informatica didactica 8 (2010) - http://www.informatica-didactica.de

Systematisches Programmieren in der Anfängerausbildung

Jürgen Börstler
Department of Computing Science
Umeå University, Schweden

Michael Sperber
DeinProgramm
Tübingen

31. August 2010

Zusammenfassung

Die didaktische Behandlung der Konstruktion von Programmen ist von zentraler Bedeutung in der Anfängerausbildung im Programmieren: Die meisten Anfänger brauchen Anleitung, um Programmierprobleme erfolgreich eigenständig lösen zu können. Dieses Papier beschreibt zwei solcher Ansätze entlang eines Beispielproblems:

1. objektorientierte Analyse und objektorientiertes Design unter Verwendung von *linguistischer Analyse*, *CRC-Karten* und *Szenario-Rollenspielen*,
2. die *design recipes* (*Konstruktionsanleitungen*) des TeachScheme!-Projects.

Abstract

The didactic treatment of program construction is of central importance when teaching beginners how to program: Most beginners need explicit instruction to solve programming problems successfully on their own. This paper describes two approaches to such explicit instruction, using a common example problem:

1. object-oriented analysis and object-oriented design using *linguistic analysis*, *CRC cards* and *scenario roleplay*
2. the *design recipes* of the TeachScheme! project

1 Einleitung

Im April 2009 trafen sich universitäre Lehrkräfte, die in der Anfängerausbildung tätig sind, zum Workshop 09153 „The Intro Programming Course“ auf Schloss Dagstuhl, um (unter anderem) über effektive didaktische Ansätze zur Programmierausbildung zu sprechen. Das Hauptziel des ersten Programmierkurses ist typischerweise, die Studierenden zu befähigen, eigenständig Programme zu schreiben. Dabei reicht es nicht aus, die Sprachelemente und Features einer Programmiersprache zu beschreiben: Die meisten Studierenden können daraus nicht ohne Hilfe herleiten, wie sie zu Problemen Programme entwickeln, die diese lösen. Die Schwierigkeiten liegen nicht darin, die grundlegenden Sprachelemente und Features einer Programmiersprache zu verstehen, sondern einzelne Sprachelemente zu einem funktionierendem Programm zusammen zu setzen [DB86, SS88]. Wie erwarten Lehrkräfte also, dass Studierende aus einer Programmieraufgabe ein Programm machen, das diese Aufgabe löst?

Dieses Papier beleuchtet den Prozess vom Lesen der Aufgabenstellung bis hin zum fertigen Programm für eine beispielhafte Programmieraufgabe (entworfen von Matthias Felleisen) für zwei Ansätze:

- objektorientierte Analyse unter Verwendung von *linguistischer Analyse*, *CRC-Karten* sowie *Szenario-Rollenspielen* [Bör04]
- *design recipes* bzw. *Konstruktionsanleitungen* [FFFK01, KS07]

Die Aufgabe wurde in einer exemplarischen Sitzung im Rahmen des Workshops jeweils mit beiden Ansätzen bearbeitet. Primäres Ziel dieses Papiers ist es, die dabei angewendeten Vorgehensweisen, sowie die Einschränkungen und Stärken der beiden verwendeten Techniken nachvollziehbar darzustellen. (Dementsprechend ist die Darstellung gegenüber dem tatsächlichen Workshop-Experiment idealisiert.) Gleichzeitig beleuchtet es den Einsatz einer speziellen Lehrsprache beim Einsatz der Konstruktionsanleitungen. Das Papier ist allerdings keine ausführliche Einführung in objektorientierte Analyse, objektorientiertes Design oder Konstruktionsanleitungen – diese werden nur soweit eingeführt, wie für die Beispielaufgabe benötigt.

Überblick Abschnitt 2 stellt zunächst die Aufgabenstellung vor. Darauf beschäftigt sich Abschnitt 3 mit der grundsätzlichen Frage, ob derartige Aufgabenstellungen in der Ausbildung durch konkrete Programmierung gelöst werden sollten, oder ob der Modellierungsprozess von der Programmierung getrennt werden sollte. Abschnitt 4 ist ein Überblick über die OOA/OOD-Techniken, die in diesem Papier demonstriert werden. Dementsprechend stellt Abschnitt 5 die Konstruktionsanleitungen vor. Die nächsten beiden Abschnitte 6 und 7 demonstrieren dann die Anwendung von OOA/OOD-Techniken bzw. der Konstruktionsanleitungen auf die Beispielaufgabe. Abschnitt 8 vergleicht die beiden entstandenen Lösungen auf technischer Ebene. Abschnitt 9 zieht einige Schlussfolgerungen aus dem Vergleich der beiden Lösungen für die Lehre. Abschnitt 10 stellt die Schritte der beiden Ansätze gegenüber und beleuchtet Kombinationsmöglichkeiten. Abschnitt 11 fasst kurz zusammen.

2 Beispielaufgabe

Dieser Abschnitt präsentiert und erläutert die Beispielaufgabe, die zur Demonstration der beiden Lösungsansätze dient. Der Aufgabentext, wie er Studierenden präsentiert wird, ist wie folgt:

> Eine geometrische Figur in der zweidimensionalen Ebene ist entweder
>
> - ein *Quadrat* parallel zu den Koordinatenachsen,
> - ein *Kreis*
> - oder eine *Überlagerung* zweier geometrischer Figuren. (Die Figuren werden übereinandergelegt; die Flächen der beiden Figuren werden also vereinigt.)
>
> Programmieren Sie geometrische Figuren! Schreiben Sie ein Programm, in dem es möglich ist, geometrische Figuren anzulegen und zu überprüfen, ob ein gegebener Punkt innerhalb der Fläche einer gegebenen geometrischen Figur liegt.[1]

Diese Aufgabenstellung ist einfach genug, um beide Lösungsansätze zu demonstrieren. Andererseits erfordert sie die Anwendung mehrerer Problemlösungstechniken –

[1] Die Aufgabe könnte nachträglich um eine weitere Sorte Figuren erweitert werden: „... oder einen *Schnitt* zweiter geometrischer Figuren. (Die Fläche einer solchen Figur ist die Schnittfläche der beiden Figuren.)“ Diese Ergänzung sollte genau wie die Überlagerungen behandelt werden können.

primär bei der Datenmodellierung und der Anwendung bereichsspezifischen Wissens – und zeigt typische Schwierigkeiten beim Programmieren auf. Die Aufgabe erlaubt sowohl Datenmodelle, die mit algebraischen Datentypen formuliert sind, als auch solche mit objektorientierten Klassenhierarchien.

Die Aufgabenstellung ist bereits möglichst eindeutig formuliert und identifiziert die verwendeten Entitäten klar. Sie eignet sich also nicht zur Demonstration von Techniken zur Extraktion von Modellen aus mehrdeutigen oder unklar formulierten Aufgabenstellungen.

Des weiteren fordert die Aufgabenstellung nur die Beantwortung einer Frage über Daten, nicht aber die Durchführung eines zeitgebundenen Prozesses mit veränderlichem Zustand.

Lehrkräfte, die Wert auf konkreten Anwendungsbezug legen, können darauf hinweisen, dass die Lösung dieser Aufgabe als Teil eines „geometric region server“ dienen kann [HJ94].

3 Modellierung vs. Programmierung

Eine Reihe multinationaler Studien hat gezeigt, dass die traditionelle Anfängerausbildung in der Programmierung nicht zum gewünschten Erfolg führt [LSS+04, MWA+01]. Sogar Studierende im letzten Studienabschnitt haben noch große Schwierigkeiten, selbst einfachste Softwaresysteme zu entwerfen [EMM+06].

Einige neuere Arbeiten argumentieren, dass die Details des Programmierens – insbesondere in professionellen Programmiersprachen – Lehrende und Lernende oft vor große Schwierigkeiten stellen, und schlagen deshalb vor, mit der Modellierung von Problemen und Lösungen zu beginnen [Bri04, Die07, HH07]: Die Studierenden konstruieren zunächst überwiegend Klassen- oder Objektmodelle; die konkrete Programmierung wird erst später eingeführt und/oder mit Werkzeugen automatisiert.

Dieser *models-first*-Ansatz erleichtert den Einstieg in die allgemeine Thematik der Software-Entwicklung, verschiebt aber das eigentliche Problem der Konstruktion des konkretes Programms nur nach hinten. Die Programmkonstruktion kann von einer Vorbereitung durch ein Modell zwar profitieren, wird durch diese aber nicht ersetzt: Für die Studierenden bleiben zahlreiche Fragen unbeantwortet, die bei der Programmierung entstehen, wovon die Beispielaufgabe einige aufzeigt.

Models first bezieht seine Stärke aus der Abstraktion. Die Semantik der Modelle ist dabei jedoch nicht eindeutig spezifiziert. In der Praxis entstehen daher beim Modellieren oft Fehler, die erst der Programmierprozess aufdeckt. Des weiteren gibt es keine eindeutigen „Regeln“, um die Modelle in konkreten Programmcode umzusetzen. Außerdem werden ein wichtiger Motivationsfaktor – das Erstellen eines lauffähigen Programmes – sowie die damit zusammenhängenden Übungseffekte ebenfalls nach hinten verschoben.

Andererseits ist es durchaus möglich, mit geeigneten Lehrsprachen, Entwicklungsumgebungen und Konstruktionsanleitungen, Anfänger dazu zu befähigen, von Anfang an selbständig Programme zu schreiben. Aus diesen Gründen beschreibt dieses Papier bei beiden Ansätzen den gesamten Entwicklungsprozess. Dementsprechend sind der konkrete Programmcode und die Schritte, die zu seiner Entstehung führen, von entscheidender Bedeutung: Ohne konsequente Betrachtung des gesamten Prozesses bleiben bei der Ausbildung der Studierenden sonst Lücken, die diese nicht immer aus eigener Kraft füllen können.

4 OOA/OOD-Ansatz

In diesem Ansatz wird vor der eigentlichen Programmierung eine informelle Analyse- und Entwurfs-Phase durchgeführt. Diese dient dazu, die Aufgabenstellung für die spätere Lösung aufzubereiten, einen Überblick über die später entstehenden Klassen und deren Verantwortlichkeiten zu schaffen und schließlich zu präzisieren, welche Methoden von den Klassen später implementiert werden müssen. Erst dann erfolgt die eigentliche Programmierung.

Dazu wird der Text der Problembeschreibung analysiert, um potentielle Kandidaten für die Klassen einer Lösung zu finden (*linguistische Analyse*). Das Ergebnis dieser Analyse wird mit Hilfe von *CRC-Karten* dokumentiert. Mit dem so erstellten informellen Modell einer Lösung werden verschieden Anwendungs-Szenarien durchgespielt, um das Modell zu validieren (*Rollenspiele*).

4.1 Linguistische Analyse

Ein linguistischer Ansatz zur objektorientierten Analyse wurde bereits 1982 von Booch vorgestellt [Boo82]. Die grundlegende Idee ist es, nacheinander die Substantive, Adjektive und Verben und Adverbien in einem Text zu identifizieren. Die Substantive werden als potentielle Kandidaten für Klassen behandelt, die Adjektive als deren Attribute und die Verben und Adverbien als deren Methoden. Diese einfache „Methode“ erlaubt aber keine eindeutige Zuordnung von Attributen und Methode zu Klassen. Verfeinerte Ansätze berücksichtigen spezifische „Schlüsselworte“ oder Phrasen wie „hat“ oder „ist ein“ und/oder die grammatikalische Struktur der einzelnen Sätze.

Kristen [Kri94] beschreibt (unter anderem) konkrete Regeln, um Sätze mit der allgemeinen Struktur

{Subjekt} {Prädikat} {Direktes Objekt} [{Präposition} {Indirektes Objekt}]

wie folgt in Informationsmodelle zu verwandeln:

- Das Subjekt führt eine Aktion aus (kontrolliert diese).
- Das Prädikat beinhaltet oder beschreibt die Aktion.
- Das direkte Objekt ist das Ziel dieser Aktion.
- Die Aktion kann zu einer Zustandsänderung im direkten Objekt führen.
- Das indirekte Objekt hilft dem direkten Objekt diese Aktion durchzuführen.
- Die Präposition ist ein Indikator für die Art der Zusammenarbeit zwischen Subjekt und indirektem Objekt.

Darüber hinaus beschreibt Kristen weitere Regeln, um Klassenbeziehungen wie „hat“ oder „ist ein“ genauer zu bestimmen.

Klassendiagramme können auch mit Hilfe komplexer Spracherkennungssysteme halbautomatisch erstellt werden [JML00, OLR01]. In ihrer einfachsten Form sind solche Ansätze auch in einigen UML-Werkzeugen vorhanden.[2] In Ansätzen mit CRC-Karten beschränkt man sich jedoch üblicherweise auf die Identifikation von Substantiven als Grundlage für ein initiales Klassenmodell.

[2] Visual Paradigm for UML (`http://www.visual-paradigm.com/product/vpuml/`) enthält z.B. ein einfaches Tool *Textual analysis*.

4.2 CRC-Karten

CRC-Karten [BC89, BS97, Bör04, Bör05, WBM03] sind eine informelle Technik, um objektorientierte Modelle zu erstellen und zu testen. Eine CRC-Karte ist in der Regel eine (reale) Karteikarte mit drei Angaben auf der Vorderseite (siehe Abb. 1):

- Klasse (*Class*)
- Verantwortlichkeiten (*Responsibilities*)
- Kollaborateure (*Collaborators*).

Die Rückseite der Karte kann für eine Beschreibung der generellen Rolle der Klasse und/oder weitere Kommentare genutzt werden.

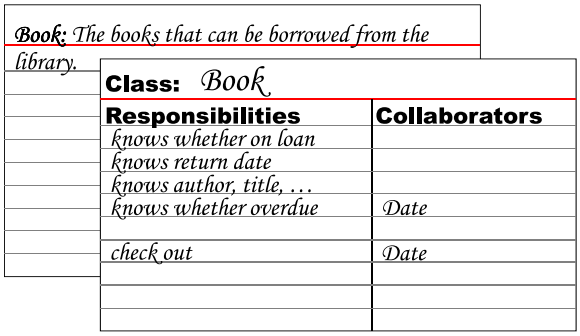

Abbildung 1: CRC-Karte für eine Buch-Klasse in einem Bibliothekssystem.

Bei der Arbeit mit CRC-Karten wird also jeweils eine Karteikarte für jede Klasse mit diesen Angaben angefertigt. Diese dienen dann als Vorlage für die tatsächlichen Klassen.

Die *Verantwortlichkeiten* beschreiben die „Dienste“, die die Objekte dieser Klasse zur Verfügung stellen.

Um diesen Schritt zu erleichtern, sollen sich die Studierenden zwei Fragen stellen:

1. Was müssen die Objekte der Klasse *wissen*, um ihrer Rolle gerecht zu werden („knows“-Verantwortung)?
2. Was müssen die Objekte der Klasse *tun*, um ihrer Rolle gerecht zu werden („does“-Verantwortung)?

Die Dienste können Auskunfts- oder Aktivitätscharakter haben. Ein Buch kann z.B. über seinen Titel und seinen Ausleihstatus Auskunft geben (`knows title`, `knows whether overdue`) und das Ausleihen übernehmen (`check out`). Manche Dienste können nur in Zusammenarbeit mit anderen Objekten (anderer Klassen) ausgeführt werden. Zur Überprüfung des Ausleihstatus muss z.B. das Rückgabedatum (`knows return date`) mit dem aktuellen Datum verglichen werden. Da diese Information einem Buch nicht bekannt ist, muss es diese Information von einem anderen Objekt (Kollaborateur) bekommen. Die Klasse(n) der *Kollaborateure* werden in der entsprechenden Zeile in der Spalte „Collaborators“ eingetragen.

Die „knows“-Verantwortlichkeiten werden in einfachen Fällen später als Attribute implementiert (sonst als Methoden), die „does“-Verantwortlichkeiten als Methoden. Alle Verantwortlichkeiten (Aufgaben) werden in der linken „Responsibilities“-Spalte eingetragen. Sind die Aufgaben komplex, können die Objekte auf die Hilfe anderer Objekte (anderer Klassen) angewiesen sein, an die sie Teilaufgaben delegieren. Die Klassen dieser Objekte werden in der „Collaborator“-Spalte in die entsprechende Zeile eingetragen.

4.3 Rollenspiele

Um das erstellte Modell zu testen, werden Rollenspiele mit Beispiel-Szenarien durchgeführt. Dieser Ansatz eignet sich insbesondere für Gruppenarbeit. Er dient zur Klarstellung der Aufgaben der einzelnen Klassen. Ggf. führen die Rollenspiele zu neuen Erkenntnissen über die Aufgabenstellung oder ihre Lösung; das Modell wird dann entsprechend angepasst.

In der Literatur dienen die CRC-Karten im Rollenspiel als Surrogate für konkrete Objekte [BC89, BS97]. Wir halten dieses Vorgehen in der Anfängerausbildung für unangebracht, da es mögliche Missverständnisse untermauert [Bör05]: Das Verständnis für die wichtigen Unterschiede zwischen Klassen und ihren Instanzen wird dadurch entscheidend gestört, und zwar umso mehr, wenn die Beispiel-Szenarien mehrere Instanzen derselben Klasse involvieren. Gerade dies ist aber wünschenswert, um dieses Verständnis zu fördern.

Zur Unterstützung und Dokumentation der Rollenspiele schlagen wir sogenannte *Rollenspieldiagramme* vor [Bör04]. Rollenspieldiagramme entsprechen in etwa vereinfachten UML-Kommunikationsdiagrammen, bei denen konkrete Attributwerte annotiert werden (wie in UML-Objektdiagrammen). Jedes Objekt wird durch eine *Objekt-Karte* (z.B. einen PostIt-Zettel) repräsentiert. Auf der Objekt-Karte werden Name und/oder Typ des Objektes notiert und die für das aktuelle Szenario relevanten Attributwerte vermerkt (siehe Abb. 2). Aktuelle Beziehungen zwischen Objekten werden durch Verbindungen zwischen den entsprechenden Objekt-Karten verdeutlicht. Entlang dieser Verbindungen können Nachrichten versendet werden. Im oberen Diagramm in Abb. 2 sendet z.B. das Objekt `:Librarian` die Nachricht `on loan?` an das Objekt `aBook` und erhält daraufhin `no` als „Antwort“. Das untere Diagramm in Abb. 2 zeigt, dass `:Librarian` keine Nachricht an `aDate` senden kann, da keine Verbindung zwischen diesen beiden Objekt-Karten besteht.

Ein Rollenspieldiagramm wird dann während des Rollenspiels entsprechend dem laufenden Szenario sukzessiv aufgebaut. Die Objekt-Karten und ihre Beziehungen werden dabei entsprechend geändert. Werden Objekte erzeugt oder gelöscht, werden in dem Diagramm entsprechende neue Objekt-Karten hinzugefügt oder gelöscht.

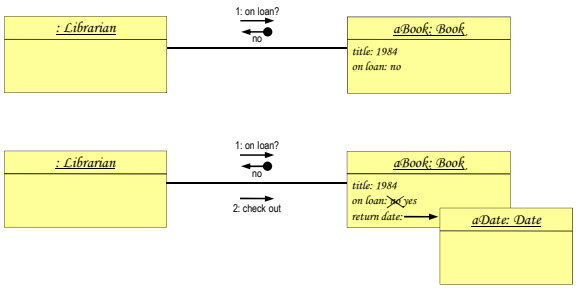

Abbildung 2: Dokumentationsschritte aus dem Beispielszenario „das Buch ‘1984’ wird ausgeliehen“ (leicht vereinfacht).

Rollenspiele sind informell; sie bauen auf Gruppendynamik auf und fördern diese. Da die Objekte durch reale Gegenstände – die Notizzettel – abgebildet werden, unterstützt diese Methode das Verständnis von Objektidentität und Methodenzuordnung („object thinking“). Der Fokus liegt dabei nicht auf der Anwendung konkreter fester Regeln um die Methodenzuordnung festzulegen, sondern auf der Diskussion in der Gruppe, die die Effekte unterschiedlicher Design-Entscheidungen beim Rollenspiel konkret erlebt.

Das Rollenspiel gibt wenig Unterstützung beim konkreten Implementieren einzelner Methoden. Die Rollenspieldiagramme geben aber Auskunft über die Platzierung

konkreter Methoden und welche Parameter sie benötigen bzw. welche Resultate sie liefern. Aus dem Beispiel in Abb. 2 kann man z.B. ableiten, dass die Nachricht `on loan` einen Wahrheitswert als Resultat liefert und den Zustand von `aBook` nicht ändert. Die Nachricht `check out` hingegen hat recht komplexe Effekte auf das Objekt `aBook`; es ändert seinen Ausleihstatus `on loan` und instanziert ein lokales Objekt `aDate`, welches das Rückgabedatum `return date` repräsentiert.

5 Konstruktionsanleitungen

Die Konstruktionsanleitungen, entwickelt (unter dem Namen *design recipes*) im Lehrbuch *How to Design Programs* (kurz *HtDP*) [FFFK02, FFFK01] [KS07]) sind eine Sammlung von Techniken zur Konstruktion von Programmen. Zentral ist dabei *systematisches Vorgehen*: Der Ansatz betont, dass bei jedem Problem eine feste Folge von Schritten durchlaufen wird, zu denen auch die Konstruktion von Beispielen und Testfällen gehören. Er basiert damit auf konkreten Regeln, deren korrekte Anwendung zur Problemlösung führt. Die Konstruktionsanleitungen operieren im Vergleich mit dem OOA/OOD-Ansatz auf einer feineren Granularität: Jeder Schritt produziert ein konkretes Element des Programms, und die Programmkonstruktion wird „bis ganz nach unten“ gesteuert.

Der Programmierer führt bei Anwendung der Konstruktionsanleitungen zunächst eine *Datenanalyse* des Problems durch. Dann wird das zu lösende Problem als *Signatur* einer zu schreibenden Prozedur formuliert, also einer Spezifikation, was für Datensorten die Prozedur als Eingaben akzeptiert und aus welcher Datensorte die Ausgaben kommen. Die Ergebnisse der Datenanalyse steuern direkt die Konstruktion einer *Schablone* für die zu schreibende Prozedur, die schließlich nur noch um problemspezifisches Wissen zur fertigen Lösung ergänzt wird.

Die Konstruktionsanleitungen sind in eine feste Schrittfolge bei der Konstruktion von Prozeduren eingepasst. Hier die Beschreibung der Schrittfolge als Anleitung für Anfänger:

Kurzbeschreibung Schreiben Sie eine einzeilige Kurzbeschreibung.

Datenanalyse Führen Sie eine Analyse der beteiligten Daten durch. Stellen Sie dabei fest, zu welcher Sorte die Daten gehören, ob Daten mit Fallunterscheidung vorliegen und ob zusammengesetzte oder gemischte Daten vorliegen.

Signatur Wählen Sie einen Namen und schreiben Sie einer Signatur für die Prozedur.

Testfälle Schreiben Sie einige Testfälle.

Gerüst Leiten Sie direkt aus der Signatur das Gerüst der Prozedur her.

Schablone Leiten Sie aus der Signatur und der Datenanalyse mit Hilfe der Konstruktionsanleitungen eine Schablone her.

Rumpf Vervollständigen Sie den Rumpf der Prozedur.

Test Vergewissern Sie sich, dass die Tests erfolgreich laufen.

Diese feste Schrittfolge ist einerseits für die Studierenden hilfreich, da sie die Lösung der Aufgabe erleichtert. Außerdem vereinfacht sie für die Tutoren die Bewertung, Problemanalyse und Hilfe. Für die Konstruktion von Schablonen gibt es differenzierte Anleitungen, die in Abschnitt 7 beschrieben sind.

Die Konstruktionsanleitungen werden in Anfängerkursen unter Benutzung spezieller Programmiersprachen und einer speziellen Programmierumgebung eingeführt, die auf die Bedürfnisse von Programmieranfängern zugeschnitten sind. Dies trifft in besonderem Maße auf die speziellen Sprachebenen für *How to Design Programs*

[FFFK01] und *Die Macht der Abstraktion* [KS07] zu, die bei DrRacket [FCF+02] (das früher *DrScheme* hieß) mitgeliefert sind und auf der Programmiersprache Scheme basieren. Die Benutzung spezieller Anfänger-Sprachen sowie einer besonderen Anfänger-Programmierumgebung hat noch weitere Vorteile – übersichtlichere Bedienung, bessere Fehlermeldungen etc. Diese Aspekte werden in Abschnitt 7.8 kurz diskutiert.

6 Lösung mit OOA/OOD

Dieser Abschnitt erläutert die möglichen Schritte einer Lösung der Beispielaufgabe mit den Techniken aus Abschnitt 4, die der objektorientierten Analyse bzw. dem objektorientierten Design zugeordnet werden. Zuerst steht eine *linguistische Analyse*, gefolgt von der Herstellung von *CRC-Karten* und schließlich einem *Rollenspiel*. Es folgt eine Skizze, wie die Ergebnisse dieser Schritte zu einer *Implementierung* verdichtet werden können.

6.1 Linguistische Analyse

Für die linguistische Analyse werden zunächst Kandidaten für die Klassen in einer Lösung identifiziert. Dazu werden in der Aufgabenstellung alle Substantive unterstrichen:

> Eine geometrische Figur in der zweidimensionalen Ebene ist entweder
>
> - ein *Quadrat* parallel zu den Koordinatenachsen,
> - ein *Kreis*
> - oder eine *Überlagerung* zweier geometrischer Figuren. (Die Flächen der beiden Figuren werden also vereinigt.)
>
> Programmieren Sie geometrische Figuren! Schreiben Sie ein Programm, in dem es möglich ist, geometrische Figuren anzulegen und zu überprüfen, ob ein gegebener Punkt in einer gegebenen geometrischen Figur enthalten ist.

Für jedes Substantiv wird diskutiert, inwieweit es eine für die Lösung relevante Klasse ist. „Programm“ z.B. bezeichnet die Lösung allgemein und ist daher keine für die Lösung relevante Klasse. In dieser Diskussion werden auch Synonyme, unterschiedliche Schreibweisen und begriffliche Varianten diskutiert. Es bleiben also:

- geometrische Figur
- zweidimensionale Ebene
- Quadrat
- Kreis
- Überlagerung
- Punkt

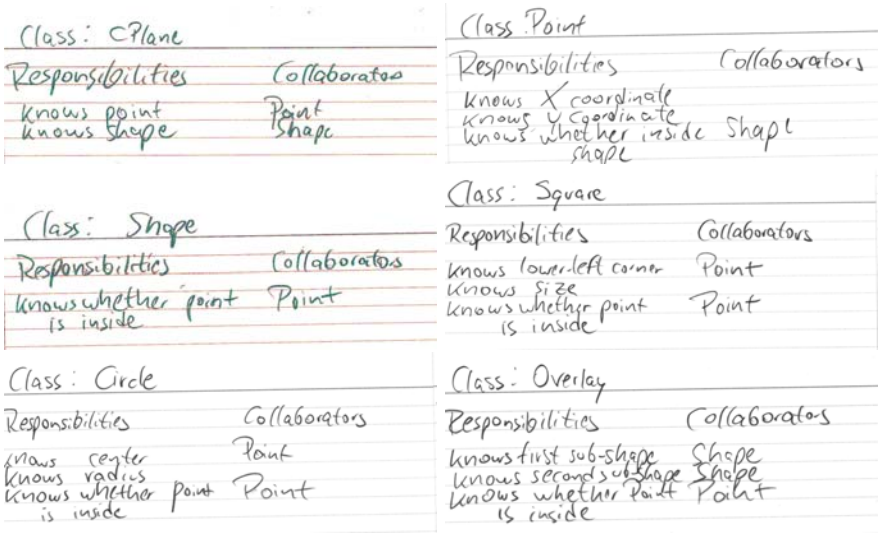

Abbildung 3: CRC-Karten für geometrische Figuren.

6.2 CRC-Karten

Für jeden verbleibenden Begriff aus der linguistischen Analyse wird eine Karteikarte – entsprechend einer Klasse – angefertigt. Dabei wird Wert darauf gelegt, dass jede Klasse eine klar definierte Rolle in der Lösung hat.

Abbildung 3 zeigt einen Satz CRC-Karten für die geometrischen Figuren, entsprechend den Substantiven aus der linguistischen Analyse. Die Karten zeigen, dass die zentrale Aufgabenstellung, die „enthalten“-Frage zu beantworten, nicht eindeutig einer einzelnen Klasse zugeordnet ist. Diese Aufgabe kann z.B. eine geometrische Figur übernehmen, wobei der Punkt als „Collaborator“ (z.B. Parameter) gebraucht wird, oder sie wird dem Punkt zugeordnet, welcher dann wiederum eine geometrische Figur als „Collaborator“ braucht. Darum ist in Figur 3 diese Aufgabe sowohl auf der `Point`-Klasse als auch auf den verschiedenen Klassen für geometrische Figuren vermerkt.

Dieses initiale Modell erklärt noch nicht die Beziehungen zwischen den Klassen. Eine weiterführende Analyse der Aufgabenstellung kann aber Hinweise auf diese Beziehungen geben. Formulierungen wie „S ist ein T“ oder „A hat ein B“ können z.B. als Hinweise auf eine Subtypbeziehung zwischen den Klassen `S` und `T` oder eine Komponentenbeziehung zwischen `A` und `B` interpretiert werden. Die „CRC-Methode“ ist da aber nicht präskriptiv. Eine weiterführende Analyse, wie in Kapitel 4.1 angedeutet, könnte aber zusätzliche Information liefern.

Die CRC-Karten liefern also nur einen initialen Satz Klassen als „Arbeitshypothese“, die in Beispiel-Szenarien getestet und verfeinert werden muss.

6.3 Rollenspiel

Mit Hilfe der CRC-Karten kann nun eine Gruppe von Studierenden ein Rollenspiel durchführen, in dem jedes Mitglied die Rolle eines Objektes übernimmt. Dazu definiert die Gruppe verschiedene Szenarien, die das Programm bewältigen muss. Ein erstes Szenario könnte z.B. wie folgt lauten:

> In der Ebene gibt es einen Kreis und einen Punkt. Prüfe, ob der Punkt im Kreis liegt.

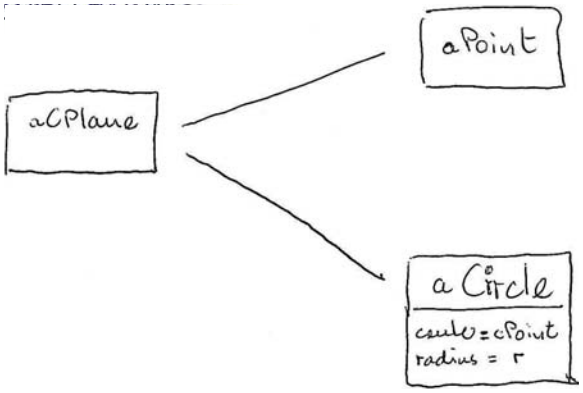

Abbildung 4: Rollenspieldiagramm: Startsituation.

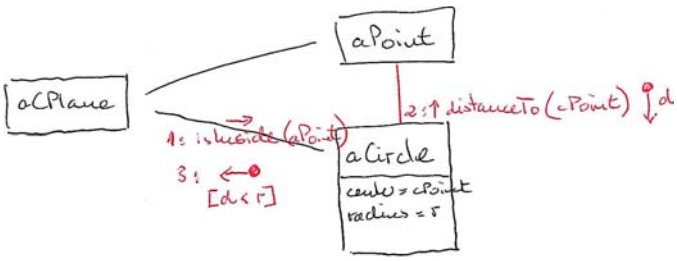

Abbildung 5: Rollenspieldiagramm: Nach Bearbeiten der Anfrage „isInside“.

Ein entsprechendes initiales *Rollenspieldiagramm* würde dann wie in Abbildung 4 aussehen. Die Objekt-Karten für `aCircle` und `aPoint` sind bereits mit `aCPlane` verbunden, da die CRC-Karte für `CPlane` angibt, dass `CPlane` die Objekte in der Ebene „kennt“.

Anfänger werden gehalten, das Startszenario sorgfältig zu konstruieren und insbesondere folgende Fragen zu stellen und zu beantworten:

- Welche Objekte gibt es?
- Welche Beziehungen haben die Objekte zueinander?
- Wie ist der aktuelle Zustand dieser Objekte?

Die Gruppe kann nun verschiedene Alternativen durchspielen. Zum Beispiel kann `aCPlane` entweder `aPoint` „fragen“, ob er in `aCircle` liegt, oder `aCircle`, ob `aPoint` in ihm enthalten ist. Beide Klassen, `Point` und `Circle`, haben die entsprechenden Verantwortungen `knows whether inside`. Um die Aufgabe für einen Kreis zu lösen, ist es nötig, den Abstand zwischen zwei Punkten zu berechnen. Wenn die Gruppe entscheidet, die Aufgabe an `aCircle` zu „delegieren“, fällt der Gruppe auf, dass `aCircle` nicht mit `aPoint` kommunizieren kann; es gibt im Diagramm keine Verbindung zwischen `aCircle` und `aPoint`. Um dieses Problem zu lösen, muss der Nachricht `isInside` ein Parameter mitgegeben werden, der die beiden Objekte miteinander „bekannt macht“ (siehe Nachricht `1:` in Abb. 5). Die neue Beziehung wird im Diagramm notiert.

Als nächsten Schritt kann nun `aCircle` eine Nachricht an einen der (bekannten) Punkte `cPoint` oder `aPoint` nach dem Abstand zu dem jeweils anderen Punkt fragen (und diesen als Parameter angeben, siehe Nachricht `2:` in Abb. 5). Da die CRC-Karte für `Point` keine entsprechende Verantwortung hat, muss diese nachgetragen werden, bevor das Rollenspiel fortgesetzt werden kann.

Als Resultat der Nachricht `distanceTo` wird ein Abstand `d` zurückgegeben. Dieser wird dann wiederum dazu verwendet, um die initiale Nachricht abschließend zu bearbeiten (Nachricht `3:` in Abb. 5).

Wenn eine Gruppe nun ein analoges Szenario mit anderen Figuren durchspielt, kann sie entdecken, dass das „Interface“ zu den Figuren gleich ist. Die Vermutung, dass `Shape` Superklasse ist, wird also bestätigt; `Shape` erhält daher eine abstrakte Methode `isInside`. Im Rahmen des CRC-Karten-Ansatzes ist es also möglich, Richtlinien für die Gestaltung von Klassenhierarchien einzubauen. Dies ist allerdings bisher noch nicht konkret ausformuliert oder erprobt worden.

6.4 Implementierung

Die Schritte bis hier sind im Wesentlichen eine Methode für den objektorientierten Entwurf, beziehen sich also nicht auf eine konkrete Implementierung. Dennoch können die Ergebnisse dieser Schritte – die entstandenen CRC-Karten und die Rollenspieldiagramme – benutzt werden, um Klassengerüste zu schreiben. Auch wenn sie unzureichende Hilfestellung beim „Ausprogrammieren“ von Methoden geben, so können doch einige Fragen geklärt werden.

Die konkreten verwendeten Programmierkonventionen sind unabhängig von den Ergebnissen des objektorientierten Entwurfs. Im folgenden wird der Vorgang am Beispiel von Java-Code illustriert. Die `CPlane`-Klasse hat Felder, die mit jeweils einem Punkt und einer Figur besetzt werden können, plus eine Methode, um die Aufgabenstellung zu lösen.

```
class CPlane {

    private Shape shape;
    private Point point;

    public void setShape(Shape aShape) {
        shape = aShape;
    }

    public void setPoint(Point aPoint) {
        point = aPoint;
    }

    public boolean checkIfPointInsideShape() {
        return shape.isInside(point);
    }
}
```

Bei der `Point`-Klasse gibt es Felder `x` und `y`, die mit Getter-Methoden versehen werden. Die Methode `distanceTo` wird mit einer Formel aus der Formelsammlung programmiert. Eine Methode für die Verantwortung `knows whether inside shape` entfällt, da sich die alternative Methode, die im Rollenspiel getestet wurde (siehe Abschnitt 6.3), als erfolgreich erwiesen hat.

```
class Point {

    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
```

```
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double distanceTo(Point aPoint) {
        return Math.sqrt(sqr(x - aPoint.getX()) + sqr(y - aPoint.getY()));
    }

    private double sqr(double x) {
        return x * x;
    }
}
```

Die Klasse `Shape` ist eine abstrakte Superklasse für alle Figuren und definiert die Typsignatur der gemeinsamen Methode `isInside`.

```
abstract class Shape {
    public abstract boolean isInside(Point aPoint);
}
```

Ein interessantes Detail der Klasse `Circle` ist, dass die Realisierung der Methode `isInside` direkt aus dem in Abschnitt 6.3 beschriebenen Rollenspiel abgeleitet werden kann. Das setzt aber voraus, dass die Gruppe das Szenario auf dem beschriebenen Detailniveau durchführt.

```
class Circle extends Shape {

    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() {
        return center;
    }

    public double getRadius() {
        return radius;
    }

    public boolean isInside(Point aPoint) {
        return aPoint.distanceTo(center) <= radius;
    }
}
```

Bei `Square` ist die Ausprogrammierung ebenfalls mit Hilfe von Wissen aus der Geometrie möglich:

```
class Square extends Shape {
    private Point lowerLeftCorner;
    private double size;

    public Square(Point lowerLeftCorner, double size) {
        this.lowerLeftCorner = lowerLeftCorner;
        this.size = size;
    }

    public Point getLowerLeftCorner() {
        return lowerLeftCorner;
    }

    public double getSize() {
        return size;
    }

    public boolean isInside(Point aPoint) {
        return (aPoint.getX() >= lowerLeftCorner.getX())
            && (aPoint.getX() <= lowerLeftCorner.getX() + size)
            && (aPoint.getY() >= lowerLeftCorner.getY())
            && (aPoint.getY() <= lowerLeftCorner.getY()y + size);
    }
}
```

Bei `Overlay` gibt es wiederum keine besondere Hilfestellung bei der Ausprogrammierung von `isInside` – hier wären rekursive Aufrufe nötig, um den Selbstbezügen bei `topShape` und `botShape` (die auf den CRC-Karten nicht unmittelbar zu erkennen sind) zu folgen.

```
class Overlay extends Shape {
    Shape topShape;
    Shape botShape;

    public Overlay(Shape topShape, BotShape botShape) {
        this.topShape = topShape;
        this.botShape = botShape;
    }

    public Point getTopShape() {
        return topShape;
    }

    public Point getBotShape() {
        return botShape;
    }

    public boolean isInside(Point aPoint) {
        ...
    }
}
```

6.5 Zusammenfassung

Die objektorientierte Analyse und das objektorientierte Design bieten informelle Hilfestellungen bei der Strukturierung von Problemlösungen „im groben und im großen“:

- Die linguistische Analyse hilft, die Identitäten zu identifizieren, die im späteren Programm modelliert werden müssen.
- Die CRC-Karten helfen, die Ergebnisse der linguistischen Analyse zu benutzen, um festzulegen, welche Klassen und Methoden im späteren Programm existieren müssen sowie Verantwortlichkeiten und damit Methoden den Klassen zuzuordnen.
- Die Rollenspiel-Technik hilft, die Erkenntnisse der CRC-Karten grob zu validieren und ggf. zu modifizieren.
- Die Rollenspieldiagramme helfen dabei festzulegen, welche Teilaufgaben in welchen Methoden gelöst werden müssen.

Diese Techniken bieten – abseits von der Skizzierung der Problemlösung als Satz von Klassen und Methoden – wenig direkte Hilfestellung beim konkreten Programmieren.

6.6 Erfahrungen

CRC-Karten sind ein verbreitetes Hilfsmittel, um Studierenden einen Einblick in die objektorientierte Modellierung zu geben. Ihre erfolgreiche Einführung und Anwendung wird jedoch durch verschiedene Probleme erschwert [Bör05], die im folgenden kurz diskutiert werden. Die Probleme fallen im wesentlichen in drei Bereiche:

1. der Ansatz an sich
2. die Durchführung von Rollenspielen
3. die Dokumentation der Ergebnisse.

Ansatz an sich Trotz der Popularität von CRC-Karten gibt es keine Bücher, die den Ansatz im Detail für die Zielgruppe von Programmieranfängern beschreiben. Viele Lehrbücher für Einführungskurse in die objektorientierte Programmierung enthalten jedoch kurze Einführungen, die allerdings viel zu oberflächlich sind, um eine wirkliche Hilfe bei der konkreten Anwendung zu sein. Das Gleiche gilt für die Fülle von Tutorials, die über das Internet zugänglich sind. Die Einführungen konzentrieren sich im wesentlichen auf die initiale Phase der Modellierung. Das Rollenspiel wird, falls überhaupt, nur angedeutet.

Das Hauptproblem des Originalansatzes ist die Vermischung der Begriffe *Klasse* und *Objekt*. Beck und Cunningham [BC89] betreiben diese Vermischung in ihrem Originalartikel absichtlich. Ihre Zielgruppe sind jedoch erfahrene Programmierer, für die dies kein Problem darstellt. Für Programmieranfänger ist dies doch ein erhebliches Problem, das durch die vorhandene Literatur eher noch verstärkt als gemildert wird. Wir halten es aus diesen Gründen auch für falsch, die CRC-Karten im Rollenspiel als Surrogate für Objekte zu benutzen.

Der CRC-Karten Ansatz propagiert eine vermenschlichte Objektsichtweise („object-as-person metaphor“ [Wes04]). Dies gibt Anfängern ein plastisches Verständnis für viele objektorientierte Konzepte und erleichtert deren Verständnis, wie z.B. das Versenden von Nachrichten. Es kann aber auch dazu führen, dass Studierende Modell und Realität nicht auseinander halten können und den Modellobjekten Eigenschaften zugeschrieben werden, die sie eigentlich nicht haben (siehe z.B. [Bör05].

Rollenspiel Das Rollenspiel kann recht ineffektiv werden, wenn die Szenarien zu allgemein gehalten werden. In unserem Figuren-Beispiel wird dieses Problem noch nicht deutlich. In einem Bibliothekssystem aber wird z.B. beim Szenario „User U leiht das Buch B aus“ oft zu einfach argumentiert: U schickt an B `checkOut(`U`)`. Dabei wird dann oft übersehen, dass sich U und B mglw. gar nicht „kennen“, d.h. sich keine Nachrichten senden können. Der Erkenntnisgewinn aus solchen trivialen Szenarien ist minimal. Des weiteren wird das Rollenspiel unübersichtlich, da der Zustandsraum mit jeder neuen Entscheidung explodiert. Dieser Aspekt wird von vielen Anhängern der Methode übersehen. Schlecht konstruierte Szenarien werden unkontrollierbar, sind schwer zu wiederholen und werden dadurch wertlos, weil Entscheidungen nicht unabhängig voneinander betrachtet werden können.

Für ein „gutes“ Szenario ist es wichtig, einen spezifischen „Startzustand“ festzulegen. Für das Ausleihbeispiel kann das z.B. bedeuten, welche Bücher tatsächlich vorhanden sind, ob User U ein legitimer User ist usw.

Um die in Abschnitt 6.3 aufgeführten Details zu klären, müssen außerdem die „richtigen“ Fragen gestellt werden. Für Anfänger ist das oft schwierig, da sie den Objekten viel mehr „Wissen“ zuschreiben als sie eigentlich haben. Dies führt gelegentlich zu Szenarien, die für die weitere Bearbeitung der Aufgabe nicht hilfreich sind.

Wie bereits oben angedeutet, halten wir es für grundsätzlich falsch, die CRC-Karten im Rollenspiel als Surrogate für Objekte zu benutzen. Die Einführung von Rollenspieldiagrammen hat unserer Erfahrung nach das Rollenspiel erheblich vereinfacht und das Verständnis für grundlegende objektorientierte Konzepte verbessert. Die Anzahl der Studierenden, die über Probleme berichten, hat seitdem erheblich abgenommen [Bör05].

Dokumentation Szenarien oder Rollenspiele werden in der CRC-Karten Literatur oft als UML-Sequenzdiagramme dokumentiert. Nach unseren eigenen Erfahrungen ist es jedoch unpraktisch oder gar unmöglich, ein Rollenspiel „live“ als Sequenzdiagramm zu protokollieren. Wenn Entscheidungen revidiert werden, ist es nicht möglich, Sequenzdiagramme inkrementell anzupassen. Diese müssen oft im wesentlichen neu erstellt werden. Dies stört die Dynamik des Rollenspiels erheblich. Mit der Einführung von Rollenspieldiagrammen hat sich dies erheblich verbessert.

7 Lösung mit Konstruktionsanleitungen

Dieser Abschnitt stellt die Lösung der Beispielaufgabe mit Hilfe von Konstruktionsanleitungen vor. Anders als bei der OOA/OOD-Lösung produziert jeder Schritt der Konstruktionsanleitungen ein Element des späteren fertigen Programms – die Vorgehensweisen lassen sich also direkt Schritt für Schritt gegenüberstellen. Die Schritte folgen der Liste in Abschnitt 5: *Kurzbeschreibung* der zu schreibenden Prozedur, *Datenanalyse*, *Signatur* der zu schreibenden Prozedur, *Testfälle*, *Gerüst*, *Schablone*, *Rumpf* und schließlich *Tests*.

Die Lösung mit Konstruktionsanleitungen benutzt die DrRacket-Sprachebene *Die Macht der Abstraktion – Anfänger*.

Kurzbeschreibung Die Kurzbeschreibung fasst die Aufgabe der Prozedur in einer einzelnen Zeile zusammen. Dieses Format zwingt die Studierenden zu einer knappen Formulierung und verursacht gleichzeitig nur wenig Arbeit, wird also nicht als „lästige Formalie“ aufgefasst. Da die Aufgabe eine „Ja/Nein-Frage“ beantwortet, sollte die Kurzbeschreibung ebenfalls die Form einer Ja/Nein-Frage haben:

```
; ist ein Punkt innerhalb einer Figur?
```

7.1 Datenanalyse

Die Datenanalyse ist der zentrale Teil der Konstruktionsanleitungen. In der Kurzbeschreibung tauchen folgende Größen auf:[3]

1. eine geometrische Figur
2. ein Punkt

Der Begriff der geometrischen Figur kann anhand der Aufgabenstellung noch weiter ausgeführt werden:

Eine geometrische Figur ist eins der folgenden:

- ein Kreis,
- ein Quadrat,
- oder eine Überlagerung.

Diese Formulierung (eine *Datendefinition*) identifiziert geometrische Figuren als *gemischte Daten*: Jede geometrische Figur ist eine der genannten Alternativen. Bei verschiedenen Aufrufen einer Prozedur, die geometrische Figuren als Eingabe akzeptiert, können die verschiedenen Arten geometrischer Figuren gemischt auftreten.

Der Begriff des Kreises könnte folgendermaßen ausformuliert werden:

Ein Kreis hat:

- einen Mittelpunkt
- und einen Radius.

(Andere Formulierungen sind möglich, z.B. Durchmesser statt Radius, der weitere Lösungsweg ist jedoch stets der gleiche.)

Formulierungen der Form „x hat“ oder „x besteht aus“ identifizieren x als *zusammengesetzte Daten*: Ein Objekt besteht aus mehreren Bestandteilen, oder es gibt dazu mehrere Angaben.

Auch Quadrate und Überlagerungen sind zusammengesetzte Daten:

Ein Quadrat hat:

- eine untere linke Ecke
- und eine Kantenlänge.

Eine Überlagerung besteht aus:

- einer geometrischen Figur oben
- und einer geometrischen Figur unten.

(„Oben“ und „unten“ spielen für die Aufgabenstellung, für einen Punkt zwischen „innerhalb“ und „außerhalb“ zu unterscheiden, keine Rolle.)

Die letzte Formulierung identifiziert nicht nur Überlagerungen als zusammengesetzte Daten, sondern zeigt auch die beiden *Selbstbezüge* der Datendefinition für geometrische Figuren auf – die Datendefinition für geometrische Figuren benutzt (mittelbar) wieder den Begriff der geometrischen Figur.

Es bleibt der Begriff des Punkts:

Ein Punkt besteht aus:

[3] In der Aufgabenstellung taucht auch noch die zweidimensionale Ebene auf. Dies ist die „Umgebung“, in der sich der Punkt und die Ebene befinden, sie bedarf also keiner expliziten Modellierung, auch wenn die explizite Modellierung nicht prinzipiell schadet.

- X-Koordinate
- und Y-Koordinate.

Es handelt sich also auch hier um zusammengesetzte Daten.

Die Datendefinitionen werden nun direkt in Code übersetzt. Dazu gibt es spezialisierte Konstruktionsanleitungen. Gemischte Daten werden zu einer Signaturdefinition. Die Konstruktionsanleitung dafür sieht folgendermaßen aus:

> Wenn bei der Datenanalyse gemischte Daten auftauchen, schreiben Sie eine Datendefinition der Form:
>
> ; Ein x ist eins der folgenden:
> ; - $Sorte_1$ (sig_1)
> ; - ...
> ; - $Sorte_n$ (sig_n)
>
> Dabei sind die $Sorte_i$ umgangssprachliche Namen für die möglichen Sorten, die ein Wert aus diesen gemischten Daten annehmen kann. Die sig_i sind die zu den Sorten gehörenden Signaturen. Wählen Sie außerdem einen Namen *sig* für die Verwendung als Signatur.
>
> Aus der Datendefinition entsteht eine Signaturdefinition folgender Form:
>
> (define *sig*
> (signature
> (mixed sig_1
> ...
> sig_1)))

Die Konstruktionsanleitung stellt außerdem noch eine Schablone für den späteren Rumpf; diese wird weiter unten diskutiert.

Die Datendefinition wird also als Kommentar im Programmtext verfeinert und um die Definition einer Signatur ergänzt, die direkt aus der Datendefinition hervorgeht:

```
; Eine geometrische Figur ist eins der folgenden:
; - ein Kreis (circle)
; - ein Quadrat (parallel zu den Achsen) (square)
; - eine Überlagerung zweier geometrischen Figuren (overlay)
(define shape
  (signature
   (mixed circle
          square
          overlay)))
```

Hier wird zum Ausdruck gebracht, dass die `shape`-Signatur (ähnlich einem Summentyp) zu gemischten Daten („`mixed`“) gehört, bei denen es drei Möglichkeiten gibt: `circle`, `square` und `overlay`. Diese Signaturen sind noch zu definieren.

Weiter geht es mit Kreisen: Datendefinitionen für zusammengesetzte Daten werden zu Record-Typ-Definitionen. Auch hier gibt es eine entsprechende Konstruktionsanleitung:

> Wenn bei der Datenanalyse zusammengesetzte Daten vorkommen, stellen Sie zunächst fest, welche Komponenten zu welchen Sorten gehören. Schreiben Sie dann eine Datendefinition, die mit folgenden Worten anfängt:

```
; Ein x besteht aus / hat:
; - Feld_1 (sig_1)
; ...
; - Feld_n (sig_n)
```

Dabei ist x ein umgangssprachlicher Name für die Sorte, die $Feld_i$ sind umgangssprachliche Namen und kurze Beschreibungen der Komponenten und die sig_i die dazugehörigen Signaturen.

Übersetzen Sie die Datendefinition in eine Record-Definition, indem Sie auch Namen für die Record-Signatur *sig*, Konstruktor *constr*, Prädikat *pred*? und die Selektoren $select_i$ wählen:

```
(define-record-procedures sig
  constr pred?
  (select_1 ... select_n))
```

Schreiben Sie außerdem eine Signatur für den Konstruktor der Form:

```
(: constr (sig_1 ... sig_n -> sig))
```

Entsprechend wird die Datendefinition als Kommentar verfeinert, und eine Record-Typ-Definition kommt hinzu:

```
; Ein Kreis besteht aus:
; - Mittelpunkt (point)
; - Radius (real)
(define-record-procedures circle
  make-circle circle?
  (circle-center circle-radius))
(: make-circle (point real -> circle))
```

Die Signaturdeklaration für `make-circle` richtet sich primär an Leser des Programms. Außerdem meldet DrRacket zur Laufzeit Signaturverletzungen, die Anfängern bei der Fehlersuche und -behebung helfen.

Die `define-record-procedures`-Form definiert die Signatur `circle` für Kreise sowie mehrere Prozeduren, die den Umgang mit Record-Werten erlauben:

- den Konstruktor `make-circle`, um Kreise herzustellen,
- das Prädikat `circle?`, um Kreise von anderen Sorten von Werten zu unterscheiden,
- den Selektor `circle-center`, der für einen Kreis dessen Mittelpunkt liefert,
- den Selektor `circle-radius`, der für einen Kreis dessen Radius liefert.

Schließlich wird noch die Signatur für die Konstruktorprozedur `make-circle` deklariert: `make-circle` akzeptiert einen Punkt (den Mittelpunkt des Kreises) und eine reelle Zahl (den Radius) als Eingaben und liefert einen Kreis.

Diese Definitionen folgen direkt und weitgehend mechanisch aus der Datendefinition: Diese besagt, dass ein Kreis aus zwei Teilen besteht – entsprechend muss es zwei Selektoren geben. Der erste Teil ist der Mittelpunkt, ein Punkt – der erste Selektor ist für den Mittelpunkt zuständig, und in der Signatur für `make-circle` ist das erste Argument als `point` ausgemacht, entsprechend für den Radius.

Bei Anfängern ist es außerdem sinnvoll zu verlangen, dass auch die Signaturen für Prädikat und Selektoren notiert werden, die sich aus der Signatur für den Konstruktor direkt ergeben:

```
(: circle? (%a -> boolean))
(: circle-center (circle -> point))
(: circle-radius (circle -> real))
```

Hier wird zusätzlich deutlich, dass `circle?` einen beliebigen Wert als Eingabe akzeptiert und einen Boolean zurückgibt (der anzeigt, ob der Wert einen Kreis repräsentiert oder nicht). Der Selektor `circle-center` akzeptiert einen Kreis als Eingabe und liefert einen Punkt (den Mittelpunkt); `circle-radius` akzeptiert einen Kreis als Eingabe und liefert eine reelle Zahl (den Radius).

Entsprechend für Quadrate:

```
; Ein Quadrat besteht aus:
; - unterer linker Ecke (point)
; - Kantenlänge (real)
(define-record-procedures square
  make-square square?
  (square-corner square-size))
(: make-square (point real -> square))
(: square? (%a -> boolean))
(: square-corner (square -> point))
(: square-size (square -> real))
```

... und für Überlagerungen:

```
; Eine Überlagerung besteht aus:
; - einer geometrischen Figur oben (shape)
; - einer geometrischen Figur unten (shape)
(define-record-procedures overlay
  make-overlay overlay?
  (overlay-top-shape overlay-bot-shape))
(: make-overlay (shape shape -> overlay))
(: overlay? (%a -> boolean))
(: overlay-top-shape (overlay -> shape))
(: overlay-bot-shape (overlay -> shape))
```

Es bleiben die Punkte:

```
; Ein Punkt besteht aus:
; - X-Koordinate (real)
; - Y-Koordinate (real)
(define-record-procedures point
  make-point point?
  (point-x point-y))
(: make-point (real real -> point))
(: point? (%a -> boolean))
(: point-x (point -> real))
(: point-y (point -> real))
```

7.2 Signatur

Die Signaturdeklaration ist eine Aussage über den Namen sowie die Sorten der Ein- und Ausgaben der zu schreibenden Prozedur, ähnelt also einer Typspezifikation für Parameter und Rückgabewert. Aus der Aufgabenstellung ist ersichtlich, dass die Prozedur einen Punkt und eine Figur als Eingaben akzeptiert und einen Wahrheitswert zurückgibt:

```
(: point-in-shape? (point shape -> boolean))
```

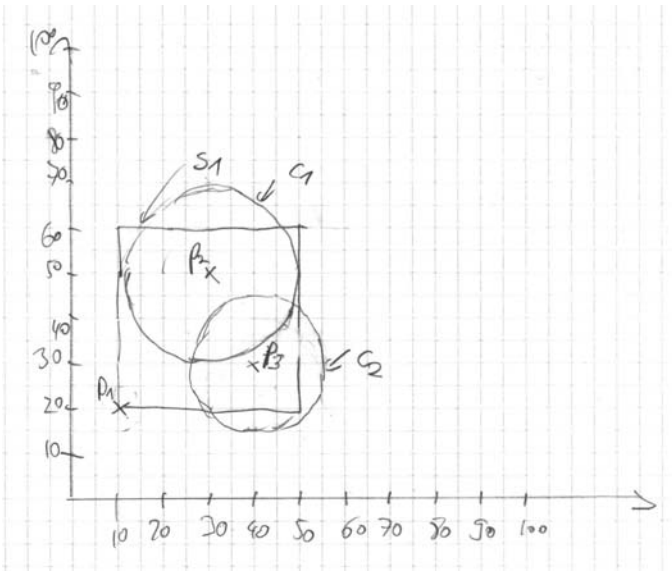

Abbildung 6: Zeichnung zu den Beispielen.

Die Signatur `point` stammt aus der Record-Typ-Definition für Punkte, die Signatur `shape` aus der Signatur für geometrische Figuren. Die Signatur `boolean` ist eingebaut. (Per Konvention sind die Namen von Prozeduren, die einen Wahrheitswert zurückliefern, mit einem Fragezeichen zu beenden.)

7.3 Testfälle

Testfälle erlauben den Studierenden, erste Korrektheitskriterien aufzustellen. Um zu vermeiden, dass die Testfälle nur auf das tatsächliche anstatt auf das erwartete Verhalten testen, werden sie vor der Entwicklung der eigentlichen Prozedur geschrieben.

Der erste Schritt bei der Entwicklung der Testfälle ist die Formulierung von Beispielen für Daten. Diese Beispiele sollten durch Kommentare die Beziehung zwischen den Daten und der repräsentierten Information klarstellen. In diesem Fall könnten Beispiele so aussehen:

```
; Beispiele:
(define p1 (make-point 10 20)) ; Punkt bei X=10, Y=20
(define p2 (make-point 30 50)) ; Punkt bei X=30, Y=50
(define p3 (make-point 40 30)) ; Punkt bei X=40, Y=30
(define s1 (make-square p1 40)) ; Quadrat mit Ecke bei p1, Kantenlänge 40
(define c1 (make-circle p2 20)) ; Kreis um p2, Radius 20
(define o1 (make-overlay c1 s1)) ; Überlagerung von Kreis und Quadrat
(define c2 (make-circle p3 15)); Kreis um p3, Radius 10
(define o2 (make-overlay o1 c2)) ; Überlagerung aus o1 und c2
```

Da die Beispiele geometrischer Natur sind, empfiehlt es sich, eine Skizze anzufertigen wie in Abbildung 6. (Besondere Genauigkeit ist dabei nicht erforderlich.) Die Skizze hilft nun bei der Formulierung von Testfällen für die Prozedur `point-in-shape?`, bei denen die Beispieldaten benutzt werden. Dabei wird die Form `check-expect` benutzt: sie akzeptiert zwei Operanden als Eingaben, deren Werte gleich sein sollen – sind sie es nicht, werden die Testfall-Fehlschläge von DrRacket in einem Testfall-Protokoll festgehalten:

```
(check-expect (point-in-shape? p2 c1) #t)
(check-expect (point-in-shape? p3 c2) #t)
(check-expect (point-in-shape? (make-point 51 50) c1) #f)
(check-expect (point-in-shape? (make-point 11 21) s1) #t)
(check-expect (point-in-shape? (make-point 49 59) s1) #t)
(check-expect (point-in-shape? (make-point 9 21) s1) #f)
(check-expect (point-in-shape? (make-point 11 19) s1) #f)
(check-expect (point-in-shape? (make-point 51 59) s1) #f)
(check-expect (point-in-shape? (make-point 49 61) s1) #f)

(check-expect (point-in-shape? (make-point 40 30) o2) #t)
(check-expect (point-in-shape? (make-point 0 0) o2) #f)
(check-expect (point-in-shape? (make-point 30 65) o2) #t)
(check-expect (point-in-shape? (make-point 40 17) o2) #t)
```

Die Tests können außerdem beim Ausfüllen des Rumpfes später helfen. In typischen HtDP-Kursen wird außerdem noch ein expliziter Unterschied zwischen *Beispielen* gemacht, die an dieser Stelle im Plan in tabellarischer Form angefertigt werden und die *Tests*, die erst im Schritt **Test** aus den Beispielen generiert werden.

7.4 Gerüst

Nun kann die Konstruktion der eigentlichen Prozedur beginnen. Diese beginnt mit dem *Gerüst* – dem Teil der Prozedurdefinition, der sich direkt aus der Signaturdeklaration ergibt. Dazu muss der Studierende Namen für die Parameter wählen und kann dann direkt schreiben

```
(define point-in-shape?
  (lambda (p s)
    ...))
```

Das `define` benennt einen Wert mit dem Namen `point-in-shape?` – der `lambda`-Ausdruck liefert eine Prozedur mit Parametern `p` (für den Punkt) und `s` (für die Figur). Die Ellipse steht für den noch zu schreibenden Rumpf der Prozedur.

7.5 Schablone

Bis zu diesem Zeitpunkt ist in die Konstruktion von `point-in-shape?` noch keinerlei spezifisches Wissen über die Aufgabenstellung außer den Datendefinitionen und den Ein- und Ausgaben eingegangen. Die „eigentliche Programmierung“ beginnt also jetzt, und damit auch der traditionell schwierigste Teil. Dieser Prozess wird durch die Konstruktion einer *Schablone* dramatisch vereinfacht: an der reinen Form der Daten (also ohne die Bedeutung in Betracht zu ziehen oder den Zweck der Prozedur) kann der Studierende bereits Programmelemente ablesen, die im späteren Rumpf auftauchen.

Da die Signaturen der meisten Prozeduren mehrere Datensorten enthalten, gehen in die Schablone des Rumpfes in der Regel mehrere Teilschablonen ein, eine für jeden Parameter sowie ggf. eine für die Rückgabesorte. Die Studierenden sind gehalten, die Schritte separat abzuarbeiten und aufzuschreiben, insbesondere die Ellipsen für die noch unbekannten Teile des Prozedurrumpfes.

1. Die Parameter `p` und `s` müssen im Rumpf auftauchen:

   ```
   (define point-in-shape?
     (lambda (p s)
       ... p ... s ...))
   ```

2. Da es sich bei `p` um zusammengesetzte Daten handelt, wird eine Prozedur wahrscheinlich die Bestandteile des Punktes benötigen. Die Schablone wird also erweitert:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...
    ... (point-x p) ... (point-y p) ...))
```

3. Da es sich bei `s` um gemischte Daten handelt, muss die Prozedur zwischen den verschiedenen Arten geometrischer Figuren mit einer Verzweigung unterscheiden. Die Konstruktionsanleitung für gemischte Daten beschreibt dabei die Schablone folgendermaßen:

   > Wenn die Prädikate für die einzelnen Sorten $pred?_1$ ... $pred?_n$ heißen, hat die Schablone für eine Prozedur, die gemischte Daten als Eingabe akzeptiert, die folgende Form:
   >
   > ```
   > (: proc (sig -> ...))
   >
   > (define proc
   >   (lambda (a)
   >     (cond
   >       ((pred?_1 a) ...)
   >       ...
   >       ((pred?_n a) ...))))
   > ```
   >
   > Die rechten Seiten der Zweige werden dann nach den Konstruktionsanleitungen der einzelnen Sorten ausgefüllt.

   In der Vorlesungspraxis wird dieser Teil der Konstruktionsanleitung für die Studierenden i.d.R. noch weiter aufgeschlüsselt, um Fehler bei der Anwendung zu vermeiden: Insbesondere ist wichtig, dass die Anzahl der Fälle in der Datendefinition der Zahl der Zweige im `cond` entspricht. Hier: „Wie viele Fälle hat die Datendefinition für geometrische Figuren?“ „Drei!“ „Wie viele Zweige hat also die Verzweigung?“ „Auch drei!“ (In einem HtDP-Kurs werden typischerweise diese Zusammenhänge auf der Rückseite der Karten mit den Konstruktionsanleitungen notiert.) Der Dozent spielt ähnliche Frage-und-Antwort-Folgen z.B. bei der Anzahl der Komponenten zusammengesetzer Daten und bei anderen Aspekten der Konstruktionsanleitungen durch. Obwohl die Zusammenhänge aus Sicht des Dozenten trivial sind, machen Studierende ohne solche Anleitung oft gerade bei solchen Aspekten der Programmkonstruktion Fehler.

   In diesem Fall entsteht direkt folgende Schablone:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...
    ... (point-x p) ... (point-y p) ...
    (cond
      ((circle? s) ...)
      ((square? s) ...)
      ((overlay? s) ...))))
```

4. Bei jedem der Fälle von `shape` – `circle`, `square`, `overlay` – handelt es sich um gemischte Daten. In den Fällen der Verzweigung werden also wie bei `p` die Selektoren aufgerufen:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...
    ... (point-x p) ... (point-y p) ...
    (cond
      ((circle? s)
       ... (circle-center s) ... (circle-radius s) ...)
      ((square? s)
       ... (square-corner s) ... (square-size s) ...)
      ((overlay? s)
       ... (overlay-top-shape s) ... (overlay-bot-shape s) ...))))
```

5. Schließlich gibt es im Fall `overlay` zwei Selbstbezüge. Diese Selbstbezüge führen zu rekursiven Aufrufen:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...
    ... (point-x p) ... (point-y p) ...
    (cond
      ((circle? s)
       ... (circle-center s) ... (circle-radius s) ...)
      ((square? s)
       ... (square-corner s) ... (square-size s) ...)
      ((overlay? s)
       ... (point-in-shape? p (overlay-top-shape s))
       ... (point-in-shape? p (overlay-bot-shape s)) ...))))
```

Die Programmelemente der Schablone sind das Rohmaterial für den späteren vollständigen Rumpf der Prozedur. Die Schablone ist von zentraler Bedeutung für den Konstruktionsprozess – sie wird schrittweise konstruiert und entsteht rein *datengesteuert*, ergibt sich also aus der Signatur und den Datendefinitionen der in der Signatur vorkommenden Sorten. Es hätte also jede Prozedur mit der gleichen Signatur, unabhängig von ihrer Aufgabe, die gleiche Schablone. Die Studierenden müssen sich also bei diesem Punkt keine Gedanken über die eigentliche Aufgabe machen und können trotzdem Fortschritte erzielen. Ohne diesen Schritt scheitern viele Studierende bereits, ohne überhaupt mit dem Schreiben des Rumpfes der Prozedur begonnen zu haben.

7.6 Rumpf

Bei der Konstruktion des Rumpfes geht es nur noch darum, die Lücken in den Fällen der Verzweigung zu füllen:

- Im ersten Fall – `circle` – stehen bereits Mittelpunkt und Radius in der Schablone. Hier müssen die Studierenden *bereichsspezifisches Wissen* einsetzen – also die Erkenntnis, dass ein Punkt innerhalb eines Kreises liegt, wenn der Abstand des Punktes zum Mittelpunkt des Kreises kleiner als sein Radius ist. Der Abstand ist eine *Zwischengröße* und verdient damit eine eigene Prozedur mit folgender Kurzbeschreibung und folgender Signatur:

```
; Abstand zwischen zwei Punkten berechnen
(: distance (point point -> real))
```

Diese Prozedur wird zunächst vorausgesetzt („Wunschdenken“) und später geschrieben. Damit ergibt sich der Fall für Kreise direkt aus der obigen Formulierung:

```
...
    (cond
      ((circle? s)
       (<= (distance p (circle-center s))
           (circle-radius s)))
...
```

- Im zweiten Fall – `square` – müssen die Koordinaten des Punktes mit den Koordinaten der Quadratecke verglichen werden: Diese Elemente stehen bereits in der Schablone. Auch dies ist nur mit bereichsspezifischem Wissen möglich, aber nicht besonders schwierig:

```
...
    (cond
      ...
      ((square? s)
       (and (>= (point-x p) (point-x (square-corner s)))
            (<= (point-x p)
                (+ (point-x (square-corner s)) (square-size s)))
            (>= (point-y p) (point-y (square-corner s)))
            (<= (point-y p)
                (+ (point-y (square-corner s)) (square-size s)))))
...
```

- Im dritten Fall stehen schon zwei rekursive Aufrufe in der Schablone, die folgende Fragen beantworten:
  - Ist der Punkt innerhalb der ersten Teilfigur?
  - Ist der Punkt innerhalb der zweiten Teilfigur?

  Es geht also nur noch darum, wie diese beiden Antworten zur Antwort auf die Frage „Ist der Punkt innerhalb der Überlagerung?“ kombiniert werden müssen. Der Kombinator ist das logische „Oder“.

```
...
    (cond
      ...
      ((overlay? s)
       (or (point-in-shape? p (overlay-top-shape s))
           (point-in-shape? p (overlay-bot-shape s)))))))
...
```

Damit ist die Konstruktion des Rumpfes abgeschlossen, der vollständig so aussieht:

```
(define point-in-shape?
  (lambda (p s)
    (cond
      ((circle? s)
```

```
      (<= (distance p (circle-center s))
          (circle-radius s)))
     ((square? s)
      (and (>= (point-x p) (point-x (square-corner s)))
           (<= (point-x p)
               (+ (point-x (square-corner s)) (square-size s)))
           (>= (point-y p) (point-y (square-corner s)))
           (<= (point-y p)
               (+ (point-y (square-corner s)) (square-size s)))))
     ((overlay? s)
      (or (point-in-shape? p (overlay-top-shape s))
          (point-in-shape? p (overlay-bot-shape s)))))))
```

Zusatz: Später werden Studierende ermutigt, mehrfach vorkommende Ausdrücke (z.B. `(point-x corner)` oder `(point-x p)` an lokale Variablen zu binden. Das Ergebnis sähe dann so aus:

```
(define point-in-shape?
  (lambda (p s)
    (cond
      ((circle? s)
       (<= (distance p (circle-center s))
           (circle-radius s)))
      ((square? s)
       (let ((corner (square-corner s)))
         (let ((cx (point-x corner))
               (cy (point-y corner))
               (size (square-size s))
               (x (point-x p))
               (y (point-y p)))
           (and (>= x cx)
                (<= x (+ cx size))
                (>= y cy)
                (<= y (+ cy size))))))
      ((overlay? s)
       (or (point-in-shape? p (overlay-top-shape s))
           (point-in-shape? p (overlay-bot-shape s)))))))
```

Damit das Programm vollständig wird, fehlt noch die Hilfsprozedur `distance`, die in `point-in-shape?` per Wunschdenken vorausgesetzt ist. Auch sie entsteht durch Anwendung der Konstruktionsanleitungen sowie die Anwendung bereichsspezifischen Wissens – der Formel für den Abstand. Das Ergebnis (aus Platzgründen ohne die Tests) sieht folgendermaßen aus:

```
; Abstand zwischen zwei Punkten berechnen
(: distance (point point -> real))
; ... Tests ...
(define distance
  (lambda (p1 p2)
    (sqrt
     (+ (sqr (- (point-x p1) (point-x p2)))
        (sqr (- (point-y p1) (point-y p2)))))))

; Zahl quadrieren
```

```
(: sqr (real -> real))
; ... Tests ...
(define sqr
  (lambda (x)
    (* x x)))
```

7.7 Zusammenfassung

Die Konstruktionsanleitungen sind ein systematischer Ansatz für die Konstruktion von Programmen:

- Die Konstruktionsanleitungen arbeiten „all the way down“, steuern also den Programmkonstruktionsprozess von der Problemanalyse bis zur Fertigstellung des Programms.
- Die Einteilung in separate Schritte, von denen jeder einen Teil des späteren Programms beiträgt, bietet zu nahezu jedem Zeitpunkt klare Hilfestellung für den Anfänger.
- Die systematische Vorgehensweise mechanisiert und vereinfacht damit weite Teile der Programmkonstruktion; der „kreative“ Teil, in dem bereichsspezifisches Wissen eingebracht wird, ist klar isoliert.

Die Konstruktionsanleitungen beschäftigen sich nicht direkt damit, wie aus unklaren oder mehrdeutigen Problemformulierunge eindeutige Aufgabenstellungen gewonnen werden. Durch die Klassifikation von Daten bieten sie allerdings sprachliche Hilfsmittel bei der Formulierung dieser Aufgabenstellungen.

Die Konstruktionsanleitungen entfalten in der Anfängerausbildung ihre Wirkung bei Verwendung mit einer anfängergerechten Programmiersprache [FFFK02]. Diese Lehrsprachen erlauben die direkte Abbildung der Konzepte, derer sich die Konstruktionsanleitungen bedienen und erlauben damit Anfängern frühe Erfolgerlebnisse ohne die Verwendung „magischer“ Programmfragmente, die vom Dozenten gestellt werden müssen, ohne dass die Bedeutung früh erklärt werden kann.

Die Verwendung der anfängergerechten Programmierumgebung DrRacket und die Verwendung von Sprachebenen unterstützt Anfänger zusätzlich beim erfolgreichen eigenständigen Konstruieren von Programmen [CS10].

7.8 Erfahrungen

Das Konzept der Konstruktionsanleitungen hat sich in der Lehrpraxis, sowohl in Schulen (im Rahmen des TeachScheme!-Projekts [FFFK04]) als auch im Rahmen der universitären Lehre vielfach bewährt [BDH+08]. Die Erfahrung zeigt, dass die Konstruktionsanleitungen Blockaden beseitigen, die viele Anfänger davon abhalten, eigenständig funktionierende Programme zu schreiben. Umgekehrt haben Anfänger in solchen Kursen vor allem dann Schwierigkeiten, wenn sie die Konstruktionsanleitungen nicht anwenden und stattdessen versuchen, „auf eigene Faust“ zu programmieren. Es empfiehlt sich deshalb, einen Teil der praktischen Übungen unter Aufsicht von Tutoren durchzuführen [BDH+08]. Die Hilfestellung, welche die Tutoren leisten, fällt durch die Konstruktionsanleitungen eindeutig und nachvollziehbar aus. Gleichzeitig liefert solche Aufsicht wertvolle Beobachtungen, die helfen können, einen laufenden Kurs kontinuierlich zu verbessern.

Die Konstruktionsanleitungen offenbaren eine andere Gewichtung von Schwierigkeiten in der Programmierausbildung:

- Der Umgang mit zusammengesetzten Daten – ein Thema, das in der traditionellen Programmierausbildung oft nur gestreift wird – verdient besondere

Aufmerksamkeit: Das Konzept „mehrere Dinge bilden zusammen ein Ganzes“ ist für viele Studierende anfänglich schwierig. Durch ausführliche Behandlung können diese Probleme jedoch ausgeräumt werden [BDH+08].

- Es ist nicht nötig, „Schleifen“ einzuführen, um Berechnungen über lineare zusammengesetzte Daten wie Listen durchzuführen. Stattdessen werden hierfür die schon bekannten Prozeduraufrufe verwendet; aus den Selbstreferenzen in den Daten werden rekursive Aufrufe. Die Higher-Order-Programmierung ermöglicht dann, verschiedene schleifenartige Abstraktionen wie `map` (eine Operation auf alle Elemente einer Liste) oder `fold` (aus allen Elementen einer Liste einen kumulativen Wert berechnen) selbst zu programmieren (wiederum, ohne neue Konstrukte der Programmiersprache einzuführen) und zu benutzen.

- Der Umgang mit Rekursion bzw. Schleifen –traditionell schwierige Themen in der traditionellen Programmierausbildung – ist dank der Verknüpfung mit der Datenmodellierung unproblematisch; sie stellt deshalb keine besondere Schwierigkeit für die Studierenden dar [BDH+08, BDM05].

- Rein iterative „Schleifen“ werden im Rahmen dieser Ausbildung unter der Überschrift „Programmieren mit Akkumulatoren“ behandelt – dafür wird statt spezieller Schleifenkonstrukte wieder Rekursion benutzt.

Gegenüber der Verwendung imperativer Sprachen vermeidet die Verwendung rein funktionaler Lehrsprachen eine Vielzahl von Problemen, mit denen Anfänger sonst konfrontiert werden [FFFK02]. Dazu gehören die Schwierigkeiten im Zusammenhang mit mutierbaren Variablen, Objektidentität und Referenzen [BDS08], die typischen unsystematischen Schleifenkonstrukte, die fehlerhafte Unterstützung von Endrekursion sowie die häufig historisch gewachsenen idiosynkratischen Konstrukte dieser Programmiersprachen. Das in Abschnitt 6.2 beschriebene Zuordnungsproblem, also die Frage, welcher Klasse eine Methode zugeordnet wird, entsteht ebenfalls nicht. Die rein funktionale Programmierung schließt hingegen direkt an mathematische Notation an und erlaubt deshalb einen seichten Übergang von der Schulmathematik.

Anfänger machen beim Programmieren außerdem oft einfache Fehler und sind oft schnell demotiviert, wenn sie diese Fehler nicht eigenständig korrigieren können: Bei professionellen Programmierumgebungen machen es die Komplexität der Programmiersprachen und die an professionelle Programmierer gerichteten Fehlermeldungen (die oft die Kenntnis der gesamten Programmiersprache voraussetzen) Anfängern oft schwer, eigenständig zum Erfolg zu kommen. [FCF+02, Rey09]. Die Lehrsprachen können einige aber nicht alle dieser Fehler ausschließen. Deshalb ist die Programmierumgebung zusätzlich gefordert, Anfänger zu unterstützen. Die Implementierung der Anfänger-Sprachebenen in DrRacket werden darum ständig aufgrund konkreter Beobachtungen und Erfahrungen von Studierenden angepasst, um bestmögliche Unterstützung zu leisten [CS10].

8 Vergleich der Lösungen

Sowohl die Java-Lösung und die Lehrsprachen-Lösung bauen auf äquivalenten Datendefinitionen auf, sind also auch in ihrer unmittelbaren Funktionalität vergleichbar. Es gibt dennoch einige Unterschiede:

- Wie schon angesprochen ist in der Java-Lösung der Hauptparameter zu den regulären Parametern asymmetrisch.

- Während sich die objektorientierte Lösung um weitere `Shape`-Klassen erweitern lässt, ohne die bestehenden Klassen zu ändern, ermöglicht die Lehrsprachen-Lösung es, weitere Operationen auf Figuren zu implementieren, ohne die bestehenden Prozeduren zu ändern. Dies sind die klassischen, zueinander orthogonalen Erweiterungsprobleme der objektorientierten respektive funktionalen Programmierung [FF98].
- Während die Tests im Lehrsprachen-Code integriert sind, müssen sie in Java ausgelagert werden, um z.B. in das JUnit-Framework zu passen.

9 Schlussfolgerungen

Das Beispiel der „geometrischen Figuren“ ist klein, zeigt aber bereits die wesentlichen Eigenschaften verschiedener methodischer Ansätze im Einsatz bei der Anfängerausbildung:

Ein wesentlicher Unterschied zwischen der OOA/OOD-Methode und den Konstruktionsanleitungen ist, dass bei OOA/OOD der Entwurfsprozess vom Programmieren getrennt ist, während jeder Schritt der Konstruktionsanleitungen zu einem Element des späteren Programms führt. Die Konstruktionsanleitungen sind durchgängig *konstruktiv*, während bei OOA/OOD nur der Entwurf konstruktiv ist. Der Großteil der OOA/OOD-Techniken funktioniert *deskriptiv*: Bei OOA/OOD wird das Programmieren meist durch Heuristiken [Rie96] gesteuert, die ein fertiges Programm bewerten, aber keine direkte Technik beschreiben, wie der Programmierer das Programm entsprechend der Heuristiken konstruiert. Die Konstruktionsanleitungen bieten dagegen eine direkte und durchgängige Anleitung zur Konstruktion des Programms.

Beim Experiment während des Workshops war dementsprechend der OOA/OOD-Ansatz nur bedingt erfolgreich: Die linguistische Analyse lieferte eine Aufteilung in Klassen und die CRC-Karten wurden entsprechend der Darstellung in Abschnitt 6.2 erstellt. Für die Programmierung des konkreten Programms, insbesondere die Zuordnung der `isInside`-Methode sowie die rekursiven Aufrufe in `Overlay` lieferte das Entwurfsmodell nicht genügend Informationen. Die Konstruktionsanleitungen hingegen liefern direkt nachvollziehbar eine Lösung für die Aufgabe.

Die möglichen Stärken der OOA/OOD-Techniken bei der Aufbereitung unscharfer Aufgabenstellungen kamen bei dieser Aufgabe nur bedingt zum Tragen:

In der OOA/OOD-Entwurfsphase verschafft die linguistische Analyse einen Überblick darüber, welche Daten aus einer Aufgabenstellung im späteren Programm modelliert werden können. Die Datenanalyse aus den Konstruktionsanleitungen hingegen baut bereits auf einer Vorstellung auf, *welche* Daten repräsentiert werden müssen, und gibt dann Anleitung, *wie* die Repräsentation konstruiert wird.

Entsprechend der linguistischen Analyse kann die Szenario-Analyse mit Rollenspielen helfen zu identifizieren, *welche* Operationen zu implementieren sind. Die CRC-Karten helfen primär in einem objektorientierten Setting, die dort notwendige Zuordnung von Methoden zu Klassen vorzunehmen. Beide Verfahrensweisen sind aber nicht direkt konstruktiv, sondern helfen lediglich, die gedanklichen Prozesse des Anfängers zu ordnen. Insbesondere die Szenario-Analyse hilft vor allem bei noch unklaren, mehrdeutigen oder noch nicht aufgearbeiteten Problemstellungen. Sie kann auch benutzt werden, um Methodensignaturen zu entwerfen.

10 Gegenüberstellung

Prinzipiell sind sowohl der hier dargestellte OOA/OOD-Ansatz als auch die Konstruktionsanleitungen *datengesteuert*: Die Struktur des Programms richtet sich nach

der Struktur der Daten – dies steht im Gegensatz zu eher algorithmisch orientierten Ansätzen. Damit sind beide Ansätze prinzipiell vergleichbar. Dieser Abschnitt stellt die analogen Techniken jeweils gegenüber und diskutiert Kombinationsmöglichkeiten.

linguistische Analyse und Datenanalyse Diese Schritte sind direkt analog zueinander: Während die linguistische Analyse im hier präsentierten Zusammenhang eine systematische Technik zur Identifikation der in einer Aufgabenstellung vorkommenden Datensorten ist, beschäftigt sich die Datenanalyse zentral mit der Modellierung der identifizierten Datensorten – diese Techniken können also problemlos kombiniert werden.

Wie in Abschnitt 4.1 dargestellt, kann linguistische Analyse prinzipiell auch benutzt werden, um *Beziehungen* zwischen Datensorten bzw. Klassen zu ermitteln. Die Konstruktionsanleitungen setzen hier auf die Identifikation der *Struktur* der Datensorten innerhalb der Taxonomie primitiver, gemischter und zusammengesetzer Daten.

CRC-Karten und Datendefinitionen Das Ergebnis der Datenanalyse – die Datendefinitionen – entspricht zwar oberflächlich den CRC-Karten. Tatsächlich aber haben die Datendefinitionen die Aufgabe, die Information einer Aufgabenstellung möglichst direkt abzubilden, während die CRC-Karten sich an Rolle und Verantwortlichkeiten orientieren. Dies führt in der Praxis oft zu unterschiedlichen Modellen. Operationen werden bei den Konstruktionsanleitungen (zumindest, wenn sie mit den funktionalen Lehrsprachen verwendet werden) nicht den Datendefinitionen zugeordnet, und Kollaborateure ergeben sich aus den Signaturen.

Es sollte allerdings problemlos möglich sein, im Rahmen einer Ausbildung im objektorientierten Programmieren aus den Datendefinitionen CRC-Karten zu erstellen, die dann ihrerseits helfen können, Methoden zuzuordnen.

Rollenspiel und Kurzbeschreibungen und Signaturen Das Rollenspiel identifiziert Aspekte der noch zu schreibenden Methoden, die sich bei den Konstruktionsanleitungen in der Signatur wiederfinden. Trotzdem gibt es wichtige Unterschiede, die eine Kombination problematisch erscheinen lassen:

- Rollenspiele legen explizit das Message-Passing-Paradigma zugrunde – dass also Operationen bestimmten Objekten zugeordnet sind –, das es in den funktionalen Lehrsprachen nicht gibt.

- Da in Rollenspielen für jedes Objekt ein Zettel angelegt wird, funktionieren sie am besten, wenn die Anzahl der Objekte in einem Szenario überschaubar ist. In imperativen Programmen, die im wesentlichen in Methodenaufrufen den Zustand existierender Objekte ändern, ist das eher der Fall als in funktionalen Programmen, die neue Objekte erzeugen, um neue Zustände zu repräsentieren. Das schränkt die Anwendbarkeit des Rollenspiels auf funktionale Programmierung ein, die dafür die bekannten Probleme der imperativen Programmierung mit Reihenfolge, Nebenläufigkeit vermeidet.

  In der Beispielaufgabe werden Figuren nicht verändert bzw. es werden keine neuen Figuren erzeugt. Entsprechend wird dieses Problem am Beispiel nicht sichtbar.

- Das Rollenspiel suggeriert die Objektidentität als wichtige Größe, die bei der funktionalen Programmierung unwichtig ist.

How to Design Classes Der Kurs *How to Design Classes*[4] (*HtDC*) ist als Folgekurs zu einem auf Konstruktionsanleitungen basierenden Kurs konzipiert und überträgt sie direkt auf die objektorientierte Programmierung in Java. Gegenüber dem ersten Kurs kommen noch Vererbung und Interfaces zum Repertoire der Datenanalyse dazu. Außerdem wird das Ergebnis der Datenanalyse zunächst als Klassendiagramm-Skizze festgehalten, bevor konkreter Code geschrieben wird.

Da Java z.B. eine Vielzahl spezialisierter Abstraktionsmechanismen bietet, ist Formulierung und Umsetzung der Konstruktionsanleitungen i.d.R. zwar möglich, jedoch deutlich umständlicher und aufwendiger als in den Scheme-basierten Lehrsprachen. Die für Java vorhandenen Entwicklungsumgebungen bieten außerdem nicht den gleichen Grad der Unterstützung für Anfänger wie DrRacket. Insbesondere bieten zwar auch Entwicklungsumgebungen für Anfänger wie *BlueJ* [KR96], das mit dem Kurs verwendet werden kann, Unterstützung für Test-Frameworks wie JUnit. Das Schreiben von Tests ist aber auch dann deutlich aufwendiger als bei den Scheme-Lehrsprachen.

Mit fortschreitender Abdeckung der Sprache offenbart der Kurs zunehmend Schwächen von Java – der deutlich größere Umfang der Programme gegenüber Scheme, die mangelhaften Fehlermeldungen des Java-Compilers [Jad05], das Fehlen korrekter Endrekursion, die Vielzahl eingeschränkter Abstraktionsmechanismen, die nur imperativ zu benutzenden Schleifenkonstrukte sowie die Integrationsnähte vieler nachträglich eingefügter Sprachfeatures wie Generics, for-each-Schleifen etc. Diese Probleme machen es zunehmend schwierig, in der Lehre Konzepte von ihrer Realisierung in Java zu trennen.

11 Zusammenfassung

Es ist erfahrungsgemäß in der Anfängerausbildung besonders wichtig, dass Studierende schnell befähigt werden, eigenständig Programme zu schreiben. Die Kombination der Konstruktionsanleitungen mit speziell darauf zugeschnittenen Lehrsprachen sowie einer Anfänger-Programmierumgebung erleichtern diesen Einstieg auf besondere Weise. Die vorgestellten OOA/OOD-Methoden helfen, insbesondere komplexe, unklare oder mehrdeutige Aufgabenstellungen für die Programmierung aufzubereiten. Beide Ansätze gehen datengesteuert vor und sind damit prinzipiell kombinierbar. Insbesondere sind die Konstruktionsanleitungen direkt auf die objektorientierte Programmierung übertragbar, leiden bei den für professionelle Programmierer gemachten objektorientierten Programmiersprachen aber unter aufwendiger Notation und Idiosynkrasien.

Danksagung Wir danken Matthias Felleisen für die Mitarbeit an dem Experiment sowie Christian Wagenknecht für die kritische Durchsicht einer frühen Version dieses Papiers.

Literatur

[BC89] Beck, Kent ; Cunningham, Ward: A Laboratory For Teaching Object-Oriented Thinking. In: *OOPSLA'89 Conference Proceedings*. New Orleans, Louisiana, 1989, S. 1–6

[BDH+08] Bieniusa, Annette ; Degen, Markus ; Heidegger, Phillip ; Thiemann, Peter ; Wehr, Stefan ; Gasbichler, Martin ; Crestani, Marcus ; Klaeren, Herbert ; Knauel, Eric ; Sperber, Michael: HtDP

[4] `http://www.ccs.neu.edu/home/vkp/HtDCH/`

and DMdA in the Battlefield. In: Huch, Frank (Hrsg.) ; Parkin, Adam (Hrsg.): *Functional and Declarative Programming in Education*. Victoria, BC, Canada, September 2008

[BDM05] Bruce, Kim B. ; Danyluk, Andrea ; Murtagh, Thomas: Why structural recursion should be taught before arrays in CS 1. In: *Proceedings of the 2005 ACM Symposium on Computer Science Education*, 2005, S. 246–250

[BDS08] Bornat, Richard ; Dehnadi, Saeed ; Simon: Mental models, consistency and programming aptitude. In: *ACE '08: Proceedings of the tenth conference on Australasian computing education*. Darlinghurst, Australia, Australia, 2008, S. 53–61

[Boo82] Booch, Grady: Object-oriented design. In: *Ada Letters* I (1982), Nr. 3, S. 64–76

[Bör04] Börstler, Jürgen: Object-Oriented Analysis and Design Through Scenario Role-Play / Dept. of Computing Science. Umeå University, Umeå, Sweden, 2004 (UMINF-04.04). – Forschungsbericht

[Bör05] Börstler, Jürgen: Improving CRC-card role-play with role-play diagrams. In: *OOPSLA'05 Conference Companion* (2005), Oct, S. 356–364

[Bri04] Brinda, T.: *Didaktisches System für objektorientiertes Modellieren im Informatikunterricht der Sekundarstufe II*, Universität Siegen, Fachbereich Elektrotechnik und Informatik, Dissertation, March 2004. `http://www.ub.uni-siegen.de/pub/diss/fb12/2004/brinda/brinda.pdf`

[BS97] Bellin, David ; Simone, Susan S.: *The CRC Card Book*. Reading, MA : Addison-Wesley, 1997

[CS10] Crestani, Marcus ; Sperber, Michael: Growing Programming Languages for Beginning Students. In: Weirich, Stephanie (Hrsg.): *Proceedings International Conference on Functional Programming 2010*. Baltimore, Maryland, USA : ACM Press, New York, September 2010

[DB86] Du Boulay, B.: Some difficulties of learning to program. In: *Journal of Educational Computing Research* 2 (1986), Nr. 1, S. 57–73

[Die07] Diethelm, Ira: *„Strictly models and objects first" – Unterrichtskonzept und -methodik für objektorientierte Modellierung im Informatikunterricht*, Universität Kassel, Fachbereich Elektrotechnik / Informatik, Dissertation, Mai 2007. `http://kobra.bibliothek.uni-kassel.de/bitstream/urn:nbn:de:hebis:34-2007101119340/1/DissIraDruckfassungA5.1.pdf`

[EMM+06] Eckerdal, A. ; McCartney, R. ; Moström, J.E. ; Ratcliffe, M. ; Zander, C.: Categorizing student software designs: Methods, results, and implications. In: *Computer Science Education* 16 (2006), Nr. 3, S. 197–209

[FCF+02] Findler, Robert B. ; Clements, John ; Flanagan, Cormac ; Flatt, Matthew ; Krishnamurthi, Shriram ; Steckler, Paul A. ; Felleisen, Matthias: DrScheme: A Programming Environment for Scheme. In: *Journal of Functional Programming* (2002), März, S. 159–182

[FF98] Findler, Robert B. ; Flatt, Matthew: Modular Object-Oriented Programming with Units and Mixins. In: Hudak, Paul (Hrsg.): *Proceedings International Conference on Functional Programming 1998*. Baltimore, MD, USA : ACM Press, New York, September 1998. – ISBN 1–58113–024–4

[FFFK01] Felleisen, Matthias ; Findler, Robert B. ; Flatt, Matthew ; Krishnamurthi, Shriram: *How to Design Programs*. MIT Press, 2001

[FFFK02] Felleisen, Matthias ; Findler, Robert B. ; Flatt, Matthew ; Krishnamurthi, Shriram: The Structure and Interpretation of the Computer Science Curriculum. In: *Functional and Declarative Programming in Education (FDPE)*, 2002, S. 21–26

[FFFK04] Felleisen, Matthias ; Findler, Robert B. ; Flatt, Matthew ; Krishnamurthi, Shriram: The TeachScheme! Project: Computing and Programming for Every Student. In: *Computer Science Education* (2004), März

[HH07] Hadar, I. ; Hadar, E.: An iterative methodology for teaching object oriented concepts. In: *Informatics in education* 6 (2007), Nr. 1, S. 67–80

[HJ94] Hudak, Paul ; Jones, Mark P.: Haskell vs. Ada vs. C++ vs. Awk vs. ..., An Experiment in Software Prototyping Productivity / Yale University, Department of Computer Science. Version: Juli 1994. `http://haskell.org/papers/NSWC/jfp.ps`. New Haven, CT 06518, Juli 1994. – Forschungsbericht

[Jad05] Jadud, M. C.: A first look at novice compilation behavior using BlueJ. In: *Computer Science Education* 15 (2005), Nr. 1

[JML00] Juristo, Natalia ; Moreno, Ana M. ; López, Marta: How to Use Linguistic Instruments for Object-Oriented Analysis. In: *IEEE Software* 17 (2000), Nr. 3, S. 80–89

[KR96] Kölling, M. ; Rosenberg, J.: An Object-Oriented Program Development Environment for the First Programming Course. In: *Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education*. Philadelphia, Pennsylvania : ACM, März 1996

[Kri94] Kristen, Gerald: *Object-Orientation: The KISS Method*. Boston, MA, USA : Addison-Wesley, 1994

[KS07] Klaeren, Herbert ; Sperber, Michael: *Die Macht der Abstraktion*. Erste Auflage. Teubner Verlag, 2007

[LSS+04] Lister, R. ; Seppälä, O. ; Simon, B. ; Thomas, L. ; Adams, E.S. ; Fitzgerald, S. ; Fone, W. ; Hamer, J. ; Lindholm, M. ; McCartney, R. u. a.: A multi-national study of reading and tracing skills in novice programmers. In: *ACM SIGCSE Bulletin* 36 (2004), Nr. 4, S. 119–150

[MWA+01] McCracken, M. ; Wilusz, T. ; Almstrum, V. ; Diaz, D. ; Guzdial, M. ; Hagan, D. ; Kolikant, Y.B.D. ; Laxer, C. ; Thomas, L. ; Utting, I.: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In: *ACM SIGCSE Bulletin* 33 (2001), Nr. 4, S. 125–180

[OLR01] Overmyer, S.P. ; Lavoie, B. ; Rambow, O.: Conceptual Modeling through Linguistic Analysis Using LIDA. In: *Proceedings of the 23rd International Conference on Software Engineering*, 2001, S. 401–410

[Rey09] Rey, Jeanette S.: *From Alice to BlueJ : A Transition to Java*, The Robert Gordon University, Diplomarbeit, 2009

[Rie96] Riel, Arthur J.: *Object-Oriented Design Heuristics*. Addison-Wesley, 1996

[SS88] Soloway, E. ; Spohrer, J.C.: *Studying the novice programmer*. Hillsdale, NJ, USA : L. Erlbaum Associates Inc., 1988

[WBM03] Wirfs-Brock, Rebecca ; McKean, Alan: *Object Design–Roles, Responsibilities, and Collaborations*. Boston, MA : Addison-Wesley, 2003

[Wes04] West, David: *Object Thinking*. Redmond, WA : Microsoft Press, 2004