Das Sieben-Schritte-Schema zur Dekonstruktion objektorientierter Software

Carsten Schulte, Ulrich Block

Didaktik der Informatik
Universität Paderborn
carsten|ulibl@uni-paderborn.de

Abstract: Im Informatikunterricht bedeutet Softwareentwicklung meist, ein kleineres Programm von Grund auf neu zu entwickeln. Dekonstruktion dagegen geht von einer fertigen Software aus, die im Unterricht dekonstruiert und erweitert wird. Damit wird ein realistischeres Bild von Softwareentwicklung vermittelt und gleichzeitig können so Wechselwirkungen zwischen Software und Einsatzkontext betrachtet werden. Im Folgenden wird die Unterrichtsmethode der Dekonstruktion fachlich anhand softwaretechnischer Sichtweisen und didaktisch anhand lerntheoretischer Überlegungen erläutert und präzisiert. Darauf basierend entwickeln wir ein Sieben-Schritte-Schema, mit dem objektorientierte Software dekonstruiert und erweitert werden kann. Die unterrichtliche Umsetzung des Schemas wird an einem Beispiel skizziert.

1 Informatiksysteme als Ausgangspunkt

Im Ansatz der Systemorientierten Didaktik [Ma00] ist die Dekonstruktion von Informatiksystemen [HMS99] eine wesentliche Unterrichtsmethode; das soziotechnische Informatiksystem (im Folgenden als stIFS abgekürzt) der Ausgangspunkt der unterrichtlichen Thematisierung informatischer Inhalte. Dieser Ausgangspunkt korrespondiert mit den vier Leitlinien des Gesamtkonzepts zur informatischen Bildung der Gesellschaft für Informatik (GI): (1) Interaktion mit Informatiksystemen, (2) Wirkprinzipien von Informatiksystemen, (3) Informatische Modellierung, (4) Wechselwirkungen zwischen Informatiksystemen, Individuum und Gesellschaft. Dabei wird die „Vermittlung von Hintergrundwissen in allen Phasen der informatischen Bildung, von der einfachen Anwendung eines Computers bis zur eigenen Gestaltung von Anwendungen“ [GI00] hervorgehoben. Zusammengenommen führen die Leitlinien zu einem selbstbewussten und reflektierten Umgang mit Informatik und informatischen Systemen. Die Schülerinnen und Schüler in der Sekundarstufe II sollen sich dazu „die Basiskonzepte ausgewählter Informatiksysteme durch Anwendung, Analyse, Modifikation und Bewertung“ [GI00] aneignen.
Es ergibt sich die Frage der unterrichtspraktischen Umsetzung. Anwenden und Modifikation (des Quelltextes) sind eher typische unterrichtliche Tätigkeiten. Aber wie lassen sich die Aktivitäten Analyse und Bewertung im Informatikunterricht unterbringen? Bei der Beantwortung dieser Frage hilft eine etwas weiter gefasste Betrachtungsweise, welche soziotechnische Informatiksysteme (stIFS) ins Blickfeld rückt.
Ein stIFS besteht aus Computer plus Software sowie den Entwicklern, Nutzern und Betroffenen der Nutzung (siehe Abb. 1), von denen wir im Folgenden zusammenfassend

als (verschiedene) sog. 'Interessensgruppen' sprechen. Zwischen den 'Bestandteilen' eines stIFS bestehen Wechselwirkungen, und es existieren zumindest immer eine technische Seite (Soft- und Hardware) und eine soziale Seite (Nutzer und Betroffene). Die einzelnen Bestandteile können gegebenenfalls als Subsysteme genauer analysiert werden. Zu den Interessensgruppen zählt im Übrigen auch die der Software-Entwickler, die wir im Folgenden noch genauer betrachten wollen. Unser zentrales Anliegen ist es, die Menschen, die mit der Entwicklung, der Weiterentwicklung, aber auch der Nutzung oder den Auswirkungen von informatischen Artefakten zu tun haben, in den Mittelpunkt des Unterrichts zu rücken und über diesen humanzentrierten Zugang informatische Inhalte zu erschließen.

Soziotechnisches Informatiksystem	
Hardware	Lokaler Rechner, entfernter Rechner, Netzwerkkomponenten
Software	Schichtenarchitektur (z.B. MVC), Komponenten (z.B. Module, Units, ...), Objekte, ...
Interessensgruppen	Anwender, Betroffene, Entwickler

Abbildung 1: Schema eines soziotechnischen Informatiksystems mit seinen identifizierbaren Subsystemen

Unserer Auffassung nach und aus Perspektive der systemorientierten Didaktik ist das Verständnis für die Verzahnung von technischen und sozialen Aspekten ein wesentliches Ziel informatischer Bildung. Nur so werden die Wirkprinzipien von Informatiksystemen deutlich. Software entsteht nicht losgelöst von Kontexten: Ansprüche der Nutzer, Anforderungen durch Spezifika des Einsatzkontextes, allgemeine gesellschaftliche Regeln wie Gesetze und Normen, und viele mehr. In nahezu allen Fällen gibt es diese enge Kopplung zwischen technischer und sozialer Seite, zwischen Software und deren Einsatzbedingungen. Das ist die wesentliche These, auf der die Dekonstruktion aufbaut, und gleichzeitig der Ansatzpunkt für Analyse und Bewertung im Informatikunterricht. Anhand der fachwissenschaftlichen Diskussion in der Softwaretechnik wollen wir diese These nun näher erläutern und aus informatischer Sicht begründen.

1.1 Soziotechnische Informatiksysteme vor dem Hintergrund der Softwaretechnik

Ivar Jacobson beschreibt in einer Forderung an (industrielle) Softwareentwicklungsprozesse implizit Veränderbarkeit als ein wesentliches Qualitätsmerkmal von Software: „All systems will change during their life cycles. This must be kept in mind when systems are developed that are expected to last longer than the first version, i.e., practically all systems. [..] Therefore, an industrial process should focus on system changes“ ([Ja00], S. 33). Dies führt zu inkrementell-interativen Softwareentwicklungsprozessen, wie dem von Jacobson mitentwickelten Unified Process (UP). Grundlage des gesamten Entwicklungsprozesses beim UP bilden Anwendungsfälle (use cases), wobei gilt, dass „use cases are much more than a tool for capturing requirements. They drive the whole development process. Use cases provide major input when finding and specifying classes, subsystems, and iterfaces [..], when finding and specifying test cases, and when planning development iterations and system Integration“ ([JBR99], S. 34f). Anwendungsfälle bilden somit das Bindeglied zwischen Software und ihrem Einsatzkontext. Nicht nur aus didaktischer Perspektive ist es also interessant, Software als Bestandteil eines stIFS zu betrachten, sondern auch aus Sicht der Softwareentwicklung. So können in der Testphase eines Entwicklungsprozesses Anwendungsfälle anhand von Szenarien

durchgespielt werden, um neben der Fehlersuche auch die Brauchbarkeit im Einsatzkontext zu überprüfen (vgl. [JBR99], S. 295ff). Im didaktischen Kontext können auf Anwendungsfällen basierende Szenarien ein exzellentes Mittel sein, ein stIFS im Sinne der o.g. vier GI-Leitlinien zu verstehen und zu analysieren. Unsere sieben Schritte der Dekonstruktion beinhalten daher das Durchspielen von Szenarien.
Ergänzt wird das Durchspielen von Szenarien durch die Inspektion des Quelltextes bzw. von UML-Diagrammen. Als Methode zur Fehlersuche in Software wurden Inspektionen in den 1970er Jahren von Michael Fagan bei IBM vorgeschlagen. Dabei hat er genaue Angaben zum Vorgehen und zur Organisation der Fehlersuche gemacht: An dem Verfahren sind mehre Personen mit unterschiedlichen Rollen beteiligt: der Autor der Software, die eigentlichen Gutachter, ein Leiter/Moderator und ein Schriftführer, um die Ergebnisse zu sichern. Das Verfahren selbst läuft folgendermaßen ab: Zunächst wird den Gutachtern das Produkt vom Autor vorgestellt und erläutert. Danach machen sich die Gutachter eigenständig mit der Software vertraut, sie lesen den Quelltext. Beim zweiten Treffen findet die eigentliche Fehlersuche statt, bei der die Software vom Inspektionsteam analysiert wird und der Schriftführer die Ergebnisse festhält. Danach beseitigt der Autor die festgestellten Fehler, was vom Leiter/Moderator abschließend kontrolliert wird. Inspektionen werden in unterschiedlichen Varianten durchgeführt und gelten als recht erfolgreiche Methode zur Qualitätssicherung von Software, ihr Nutzen wurde recht umfangreich empirisch untersucht ([Pr01], S. 93ff). Dabei hat sich herausgestellt, dass „stärker strukturierte Methoden wie Szenarios oder Perspektiven [..] ad-hoc-Inspektionen oder Checklisten“ ([Pr01], S. 97) überlegen sind. Wenden wir uns also einer solchen, stärker strukturierten Methode zu und sehen uns die Perspektivenbasierte Inspektion (PI) genauer an: Das zu untersuchende Artefakt wird hierbei aus den unterschiedlichen Perspektiven der verschiedenen Interessensgruppen untersucht. Auf diese Weise werden mehr Fehler entdeckt, da durch die Beachtung unterschiedlicher Perspektiven der Bereich möglicher Fehler systematisch und vollständiger als bei anderen Herangehensweisen untersucht wird.
Laitenberger und Atkinson [LA98] schließlich übertragen die PI auf auf das objektorientierte Paradigma und nennen Gründe, weshalb eine Anpassung der PI an die Objektorientierung notwendig ist. Dabei steht im Mittelpunkt der PI nicht mehr ausschließlich der Quelltext, sondern grafische Darstellungen wie z.B. Sequenz- oder Kollaborationsdiagramme. Laitenberger und Atkinson stellen außerdem heraus, dass sich die ursprüngliche PI meist auf Korrektheit als einzigen Qualitätsmaßstab beschränkte. Doch auch die übrigen, verschiedenen Softwarequalitätskriterien (siehe Abb. 2) spielen für die Softwareentwicklung – wie bereits erwähnt – eine immer bedeutendere Rolle. Daher schlagen beide vor, die einzelnen Perspektiven den Rollen der Entwickler und den Qualitätskriterien anzupassen (z.B.: Tester-Perspektive und Fehlerfreiheit).

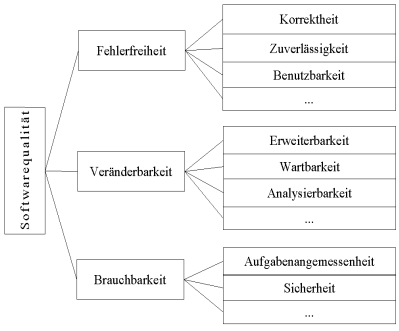

Abbildung 2: Softwarequalitätsmerkmale angelehnt an DIN ISO 9126 (vgl. [Ba00], S.1102f)

2 Dekonstruktion in sieben Schritten

Unser Sieben-Schritte-Schema der Dekonstruktion nun greift die PI insofern auf, als unterschiedliche Perspektiven auf Software nicht nur dem effektiveren Aufdecken von Fehlern und Schwächen dienen können, sondern im Informatikunterricht auch Impulse für fachlich orientierte, analysierende, bewertende und reflektierende Lernprozesse geben können. Multiple Perspektiven erleichtern es zudem, die verschiedenen Bestandteile des durch die Software gegebenen stIFS in ihren Wechselwirkungen zu verstehen – ihre Beachtung entspricht gleichzeitig der Forderung der konstruktivistischen Lerntheorie, durch multiple Perspektiven das Erlernte zu vertiefen: „Gefördert werden kann der Wissenserwerb in diesem Sinne durch verschiedene Konzepte des situierten Lernens: situated cognition, anchored instruction, cognitive flexibility und cognitive apprenticeship [..] Konkret bedeutet dies, dass durch ein Ausgehen von authentischen Aufgaben, die Einbeziehung authentischer Kontexte, die Einnahme multipler Perspektiven und Modelllernen der Wissenserwerb optimiert werden kann“ [Bl02].
Authentische Aufgaben bedeuten vor dem Hintergrund aktueller Softwareentwicklungsprozesse, dass Softwareentwicklung auch im Unterricht als inkrementell-iterativer Prozess anzulegen ist. Die Einbeziehung von Kontexten und unterschiedlichen Perspektiven bedeutet eine Abkehr vom reinen Programmieren, das Softwareentwicklung auf den Prozess des Codierens beschränkt. Aufgrund des Stellenwerts der Veränderbarkeit/Wartbarkeit in der Softwaretechnik umfasst Dekonstruktion neben der Analyse und Bewertung einer vorliegenden Software auch die Änderung bzw. Erweiterung derselben: Im Unterricht wird also eine Iteration eines Softwareentwicklungsprozesses durchgeführt. Auf diese Weise bleibt der Softwareentwicklungsprozess im Mittelpunkt des Unterrichts, und es ergeben sich vielfältige Gelegenheiten für aktives Lernen. Die Schülerinnen und Schüler können durch die eigene Erweiterung der Software einerseits zu einer vertieften Beschäftigung mit dem vorliegenden Design und Funktionsprinzip motiviert werden, andererseits können sie selbst die Anforderungen an Softwareentwicklung erfahren, die sich aus fortgesetzten Iterationen ergeben. Sie erfahren, dass ihre Ideen durch die Vorgaben des Auftrags, durch die Möglichkeiten der Werkzeuge und der Programmiersprache und durch die bereits vorhandenen Softwarestrukturen wesentlich bestimmt werden. Zudem werden sie erleben, dass sie Designentscheidungen treffen müssen, die nicht immer eindeutig im Sinne 'richtig/falsch' entscheidbar sind, manchmal nicht einmal als 'besser/schlechter'. In einer abschließenden Phase wird das erweiterte Gesamtprodukt erneut als stIFS untersucht und bewertet. Hierbei sollen die Schülerinnen und Schüler auch ihr eigenes Vorgehen reflektieren und nicht zuletzt den Bedarf für Zyklen bzw. Iterationen bei der Softwareentwicklung verstehen. Damit ergeben sich folgende drei Phasen der Dekonstruktion, aus denen sich unser Sieben-Schritte-Schema zusammensetzt: (1) *Analyse* der vorliegenden Software, (2) Erweiterung als *Synthese* und (3) *Bewertung und Reflexion*.
Die drei Phasen der Dekonstruktion steigen in den Lernzielniveaus im Sinne der Bloomschen Taxonomie ([JM94], S. 307) an: Vorausgesetzt wird *Wissen* und *Verständnis* objektorientierter Grundkonzepte (Stufen 1+2 bei Bloom). In der ersten Phase der Dekonstruktion muss dieses Wissen anhand einer unbekannten Software *angewendet* werden, um diese zu *analysieren* (Stufen 3+4). In der zweiten Phase der Dekonstruktion wird darauf aufbauend die Software geändert bzw. erweitert. Dabei müssen die Schülerinnen und Schüler selbst Teile des Softwareentwicklungsprozesses durchführen, also verschiedene

Sichtweisen, Kenntnisse und Fähigkeiten einsetzen, um neue Funktionalität zu schaffen (Stufe 5: *Synthese*). In der dritten Phase der Dekonstruktion erfolgt die *Bewertung und Reflexion* des entstandenen Produkts sowie des eigenen Vorgehens bzw. der verwendeten Vorgehensweise (Stufe 6).

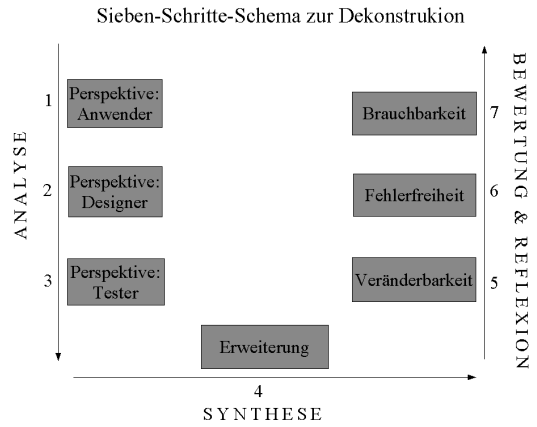

Abbildung 3: Sieben-Schritte-Schema zur Dekonstruktion

Die Einordnung der Lernziele anhand der Bloom'schen Taxonomie macht deutlich, dass Dekonstruktion nicht unbedingt auf den Anfangsunterricht abzielt, sondern eher auf Vertiefung und Reflexion des Gelernten. Gute Lernvoraussetzungen für Dekonstruktion sind daher Vorkenntnisse, wie sie etwa anhand des Phasenmodells aus dem life-Projekt[1] vermittelt werden können [SN02]: Dazu gehören Grundkenntnisse in Objektorientierung und im objektorientierten Modellieren. Weiterhin ist wünschenswert, dass die Entwicklungswerkzeuge, die Programmiersprache, Phasen der Softwareentwicklung und entsprechende UML-Diagrammtypen bekannt sind.

3 Die sieben Schritte im Detail

Die drei Phasen des obigen Sieben-Schritte-Schemas der Dekonstruktion (siehe Abb. 3) schließlich teilen sich in mehrere Schritte, die wir im Folgenden an einem Beispiel näher erläutern werden, um so den Ablauf einer Unterrichtseinheit nach obigem Schema deutlich zu machen.

Für unser konkretes Beispiel, eine Unterrichtseinheit nach dem Sieben-Schritte-Schema der Dekonstruktion, nehmen wir also als konkrete Vorkenntnisse an: Die Schülerinnen und Schüler kennen den Unterschied zwischen Klasse und Objekt, Beziehungen zwischen Klassen, Anwendungsfälle, CRC-Modellierung, UML-Klassendiagramme, Grundzüge des Softwareentwicklungsprozesses (Analyse, Design, Implementation), sie haben bereits eigene, kleine Anwendungen erstellt. Wir zeigen anhand der Verknüpfung des Themas Datenschutz mit der Thematik Datenspeicherung, Serialisierung und Streams wie mit dem Schema zur Dekonstruktion soziale und technische Aspekte verbunden werden können. Als zu dekonstruierende Software liegt dem Beispiel die Software Schulkiosk[2], ein kleines Warenwirtschaftssystem, zugrunde.

Zu beachten ist im Folgenden, dass die Inhalte der sieben Schritte natürlich von der zu dekonstruierenden Software, den Vorkenntnissen der Lerngruppe und den jeweiligen Lernzielen bestimmt werden, was bedeutet, dass einige Schritte länger, andere kürzer

[1]Siehe: www.life.upb.de

[2]Die Software Schulkiosk wurde als eine didaktische Software auf die Methodik der Dekonstruktion hin in verschieden Ausbaustufen entwickelt (siehe auch [HMS99]).

ausfallen können. Die einzelnen Schritte ordnen sich in drei Phasen, die wir nun anhand des oben umrissenen Beispiels erläutern.

3.1 Analyse

Der erste Schritt der Analyse beginnt mit der **Anwender-Perspektive (1)**[3]: Die Lehrperson stellt die Software (in unserem Beispiel die Software Schulkiosk) kurz vor und gibt einige Hinweise zu deren Benutzung. Aufgabe der Schülerinnen und Schüler ist es anschließend, über das Erkunden/Benutzen der Software Anwendungsfälle herauszufinden, welche die Software abdeckt, aber auch, ob aus ihrer Sicht wichtige Anwendungsfälle fehlen. Für jeden der gefundenen Anwendungsfälle sollen dann passende Szenarien überlegt werden. Hierbei kann die Software bereits an dieser Stelle zumindest ansatzweise als Teil eines stIFS begriffen werden.
Weiterhin sollen sie versuchen, anhand dieses Erkundungsvorgangs auf die verwendeten Fachklassen und deren Beziehungen zu schließen (dies geht allerdings über die Anwender-Perspektive hinaus und ragt in die Designer-Perspektive (2) hinein). So entwickeln die Schülerinnen und Schüler bereits eine konkretere, individuelle Vorstellung, wie das Fachklassen-Modell der Software aussehen könnte. Ihr Modell wird im nächsten Schritt unter der Design-Perspektive mit dem tatsächlichen Modell konfrontiert. Zunächst aber ein Beispiel:

> Arbeitsauftrag: Ein Manager bestellt für das Erweitern des Sortiments einen Posten einer neuen Ware, die noch nicht im Sortiment des Kiosks enthalten ist. Wie kann dieses Szenario bearbeitet werden?
> Mögliches Ergebnis: Bei Lieferung legt der Manager in der Software Schulkiosk zunächst einen neuen Warenposten an, indem er eine Bezeichnung, eine Nummer, einen Verkaufspreis und eine Warnmenge festlegt. Anschließend registriert er die gelieferte Ware, indem er Anzahl, Einkaufspreis und ggf. ein Haltbarkeitsdatum in die Software eingibt.

Abbildung 4: Beispiel für ein Szenario: Manager erweitert Sortiment

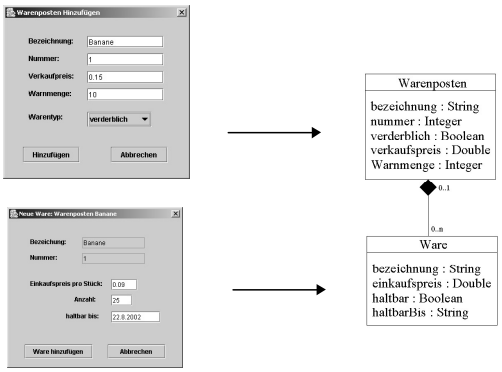

Abbildung 5: Anhand der Benutzung und der Benutzungsschnittstelle (links) entwickeln die Schülerinnen und Schüler eine Vorstellung des Klassen-Modells (rechts).

Das Erweitern des Sortiments (vgl. Abb. 4) erfordert das Hinzufügen eines neuen Warenpostens und anschließend das Auffüllen dieses Warenpostens mit neuer Ware. Daraus können die Schülerinnen und Schüler für das Fachklassendiagramm der Software Schulkiosk folgern, dass es möglicherweise eine Klasse *Warenposten* und eine Klasse *Ware* gibt, zwischen denen eine 1-n-Aggregationsbeziehung besteht. Die entsprechenden Attribute der Klassen *Warenposten* und *Ware* können z.B. die in Abb. 5 angegebenen sein.
Die **Designer-Perspektive (2)** baut auf den Vorstellungen der Schülerinnen und Schüler aus der Anwender-Perspektive auf und wird eingeleitet mit einer Analyse des tatsäch-

[3]Die fettgedruckten und nummerierten Schritte entsprechen den Schritten des Schemas (Abbildung 3).

lichen Klassendiagramms der Fachklassen (d.h. ohne die Klassen der Benutzungsoberfläche). Im Gegensatz zur PI geschieht das nicht frontal durch Erläuterung, sondern indem die Schülerinnen und Schüler sich mit Hilfe geeigneter Arbeitsaufträge selbst in das Design einarbeiten. UML-Werkzeuge[4] unterstützen die Schülerinnen und Schüler dabei, denn sie erlauben es, aus vorhandenem Quelltext grafische Repräsentationen und umgekehrt aus den grafischen Darstellungen Quelltext zu erzeugen.

Als Erstes werden die Schülerinnen und Schüler aufgefordert, sich im Fachmodell zu orientieren, Übersicht zu gewinnen und herzustellen. Dazu sollen sie zunächst unwesentliche Details ausblenden; etwa indem sie (in unserem Beispiel in Fujaba) die einzelnen Klassen des Fachklassen-Modells 'zusammenklappen', d.h. die Attribute und Methoden ausblenden. Anschließend sollen sie die Klassen sinnvoll anordnen, wobei zusammengehörige Klassen nahe beieinander liegen, sich die Beziehungs-Kanten also möglichst wenig/nicht kreuzen sollen. Auch dies dient der Übersichtlichkeit und hilft, das Design der vorliegenden Software zu verstehen.

Die nächste Aufgabe der Schülerinnen und Schüler ist, sinnvolle Subsysteme zu finden, in welche sie das ihnen vorliegende Fachklassen-Modell zerlegen können. Dies könnte in unserem Beispiel anhand der Rollenverteilung der beiden, während der Analyse der Anwendungsfälle im ersten Schritt gefundenen Anwender-Gruppen 'Manager' und 'Kassierer' geschehen, aber auch anhand der Unterscheidung von 'Daten-Klassen' und 'Steuerungs-Klassen'. In jedem Fall können sie sich dabei des Werkzeugs Fujaba bedienen, das ihnen die Möglichkeit bietet, ohne Änderungen am Modell für jedes der identifizierten Subsysteme eigene Klassendiagramme anzulegen, welche nur die für die jeweilige Sicht interessanten Klassen und Beziehungen enthalten.

Nach der Erkundung des Fachklassen-Modells erfolgt nun die Gegenüberstellung mit den Designvorstellungen der Schülerinnen und Schüler, die sie während der Anwender-Perspektive erlangt haben.

Greifen wir dazu nochmal aus dem ersten Schritt (der Anwender-Perspektive (1)) das Beispiel mit den Klassen *Warenposten* und *Ware* auf: Wir nahmen an, dass die Schülerinnen und Schüler der Klasse *Ware* neben einem Attribut *haltbarkeitsdatum* unter anderem das Attribut *haltbar* vom Typ Boolean zugeordnet hatten (Abb. 5).

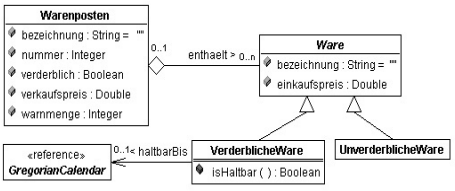

Abbildung 6: Das Fachklassendiagramm der Software weicht von den Schülervorstellungen (siehe Abb. 5) aus der Anwender-Perspektive (1) ab.

Im Fachklassendiagramm jedoch (vgl. Abb. 6) hat die (abstrakte) Klasse *Ware* stattdessen die beiden Unterklassen *VerderblicheWare* und *UnverderblicheWare*. Die Schülerinnen und Schüler sollen nun überlegen, welche Gründe ausschlaggebend für das obige Design gewesen sein können[5]. In der Diskussion sollten Lesbarkeit und Erweiterbarkeit in irgendeiner Form genannt und im Unterricht als Qualitätskriterien für das Design von Software aufgegriffen werden. Dadurch werden Unterschiede in den Entwürfen bzw. Vorstellungen wahrgenommen, im Einzelnen identifiziert und aus einer Design-Perspektive besprochen. Die Schülerinnen und Schüler lernen, dass es unterschiedliche Designmöglichkeiten geben

[4]Im Folgenden beziehen wir uns der Einfachheit halber auf ein Werkzeug, nämlich Fujaba, da wir es im life-Projekt erfolgreich einsetzen.

[5]Eine mögliche Begründung findet man u.a. bei [Fo00], S.228. Refaktorisierung: 'Typenschlüssel durch Unterklasse ersetzen'. Dort auch viele andere Verbesserungsvorschläge für objektorientierte Designs.

kann und vertiefen ihre Argumentations- und Bewertungsmuster. Die Entwicklung eines Softwaredesigns wird ihnen so als ein (kommunikativer) Entscheidungsprozess deutlich.
In der anschließenden **Tester-Perspektive (3)** stehen dynamische Aspekte der Software und die Implementation im Mittelpunkt. Die Schülerinnen und Schüler testen, wie die Software in verschiedenen Szenarien funktioniert. Um die Funktionsweise zu erklären, wird in diesem Schritt die Implementation untersucht. Im Sinne des Testens werden auch unvorhergesehene und ungewöhnliche Situationen durchgespielt (vgl. Abb. 7):

Arbeitsauftrag: Die Datei auf der Festplatte, die die Warenposten speichert, ist beschädigt worden. Wie verhält sich die Software in diesem Szenario? Und warum? Ergebnis: Die Schülerinnen und Schüler müssen neben der Anwendung auch den Quelltext untersuchen und lernen dabei die Mechanismen zum Laden/Speichern kennen.

Abbildung 7: Beispiel: Methoden zum Speichern und Laden von Objekten testen

Ein Debugger zur schrittweisen Ausführung von Programmen kann hier sinnvolle Dienste leisten. Aus dem dabei beobachteten Verhalten und dem Quelltext an sich lassen sich z.B. Sequenzdiagramme zur Ergebnissicherung ableiten.
Die Tester-Perspektive kann also gut dazu dienen, neue Konzepte und deren Implementation einzuführen, in unserer konkreten Unterrichtseinheit ist dies z.B. das Speichern mittels Serialisierung.

3.2 Synthese

Die Schülerinnen und Schüler untersuchen als Übergang von der Tester-Perspektive (3) zur **Erweiterung (4)** ein – ggf. vom Lehrer vorzugebendes – Szenario, das aufgrund fehlender Funktionalität nicht durchgespielt werden kann. Sie erkennen, dass die Software erweitert werden muss.
In unserem Beispiel entspricht dies einem Szenario, welches das Speichern von personenbezogenen Daten erforderlich macht (siehe Abb. 8):
Die Erweiterung baut auf der Analyse des Designs in Schritt (2) und der Analyse der Implementation in Schritt (3) auf. In unserem Beispiel finden die Schülerinnen und Schüler in der Software Schulkiosk bereits einen Großteil der neuen Funktionalität vor. Sie fügen lediglich im Fachklassenmodell die Funktionalität zum Speichern und Einladen hinzu. Dazu müssen sie die entsprechenden Anknüpfpunkte im vorhandenen Design fin-

Arbeitsauftrag: Der Manager will wissen, wie viele Schüler und was diese in der letzten Woche bargeldlos im Schulkiosk eingekauft haben. Ergebnis: Die Schülerinnen und Schüler stellen fest, dass die entsprechenden Daten zwar erzeugt, aber nicht gespeichert werden. Es wird deutlich, dass diese Daten gespeichert und eingelesen werden müssen.

Abbildung 8: Beispiel eines Erweiterungsszenarios

den und wissen, wie die entsprechenden Methoden realisiert werden können.
Bei mehr zu Verfügung stehender Zeit kann die Synthese-Phase jedoch eher im Sinne eines Softwareprojekts angelegt werden, indem weniger Anknüpfpunkte und weniger Funktionalität vorgegeben werden. Die Erweiterung der Software sollte in jedem Fall das Kombinieren und Anwenden der in den drei Schritten der Analyse-Phase erworbenen Kenntnisse erfordern.

3.3 Bewertung und Reflexion

In der Bewertungsphase werden die in der Analysephase eingenommenen Perspektiven erneut aufgegriffen und auf einer abstrakteren Ebene reflektiert. Es geht darum, Einschätzungen und Bewertungen vorzunehmen sowie Kriterien zur Bewertung von Software entweder selbst zu erarbeiten oder heranzuziehen und anzuwenden.
Zunächst befassen sich die Schülerinnen und Schüler mit der Beurteilung und Qualität des Designs und damit mit dem Aspekt der **Veränderbarkeit (5)** von Software. Die Schülerinnen und Schüler überlegen, ob die Erweiterung, die sie in der Synthesephase durchgeführt haben, schwierig war. Sie erörtern anschließend die Schwierigkeiten, die Erweiterungen prinzipiell mit sich bringen und welche Anforderungen an die Qualität von Software sich daraus ergeben.
Das Besprechen der **Fehlerfreiheit (6)** anhand der Erweiterung macht deutlich, dass zu ihrer Sicherstellung eigene Arbeitsschritte erforderlich sind. Dies liegt zum einen daran, dass neue Anwendungsfälle hinzugekommen sind, die getestet werden müssen. Zum anderen kann es Wechselwirkungen mit bestehender Funktionalität geben, weshalb auch bereits getestete und als fehlerfrei geltende Teile der Software nochmals zu untersuchen sind.
In unserem Beispiel betrifft dies z.B. die Funktion, einen getätigten Kauf zu stornieren – je nachdem wie und wann die Speicherung der nutzerbezogenen Einkaufsdaten erfolgt, kann es hierbei zu Inkonsistenzen kommen. Die Schülerinnen und Schüler können sich in Erinnerung an die Tester-Perspektive(3) während der Analysephase sicherlich eine Reihe von (Ausnahme-)Situationen vorstellen, die zu Fehlfunktionen der Software führen können. Soll die Software auf jede dieser Situationen reagieren können, müssten an vielen verschiedenen Stellen in der Software entsprechende Abfragen und Fehlerbehandlungsroutinen eingearbeitet werden, diese aber würden das Qualitätskriterium der Veränderbarkeit beeinträchtigen. Der Mechanismus zur Ausnahmebehandlung wird als Mittel erkannt, um Fehlerbehandlung vom normalen Programmfluss zu trennen und so Lesbarkeit und Wartbarkeit zu fördern.
Die Bewertung der **Brauchbarkeit (7)** umfasst verschiedene Aspekte: Wie integriert sich die neue Funktionalität in die vorhandenen Arbeitsabläufe? Wie ist sie softwareergonomisch in die Benutzungsschnittstelle eingearbeitet?
In unserem Beispiel wäre zu überlegen, ob die Trennung in die Nutzergruppen 'Manager' und 'Kassierer' notwendig ist und wer die neue Funktionalität benutzen darf. Diese Diskussion wird Datenschutzaspekte berücksichtigen müssen. Das belegt beispielsweise das 'school-card'-Projekt eines Bielefelder Gymnasiums, in dem ein Informatik-Leistungskurs ein bargeldloses Zahlungssystem für die Schule entwickeln und einführen wollte. Das System speicherte personenbezogene Daten. In der Presse wurde das Projekt diskutiert und schließlich sogar als abschreckendes Beispiel für besonders unverantwortlichen Umgang mit dem Thema Datenschutz mit dem bundesweit verliehenen BigBrotherAward 'ausgezeichnet' [BB01]. Da bei der Dekonstruktion generell verschiedene Perspektiven beachtet worden sind, wirkt die Thematisierung von Datenschutzaspekten auf die Schülerinnen und Schüler einfach als eine weitere Perspektive. Dies wird im Übrigen dadurch unterstützt, dass in der Unterrichtseinheit immer wieder die Software als Bestandteil eines stIFS thematisiert worden ist.

4 Schlussbemerkung

Die gewählte Reihenfolge der einzelnen Schritte des Sieben-Schritte-Schemas (insbesondere der Schritte (2) und (3) sowie (5) und (6)) ist variabel und hängt von den gewählten Inhalten ab. In unserem Beispiel orientiert sich die Reihenfolge an den Schritten der Softwareentwicklung. Sollte die Erweiterung in der Synthesephase jedoch umfangreicher (z.B. angelegt als Softwareprojekt) sein, plädieren wir dafür, einzelne Bewertungs- und Reflexionsaspekte in die Synthesephase zu integrieren, damit sie im Sinne konstruktivistischer Lerntheorie situativ an die Erfahrungen und Handlungen der Schülerinnen und Schüler gebunden werden können. In jedem Fall kann am Ende der Reflexions- und Bewertungsphase, wenn unter dem Aspekt Brauchbarkeit beurteilt wurde, wie die Software in ein stIFS integriert ist, deutlich werden, dass eine weitere Veränderung bzw. Erweiterung der Software nötig oder wünschenswert ist. Dies kann dazu führen, den Softwareentwicklungsprozess insgesamt und damit verbunden insbesondere die Gründe für ein inkrementell-iteratives Vorgehen zu besprechen.
Insgesamt führt das Sieben-Schritte-Schema der Dekonstruktion zur Beachtung unterschiedlicher Perspektiven. Wesentlich für die Methode der Dekonstruktion ist hierbei die Rückbindung des Handelns der Schülerinnen und Schüler an reflektierende und bewertende Aktivitäten. Auf diese Weise soll ihnen über die Analyse, Erweiterung und Bewertung einer Software deutlich werden, was ein stIFS ist - dieses kann auch explizit thematisiert werden. Dekonstruktion geht somit über einen reinen Programmierkurs hinaus, indem u.a. durch die Betrachtung von Softwarequalitätsmerkmalen der Bezug zum Einsatzkontext hergestellt wird: Die Schülerinnen und Schüler programmieren nicht mehr nur Software, sondern erfahren die Gestaltung von Informatiksystemen.

Literatur

[Ba00] Balzert, Helmut: Lehrbuch der Software-Technik. Software-Entwicklung. Spektrum, 2000, 2. Aufl.

[BB01] BigBrotherAward (http://www.big-brother-award.de/2001/.local/index.html)

[Bl02] Blömeke, Sigrid: Handlungsmuster von Lehrerinnen und Lehrern beim Einsatz neuer Medien im Unterricht. In: Aufenanger, Stefan [u.a.] (Hrsg.): Jahrbuch Medienpädagogik 3. Opladen, 2002.

[Fo00] Fowler, Martin: Refactoring. Addison-Wesley, 2000.

[GI00] Gesellschaft für Informatik: Empfehlungen der Gesellschaft fuer Informatik e.V. fuer ein Gesamtkonzept zur informatischen Bildung an allgemein bildenden Schulen. In: Informatik Spektrum 23, Heft 6, 2000, S. 378-382.

[HMS99] Hampel, T.; Magenheim, J.; Schulte, C.: Dekonstruktion von Informatiksystemen als Unterrichtsmethode. In (Schwill, A. Hrsg.): Informatik und Schule. Springer, 1999, S. 149-164.

[Ja00] Jacobson, Ivar: The Road to the Unified Software Development Process. Cambridge Univ. Press [u.a.], 2000.

[JBR99] Jacobson, I.; Booch, G.; Rumbough, J.: The Unified Software Development Process. Addison-Wesley, 1999.

[JM94] Jank, Werner; Meyer, Hilbert: Didaktische Modelle. Cornelsen Scriptor, 1994, 3. Aufl.

[LA98] Laitenberger, O.; Atkinson, C.: Generalizing Perspective-based Inspection to handle Object-Oriented Development Artifacts. IESE-Report 057.98/E, December 1998

[Ma00] Magenheim, J.: Informatiksystem und Dekonstruktion als didaktische Kategorien. (http://ddi.uni-paderborn.de/didaktik/Veroeffentlichungen)

[Pr01] Prechelt, Lutz: Kontrollierte Experimente in der Softwaretechnik. Springer, 2001.

[SN02] Schulte, Carsten; Niere, Jörg: Thinking in Object Structures: Teaching Modelling in Secondary Schools. In: Sixth Workshop on Pedagogies and Tools for Learning Object Oriented Concepts, ECOOP, Malaga, Spanien. 2002. (http://ddi.uni-paderborn.de/didaktik/Veroeffentlichungen)