitern, verfeinern, oder auch spezialisieren. Zudem wird Vererbung oftmals auch zur Wiederverwendung von Implementierungen eingesetzt.
2. Art – wie werden Objekte klassifiziert? Kann ein Objekt zu einer oder zu mehreren Klassen gehören? Kann ein Objekt seine Klassenzugehörigkeit zur Laufzeit ändern?

Abbildung 1 zeigt ein Beispiel eines Klassenmodells, das verschiedene Arten der Klassifikation enthält. Ein Mitarbeiter ist z.B. entweder im Innendienst oder im Außendienst beschäftigt, kann aber in den jeweils anderen Bereich wechseln. Die Klassifikation erfolgt also dynamisch und kann sich zur Laufzeit des Systems ändern. Im Gegensatz dazu ist die Unterteilung der Innendienstmitarbeiter in Sachbearbeiter und Administratoren statisch. Sogenannte multiple Klassifikation zeigt sich bei den Spezialisierten Sachbearbeitern – ein Sachbearbeiter kann u.U. verschiedene Spezialgebiete haben, die sich zudem auch noch dynamisch ändern können.

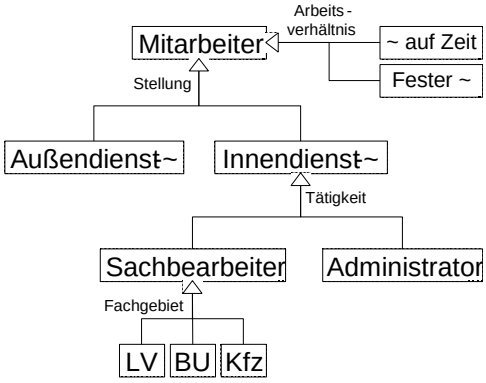
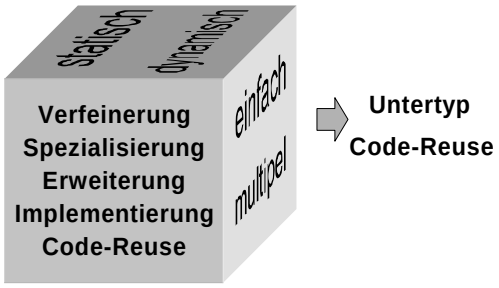


Abbildung 1: Vererbungsbeziehungen im Modell

Das Beispiel illustriert weiter, daß ein großer Teil der auftretenden Vererbungsbeziehungen zum Zweck der Spezialisierung verwendet wird. Wir werden später zeigen, daß in der Implementierung gerade bei dieser Art der Vererbungsbeziehung Probleme auftreten.

Vom Modell zum Programm

Während der Vererbungsbegriff bei der Analyse und auch noch bei der Modellierung eine sehr breite Varianz hat, steht dem auf Seite der Implementierung ein sehr eingeschränkter Vererbungsbegriff gegenüber. In der Regel beschränkt sich dieser auf die Untertypbeziehung, mit deren Hilfe sich einfache statische Klassifikation, und auch diese nur unzureichend darstellen läßt. Abbildung 2 illustriert diesen Zusammenhang.



(a) Modell *(b) Programm*

Abbildung 2: Begriffe in Modell und Programm

Durch diesen konzeptionellen Bruch entsteht beim Übergang vom Entwurf eines Systems zur Implementierung eine schwierige Aufgabe: Wie läßt sich der Modellierte Sachverhalt am besten abbilden. Eine mögliche Lösung bieten hier sogenannte Analyse- bzw. Entwurfsmuster im allgemeinen. Deren Verwendung erfordert jedoch bei den Entwicklern einen gewissen Erfahrungsschatz.

Vom Programm zum Modell

Der Weg vom Programm zum Modell ist ungleich schwerer, da bei der Implementierung explizite Informationen aufgegeben werden. Im besten Fall sind diese Informationen noch in Form strukturierter Kommentare vorhanden.

Bedingt durch langjährige Wartungsaufgaben wird die Drift zwischen Dokumentation bzw. Modell und Implementierung immer größer.

Im folgenden Abschnitt wird die zuvor beschriebene Problematik an einem einfachen Beispiel illustriert. Es wird gezeigt, welche Implementierungsmöglichkeiten bei der Umsetzung einer einfachen Vererbungsbeziehung bestehen. Dadurch werden Codemuster sichtbar, die bei der Rückgewinnung von Entwurfswissen wichtig sind.

Ein Beispiel

Im folgenden betrachten wir eine einzelne Art der Vererbung, nämlich die Spezialisierung. Anhand dieser einen Bedeutung von Vererbung werden die auftretenden Probleme einfach deutlich.

Spezialisierte Klassen

Am Beispiel der Sachbearbeiter wird die Bedeutung spezialisierender Klassifikation deutlich: Allen Sachbearbeitern allgemein sei die Fähigkeit, Schadensfälle zu regulieren. Dabei macht es jedoch kaum Sinn, einem Spezialisten für Kfz-Versicherungen die Regulierung eines Wohnungseinbruchs vorzulegen, da er diesen Schadensfall nur langsamer, evtl. aber gar nicht bearbeiten kann.

Der gleiche Sachverhalt läßt sich an dem immer wieder anzutreffenden Beispiel geometrischer Figuren zeigen: Ein Rechteck ist ein spezialisiertes Polygon, dessen Randpunkte nur sehr eingeschränkt verändert werden können. Ebenso, wie ein Sachbearbeiter keinen beliebigen Schadensfall regulieren kann, kann ein Rechteck nicht wie ein beliebiges Polygon behandelt werden, da zusätzliche Bedingungen an die Anzahl der Ecken und deren Lage existieren.

Die Probleme, die sich aus der Verwendung spezialisierender Vererbung ergeben sind seit langem bekannt. Das Kriterium für die Ersetzbarkeit von Klassen nach LISKOV [3] etwa beschreibt, wann eine Unterklasse anstelle einer Oberklasse verwendet werden kann. Arbeiten verfeinern diese Betrachtungsweise. Der zentrale Punkt ist, daß aus der im Modell vorhandenen *„is-a“*-Beziehung keine Verhaltensgleichheit abgeleitet werden kann.

In heute gängigen Programmiersprachen existiert jedoch kein Mechanismus, der es gestattet, diese Ausnahmesituationen geeignet zu spezifizieren und möglichst bei der Übersetzung prüfen zu lassen. Im nächsten Abschnitt werden deshalb Varianten der Umsetzung einer solchen Beziehung im Programm beschrieben.

Vom Modell zum Code

In diesem Abschnitt betrachten wir die Umsetzung eines sehr kleinen Ausschnitts aus der zuvor beschriebenen Hierarchie der Polygone, nämlich die Klassen *Polygon* und *Viereck*. Ein Viereck ist ein spezielles Polygon und als solches im Entwurf eine spezielle Unterklasse innerhalb der Klassenhierarchie. Ein Polygon besitze weiter eine Methode zum Einfügen eines neuen Eckpunkts. Diese Methode würde jedoch die unveränderliche Eigenschaft eines Vierecks zerstören. Es stellt sich also die Frage, wie der genannte Entwurf geeignet umgesetzt werden kann. Es stehen verschiedene Möglichkeiten zur Verfügung:

Die Semantik der Methode zum Einfügen einer neuen Ecke könnte z.B. dahingehend geändert werden, daß sie das Einfügen nur unter Vorbehalt zusichert. Aufrufer dieser Methode müssen dann sicherstellen, daß das von ihnen gewünschte Ergebnis vorliegt. Diese Art der Modellierung findet sich etwa beim *JAVA COLLECTION FRAMEWORK* der *JAVA 2 Plattform*.

Eine andere Implementierungsvariante besteht darin, daß die Eigenschaft *viereckig* als Zustand eines Polygons angesehen wird. Auf diese Art kann ein Viereck wieder zu einem Polygon werden. Diese Art der Modellierung wird von verschiedenen Autoren vorgeschlagen, hat jedoch unter Sanierungsgesichtspunkten eine zentrale Schwäche: Alle im Entwurf noch getrennt auftretenden speziellen Polygone sind jetzt in eine einzelne Klasse integriert. Bei jeder Methode muß auf evtl. vorhandene spezielle Eigenschaften Rücksicht genommen werden. Dies führt dazu, daß jede Methode sogenannten *dispatch-Code* enthält, der zur Nachbildung von Polymorphie notwendig wird. Abbildung 3 zeigt einen kleinen Teil der Implementierung einer Methode der zu Anfang eingeführten Mitarbeiterhierarchie nach dieser Vorgehensweise.

```
Mitarbeiter::erledigeAuftrag(Auftrag X) {
  if (Stellung == Innendienstler) {
    switch (Taetigkeit) {
    case Administrator :
      ...
    case Sachbearbeiter:
      if (hatFachgebiet(X.gebiet)) {
        ...
      }
    }
  } else {
    ... // Fehlermeldung?
  }
}
```

Abbildung 3: Eine typische Methode?

Es gibt noch eine Reihe anderer Implementierungsvarianten, auf die hier jedoch nicht weiter eingegangen werden soll. Wesentlich ist nur, daß sich Entwickler aus verschiedenen, oft nicht dokumentierten Gründen für eine Implementierungsalternative entscheiden. Der Grund für die Verwendung spezialisierter Zustände könnte etwa eine notwendige dynamische Klassifikation oder aber die beschriebene Implementierung einer Spezialisierungsbeziehung sein.

Erweiterungen

Bereits bei der initialen Implementierung einer Hierarchie entstehen die ersten Probleme für eine spätere Rekonstruktion des zugrundeliegenden Modells. Schlimmer wird es bei der Erweiterung eines Systems.

Bedingt etwa durch einen evolutionären Entwicklungsansatz, durch die Priorisierung der Anforderungen, durch Änderungen von Kundenwünschen oder sonstige neue Anforderungen müssen Klassenstrukturen erweitert werden. Dabei ergeben sich im wesentlichen zwei Probleme:

1. Die Erweiterungen sind oftmals nicht vollständig: Wurde zur Implementierung die zustandsbasierte Variante gewählt, dann muß bei der Erweiterung sichergestellt werden, daß alle Methoden geeignet überarbeitet werden.
2. Es kann zur Vermischung von Implementierungsvarianten kommen: Eine statische Klassenstruktur kann mit einer zustandsbasierten Erweiterung versehen werden, oder umgekehrt. Der Code ist nicht mehr homogen.

Während die erste Problematik oftmals Ursache für Fehler und damit unter Umständen Auslöser für eine Sanierung ist, erschwert der zweite Punkt vor allem die Verständlichkeit oder Lesbarkeit eines Programms.

Vom Code zum Konzept

Abbildung 3 zeigt einen Ausschnitt aus einer möglichen Implementierung der initialen Hierarchie aus Abbildung 1. Offensichtlich wurde eine zustandsbasierte Implementierung gewählt. Dabei wird aus dem gezeigten Ausschnitt nicht klar, ob eine dynamische Klassifikation notwendig ist.

Bereits bei dieser konsequenten Umsetzung eines Modells werden die Schwierigkeiten für die Sanierung klar. Glücklicherweise läßt sich ein entsprechendes Implementierungsmuster werkzeuggestützt vergleichsweise einfach auffinden. Verschiede Analysen können dann ermitteln, wie die einzelnen Variablen verwendet werden und damit Hinweise auf die Art der Klassifikation geben.

Zusammenfassung

Bedingt durch die verringerte Ausdrucksstärke der Implementierungssprache gegenüber der Modellierungssprache weichen objektorientierte Programme oftmals von ihren Modellen ab. Bei Vererbungsbeziehungen etwa ist aus der Implementierung in der Regel der Zweck der Vererbung nicht mehr oder nur schwach zu erkennen. Zudem werden Hierarchien spezialisierter Klassen oftmals als eine einzelne Klasse mit spezialisierten Zuständen implementiert. Umgekehrt kann damit von der Implementierung nur noch schwer auf das zugrundeliegende Modell geschlossen werden.

Als Konsequenz der vorgestellten Problematik ergeben sich folgende Aufgaben:

1. Verbesserung des Ausbildungsstandes der Softwareentwickler. Der Übergang vom Modell zur Implementierung erfordert systematische Vorgehensweisen und gute Kenntnisse der objektorientierten Denk- und Programmierweise. Diese Kenntnisse können im allgemeinen nicht innerhalb eines C++ oder Java Programmierkurses erlernt werden.
2. Verbesserung der Implementierungsmöglichkeiten. Hier sind vor allem die Entwickler sogenannter *Round-trip-engineering* Werkzeuge, aber auch die Entwickler neuer Programmiersprachen gefordert. Insbesondere müssen Möglichkeiten zur Konservierung von Modellierungswissen geschaffen werden, so daß bei späteren Überarbeitungen bzw. Sanierungen auf das vormals vorhandene Wissen zurückgegriffen werden kann.
3. Verbesserung der Techniken zur Rekonstruktion von Systemmodellen. Diese Arbeiten enthalten z.B. Verfahren zur Erkennung verschiedener Implementierungen von Vererbung, also z.B. die Erkennung dynamischer oder multipler Klassifikation.

Literatur

[1] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Publishing Company, Reading, Mass., 1991.

[2] Arne Frick, Gerhard Goos, Rainer Neumann und Wolf Zimmermann. *Designing Hierarchies of Robust Classes.* Erscheint in *Software Practice and Experience.* 1999.

[3] Barbara H. Liskov an J. M. Wing. *A new definition of the subtype relation.* In O. Nierstrasz, editor, *Proceedings of the ECOOP'93*, LNCS 707, pg. 118-141, Springer-Verlag, 1993.

[4] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, 2 edition, January 1997.

[5] Bernd Oestereich, *Objektorientierte Softwareentwicklung – Analyse und Design mit der Unified Modeling Language*, 4. Auflage, Oldenburg, 1998.