Using a Framework to Teach OOT to Beginners

Birgit Demuth
Heinrich Hussmann
Steffen Zschaler
Department of Computer Science
Dresden University of Technology
01062 Dresden, Germany
{demuth,hussmann,zschaler}@inf.tu-dresden.de

Lothar Schmitz
Department of Computer Science
University of the Federal Armed Forces Munich
85577 Neubiberg, Germany
lothar@informatik.unibw-muenchen.de

ABSTRACT

We report on experience from teaching OO technology to undergraduate students. Before they can successfully tackle the projects we expect them to do in their second year they have to successfully shift to the OO paradigm, pick up a working knowledge of some OO language, learn and practice OOA and OOD, and get used to advanced ideas like patterns and frameworks.

In order to relieve this heavy burden somewhat, we provided a framework as a common base for the projects. That way, the students are given an architecture which they have to adapt to their specific task instead of doing all the design by themselves. We also believe that this policy closely resembles the way beginners are integrated into ongoing projects in practice.

We describe the Java framework we used, the preparations for and the organization of the project course, educators' and students' experience and some ideas for developing this approach further.

Keywords

Teaching, object-oriented technology, framework, Java, software engineering course, patterns, UML

INTRODUCTION

At Dresden University of Technology, computer science undergraduates are taught OO software engineering very early: An introductory course in the second semester is followed by a project course in the third, where student teams are given individual tasks. This provides valuable experience for later programming assignments and advanced courses.

Since OO technology is evolving rapidly the material and organization of both courses have to be adapted frequently. Here, we describe our current approach which we have found very useful and which might be applied similarly by other organizations as well.

In order to become a proficient OO software engineer, you have to climb several rungs of a very demanding technical ladder:

- First, learn to solve problems by building small universes of interacting objects. Those with a strong background in procedural programming may have to unlearn their previous algorithm-centered approach before they can get used to the new paradigm.
- Second, adopt the habit of reusing existing classes instead of inventing new ones. This requires you to know where to look for reusable components, i.e. you must understand the scope and structure of the class libraries you are using. Getting to know a large class library in sufficient detail will take quite some time.
- OO technology has much more to offer than just OO languages and libraries: While beginners have to be taught the importance of a formal software life cycle and of requirements analysis, in particular, more experienced developers will immediately appreciate the benefits of incremental development and prototyping that are typical of OO methodology (see e.g. [4] and [5]).
- Finally, there are patterns and frameworks (cf. [2]). *Patterns* describe proven solutions: when, where and how to apply them. Pattern names are carefully chosen for ease of communication between developers. *Frameworks* are application skeletons that can be turned into complete applications by providing parameters and/or subclasses of the framework's generic classes.

The higher you climb on this ladder, the more leverage you will gain for developing your own applications.

A strong motivation for our approach stems from the following observation: In academic as well as industrial settings beginners will often join projects which are already well in progress. Finding out enough about the project's structure to be able to do your job is similar to learning how to apply an application framework. A practical course based on one common framework therefore offers several advantages:

- For the organizers it is easy to define a number of similar projects and to scale the projects' complexity from moderate to reasonably hard.
- Since they are based on the same framework, all the different tasks are still comparable. If competition is desirable you can assign identical tasks to different teams.
- As pointed out above, learning your way around a given framework corresponds to a characteristic professional activity.
- Beginners get a chance to learn good design by example: Frameworks by definition are designed for change. Therefore, they typically exhibit patterns that increase flexibility.

The rest of this paper is organized as follows: First we describe the domain, architecture, and the adaptation interface of the `SalesPoint` framework. The next Section relates how we prepared the students for their project work. Then we outline the project organization and briefly report on the results from a detailed student's questionnaire. The last section describes how our concept has evolved over time and indicates some future extensions.

THE FRAMEWORK

The `SalesPoint` framework the students were given supports the development of point of sale simulations ranging from simple vending machines to big department stores. Typical applications include an exchange office where you can obtain foreign currency, a post office offering stamps and a well-defined set of services, a drugstore, or a video shop.

All applications from this domain share the following characteristics:

- A point of sale offers articles from some fixed *catalogue*. For each article, the catalogue has an entry giving its name, price, and other relevant properties. A *stock* is a bag of articles from the catalogue. Examples of stocks are: the goods on an order form, the articles contained in a vending machine, in the shelves of a store, or in a customer's shopping basket.
- *Money* fits into this terminology as a special case: Here the catalogue is called a *currency*. It describes the set of valid bank notes and coins and their values. The contents of your purse or those of a cash register are *money stocks*.
- The main purpose of a point of sale is to sell, buy, or trade articles. Such *transactions* are considered atomic, i.e. they are performed completely or have no effect.
- In a point of sale there are background activities not visible to customers: Goods have to be ordered from wholesalers; revisions are due in regular intervals; the catalogue has to be adapted to customers' demands by removing slow-moving articles and adding new ones instead.

Accordingly, the `SalesPoint` framework supports the development of point of sale simulations by providing:

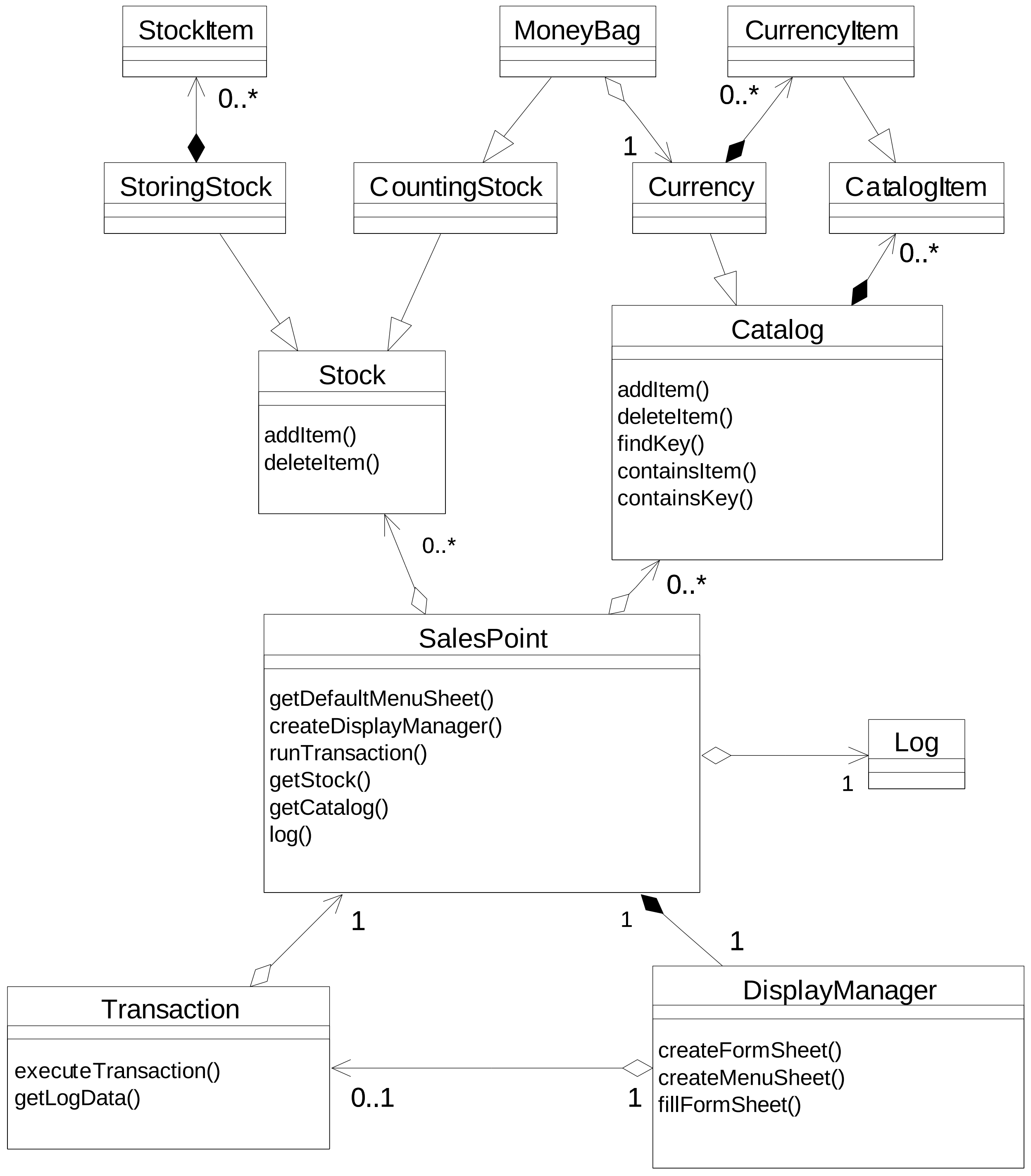
- base classes for catalogues, stocks, currencies, and money stocks;
- generic form and menu classes for user interaction (two separate layers: the *logical layer* which is part of the framework's adaptation interface, and which is based on the underlying hidden *physical layer*);
- transaction support including roll-back and logging mechanisms;
- different algorithms for the standard problem of building a stock for a given value (needed e.g. for returning change money or at a post office for assembling a collection of stamps with a given total value).

The figure below contains the UML static class structure of the `SalesPoint` framework: it shows most of the classes, their associations, and the methods that are relevant to the framework's users.

Like other frameworks, `SalesPoint` is adapted to its users' needs in several ways:

- by supplying specialized subclasses; e.g. menu sheets are easily adapted using Java's *inner classes*;
- by providing *hook methods*; e.g. the roll-back mechanism is adapted with three hook methods: one for saving the state before the transaction is started, one for restoring that state, and one for releasing files and other resources;
- by providing *parameters*; e.g. when creating a new stock object one of the constructor's parameters describes which catalogue to use, another chooses one of the algorithms for building stocks with a given value.

As indicated above, the `SalesPoint` framework is implemented in Java (see e.g. [1]). In order to make it flexible, some of the patterns from the [2] collection of design patterns were applied: e.g. the *Abstract Factory* pattern for creating catalogues, the *Bridge* pattern for choosing between different implementations of stocks and catalogues, and the *Strategy* pattern for varying among the different stock-building algorithms.



The students were given the framework's Java class files with a *javadoc*-generated documentation and a *tutorial* describing in detail how the framework can be used for building a simple ticket vending machine. The tutorial covers all aspects of the framework's application interface for use as a *black box*, and also explains its structure for *white box* use.

CLIMBING THE LADDER

For qualifying students in OO technology, several approaches are in use ranging from leaving it all to the students themselves (relying on and supporting their self-government) to teaching all the details in a bottom-up fashion. Dealing with beginners, we opted for the second alternative. We aimed to provide them with:

a thorough understanding of OO notions and terminology including familiarity with the basics of UML notation;

- an appreciation of OOA/OOD methodology and of requirements analysis, in particular;
- sufficient practical experience with the Java programming language as required for the intended projects along with a working knowledge of core parts of the Java class library;
- a good idea of patterns, frameworks, and what they are good for.

We decided to switch from C++ to Java for well-known reasons: Java is a more pure OO language than the hybrid C++. It has useful new concepts like *interfaces*, includes automatic garbage collection and it avoids some of the error-prone features of C++ such as pointer arithmetic and multiple inheritance. Therefore, it seemed (and has proven) better suited as a first OO language.

Lectures were supplied with lab exercises and homework for hands-on experience. Our aims and the bottom-up approach resulted in the following course outline (examples, exercises and homework in *italics*):

Unit A

Problems addressed by software engineering. OOT currently most popular approach to solving them. *Introductory Java programming: String handling.*

Unit B

Basic OO notions: objects, classes, inheritance, polymorphism, etc. Small subset of UML static diagram notation. *Modeling geometric objects in Java.*

Unit C

Learning from the Java library developers by looking at the Vector and Hashtable Container classes and their Enumerations. *Telephone dictionary example; AVL trees as an alternative to Hashtables.*

Unit D

Solving problems by evenly distributing intelligence over a set of cooperating classes: *Grammar example*. Using exceptions to make programs robust: *File handling example from [1]*

Unit E

Introduction to OO analysis using CRC cards [4], responsibility-driven design for design, Use Cases and UML sequence diagrams. *Dresden ticket vending machine.*.

Unit F

Architectural, design and programming patterns with *examples from the Container classes*.

Unit G

Parallel threads in Java. Applets and AWT as examples of frameworks. *Programming a graphical user interface for a file browser*

Unit H

Putting it all together: *The Dashboard example*.

While successful on the whole, one result of this course remained unsatisfactory: students with previous hacking experience tend to overestimate their abilities and to underestimate the problems of systematic software development. As a consequence, their motivation to engage themselves actively is poor.

A notable exception were the CRC card sessions we organized for student teams of about five persons each. Most students overcame their inhibitions and began to discuss their analysis and design problems freely and vividly. In future we shall, therefore, replace the pure bottom-up approach by a mixed strategy where CRC cards and the basics of OO analysis and design are presented before the rest of the material.

USING THE FRAMEWORK

Because of the large number (116) of students that participated in the project course, a rather formal mode of organization was needed. All information was distributed via WWW: the framework, its description, the tutorial, and the project specifications. The students were encouraged to present their solutions on HTML pages in the same way.

In the beginning, a lecture on teamwork organization and related problems was supplied. Then the students were asked to form teams of about five persons and to adopt a chief programmer team organization, i.e. to assign chief, assistant, secretary and developers' roles to the team members. Most of the resulting 22 teams were coached by two senior students who in turn were supervised by one assistant (first author of this paper). Technical questions and requests for framework correction or extension were handled by the student (fourth author) who had developed the framework and the tutorial according to the specifications of the third author.

A rather rigid time table was prescribed for project work. At the end of each phase, results (documents, programs) had to be presented to the tutors. Final delivery included a formal oral presentation of about half an hour per team where the main results including the working program had to be shown and questions to be answered. The time table was as follows:

Requirements definition	... 1 week
OO Analysis	... 2 weeks
OO Design	... 2 weeks
Implementation and Test	... 4 weeks
Maintenance	... 4 weeks

Alternatively, incremental development with several development cycles was allowed. This approach was adopted by most teams. The average number of design cycles was three. During the requirements definition and OO analysis phases the students also had to study the framework and its tutorial. Maintenance included removal of bugs and satisfying some minor client's wishes.

The overall results of the project course were very encouraging: 103 of 116 students (21 teams of 22) finished successfully (there were a number of drop-outs from successful teams).

During the whole process we had a lot of feedback: from the tutors, some students' questions, intermediate documents, final presentations and a detailed questionnaire we requested from the students. We learned that:

- studying the framework took more time than we had expected (about 25 \% of the whole effort); in retrospect we feel this justified since it covers a good deal of what would otherwise have been part of the design phase;
- students rated the tutorial and the on-line support rather high;
- the time table was realistic, given the students' tendency to postpone work towards the end;
- on an average, the students spent about 8 hours per week on their projects; there also seemed to be some backlog in the form of missing OO and Java knowledge from the introductory course;
- students liked the tasks they were given; some teams even tried to find out *real* clients' requirements by doing field studies;
- students rated their own achievements rather high; for them, team work experience was novel and important;
- in the tutors' opinion, most students performed rather well, but there still seemed to be some who had hacked their way without a true appreciation of OO technology; on the positive side, the framework proved practicable and accommodated all kinds of students' approaches: everyone felt they had learnt a lot.

At [6] all the material for the project course is assembled, most of it in English. If you can read German, you can also have a look at the long version of the tutorial and at the students' results.

AN EVOLVING PROCESS

Before arriving at the current course organization, we had tried out different approaches at different places as described below. Some hints on how to further evolve it are given in the end.

Learning a new programming paradigm without some practical experience seems impossible. Therefore, the third author in his Munich courses on OO technology replaced conventional written exams by individual homework projects: students were allowed to work in teams. After two weeks they had to present their design, after two more weeks to demonstrate a working (Smalltalk) prototype in class. In those courses, the primary focus was on OO programming techniques and on reuse, in particular. Students enjoyed the projects and, therefore, worked hard to achieve these goals.

While at first many different and unrelated tasks were given to the students, a few years later a more ambitious approach was chosen: Different tasks from one common domain were handed out to the students. After some teams had completed their prototypes, other teams tried to distill their experiences and solutions into a framework. Then a third group of teams were to do reimplementations of the original prototypes using this framework.

Both times we followed this new approach we were only partially successful: Still, the students with much enthusiasm finished their prototypes. But the task of abstracting the common parts into a framework and proving its usefulness by doing reimplementations turned out to be too hard. Within the short time available and with their limited experience, students produced frameworks that were too immature to be applied successfully. Another problem with the new approach was that projects had to be started before all the relevant techniques had been taught.

In the current organization, the above problems are avoided: the latter one by decoupling the introduction to OO technology and the project course, the former one by providing the students with a well-prepared framework in advance. Remaining minor problems appear to be organizational ones that we shall try to remedy as indicated.

At Dresden, previous courses had been based on OMT, the Booch method, and C++ programming language. In the lectures, emphasis was on OO analysis and design. An elementary introduction to C++ was given during lab exercises. Beyond that, students were on their own to find out about the C++ constructs and libraries they needed to do their projects. By switching to UML and Java we not only modernized our curriculum but also had the chance to teach OO programming in sufficient detail.

The existing framework can (and will) be improved in many ways. Possible major extensions include:

- using the Java JDBC interface to make catalogues and stocks persistent by storing them in a relational data base;
- using the Java RMI or CORBA interface to obtain families of cooperating `SalesPoint` applications distributed over the net; e.g. competing shops might buy their goods from different wholesalers who in turn obtain them from different factories; here, shops, wholesalers and factories all share the `SalesPoint` characteristics.

Also, our approach should carry over easily to frameworks in other domains and there provide the same advantages for beginners: *realistic professional activity* is simulated and *guidance in the form of a framework* is offered.

ACKNOWLEDGMENTS

We thank our students Axel Grossmann and Jens Stuendel working as tutors, who, through their efforts and contributions, made the organization and execution of the described software engineering course successful.

REFERENCES

1. Flanagan, D. *Java in a Nutshell*. O'Reilly & Associates, Sebastopol, 1996. Also extended second edition 1997.
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns - Microarchitectures for Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1994.
3. Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language, Documentation Set 1.0*. Rational Software Corporation, Santa Clara, 1997.
4. Wilkinson, N. *Using CRC Cards. An Informal Approach to Object-Oriented Development.* SIGS Publications, New York, 1995.
5. Wirfs-Brock, R., Wilkerson, B., and Wiener, L. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, 1990.
6. Zschaler, St. *The SalesPoint Framework Homepage*. http://www.inf.tu-dresden.de/~sz9/SWTProject/ver05