# Good Bye to Hello World — An Interactive Approach to Java

Ulrike JAEGER; Jochen SCHMIDT
University of Applied Science, D-74081 Heilbronn, Germany, Max Planck Str. 39

{ulrike.jaeger; jochen.schmidt}@fh-heilbronn.de

URL: www.fh-heilbronn.de

KEYWORDS: *education, software-engineering, programming, interactive*

ABSTRACT: *Many programming courses start with a small isolated "Hello World" example that gradually extends, as the course discusses the various programming aspects, but still remains an isolated monolithic program with little chance for structure and design.*

*In modern applications, programming is rather a way of integrating several ready made solutions, add some flavor of own invention and build a powerful, distributed, and highly interactive solution.*

*This is especially true for software engineering as a discipline. In Heilbronn, we adapted Lynn A. Stein's "Interactive Programming in Java" which is a revolution in teaching programming for undergraduates and implemented at the MIT.*

*Our introductory course to programming spans a whole year and develops a a simple and powerful application example — which is interactive from the start — towards a comfortable fully grown simulation that includes database access and a graphical user interface. The course bases on carefully designed code examples that are revealed throughout the year to the more and more advanced students.*

*This paper discusses the didactical, technical and implementation concepts and experiences so far, which bases on two classes (in two different departments) and overall 60 Students.*

## 1 INTRODUCTION

Since Kernigham and Ritchie's "bible" on C [2], "Hello World" is the mantra to introduce beginners to a programming language. Books in that tradition start with a small program which prints a friendly "Hello World" to the screen. To master this task, the student writes a small piece of code, compiles it and runs the program. The program execution has an instant effect on the screen and feels like a first and visible victory over the obstacles of compiler and runtime system.

These books next cover types, variables, control statements and functions by gradually extending the task. However, the student still writes single monolithic programs that create and cover their own application under their own control. In fact, control is the most important issue: the student creates the complete application from scratch. Connecting to the underlying system is rare, and even less do we adapt to other applications outside our own program.

This way of learning is perfect for learning from a book, and in solitude. It teaches the syntax and features of a new programming language rather than how to design a program or an application.

Unfortunately, this is not what a professional software engineer needs. Professional projects tend to be developed by teams, tend to integrate ready made software solutions, and tend to create comparably little portions of self-written code. The main task is to plan, use and provide interfaces, define a powerful yet minimal set of services which support interaction between different components for a often highly complex application. The most important issue is not control, but confidence in pre-and postconditions of a foreign service, as well as reliability of the once negotiated services of our own part. The application itself is not a single monolith, but a complex cooperation between inhomogenous participants, not unlike a human work force.

Teaching how to program at a university is a chance to avoid the "Hello World" trap. A university class is a group of inhomogeneous humans, there is no need for solitude and there are more means to teach than books only.

## 2 INTRODUCTION TO PROGRAMMING

Our course of studies in Heilbronn is called "Software Engineering". The name is a commitment to constructive, cooperative and application oriented solutions rather than pure analytical computer science. We teach software engineering courses from the very start of studies.

Our customers are beginners in their first year. About 50% have some programming experience, mostly school exercises in Pascal, Visual Basic and web applications, but almost no experience with object orientation, and absolutely no experience with working in teams.

### 2.1 Teaching Goals

The teaching goals of our course "Introduction to Programming" cover both technical and soft skills. Of course, we want to teach a programming language along, but the course is not restricted to the peculiarities of one language. Since we decided to use Java, some topics become more prominent than others due to the power and complexity of the language. The programming course has a duration of two semesters with 180 instruction hours, and an overall workload of about 250 hours for the average student. So it is a chance to do much more than the notorious "Learning *Something* in 21 Days".

Thanks to a hint from Debora Weber-Wulff [4] we found the draft of Lynn A. Steins Interactive Programming Course [3] and registered as beta testers for this approach. We did not, however, use the elaborate lab examples but decided to concentrate on one big example that expands through the year.

Along with the development of that example we teach decomposition of a complex everyday example into interaction of participants and services. Along with this we teach object orientation, interfaces, types and documentation before we deal with the algorithmic part of the solutions. This is a rather top down approach to programming: from participants to services to parameters and at last, to control flow.

Students still experience early visible effects and feel that they "do" something. The programs are not monolithic but always parts of a more complex application. Students are not in full control and always use an environment that we provide and do not reveal before the programming concepts therein can be understood by the students.

Our students almost always work in teams. The team has to discuss the programs, to plan and design an implementation. Students are encouraged to use interaction diagrams, interfaces, and later, design patterns [1].

### 2.2 The Restaurant Example

In former years, we followed a more traditional way of teaching: class was amplified by examples and small labs where exercises are closely adapted to the material just heard. This resulted in a huge collections of unconnected "finger exercises". Although these exercises are great to show a single aspect, students often lack the ability to transfer this aspect to another context. Our students — and the teacher — therefore often wished for a single, complex and powerful example where all those small exercises could be integrated.

Influenced by Stein [3] we decided to use the restaurant example. A restaurant simulation is a complex, concurrent application, and highly interactive. Participants are guests, waiters, cooks, and other active personifications. On the other hand, passive services include the menu, a storeroom, cash register, and expands as far as a database of recipes. The simulation starts with a guest entering the restaurant. She or he might choose a table, a waiter is assigned to that table. Interaction between participants requires exchange of information. For example, the guest's order is passed to the kitchen by the waiter, and the kitchen cooks according to the recipe. Back in the restaurant room, the waiter serves the dishes, the guest enjoys the meal and pays. Finally the money is added to the cashier.

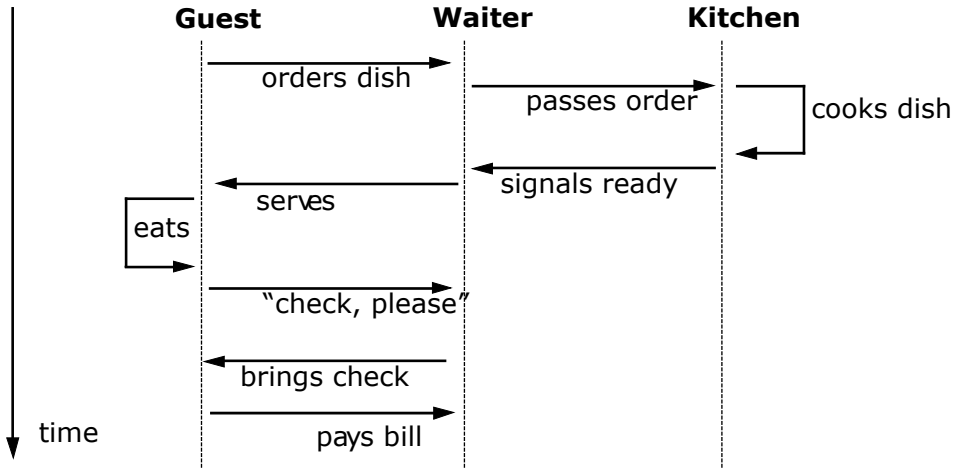

Figure 1 Example scenario

### 2.3 Didactic Concepts

Needless to say: "learning by doing" is a valuable concept. And programming courses have no difficulty to lure students into action. Many of them already have some programming experience and are hard to stop when provided with a computer. These "code warriors" [4] tend to play a guessing game against compiler and runtime system, wrangling the number of their incomprehensible error messages to zero. Once the program runs, the game is won. And over ...

One task of the course is to show those code warriors that there is more to programming than just zero error messages. They have to restrain from typing code until a plan (on paper!) has formed. They have to work in small teams and share their skills with others who might be more on the timid side. All this is very hard on them because they feel not helped but hindered.

Another type of student has less confidence and tends to sit beside the self-appointed expert watching the fingers glide over the keyboard. The timid student pays some attention to syntax like brackets and semicolons, but might soon become completely passive since the expert does not explain what happens.

In order to make things easier for both extremes, the code warriors and the timid ones, we dedicate a whole open-ended day to programming. Such a "java day" allows to concentrate on one topic only, provides time for discussion, breaking up in groups, joining the experiences during the day in class again, and a little small talk about so-called minor issues like frustration and team problems. The java days are a combination of classroom instruction, exercises, self study and projects.

• **Exercises** are traditional small tasks, solved with a slight alteration or extension of given code. They also may be theoretical questions about concepts where we expect text in a natural language We emphasize on reflection and expression of technical theoretical knowledge all the more because we know how hard this is for many students.

• **Self Study** is a form of team coaching. Teams are confronted with learning material and come together with the teacher to clarify open questions. If time allows, the team also prepares a small lecture for other students. Especially valuable is a session of test quizzes about the material. The team prepares the questions, their learning goals and solution. In class they present the questions and help others to find the solution. As a reward, a portion of those quizzes is integrated in the final exam.

• **Projects** become more prominent as the course proceeds. In the second semester, the teams choose their projects from a given set of tasks. Each task is a major extension to the given restaurant simulation. By that time students know the simulation structure and code quite well. their task is to design interfaces and use helpful patterns to integrate their parts into the final restaurant version. Typical tasks are database connections, concurrency, or persistency of the simulation's current state. We support the teams by workshops about code review, report assignments, and talk about frustration and team problems.

## 3 UNVEILING THE COMPLEXITY

The restaurant example is implemented according to the model view controller pattern MVC [1]. Each participant has a view, which is a graphical representation with a picture, some function buttons

which cause the picture to change or other participants to react in some visible way. The underlying functions are implemented in the model components. The event handling mechanism of java is the control to connect graphical objects like a button to the corresponding model method that really implements the reaction. General superclasses define the overall structure of each participant, and each participant extends the given general structure by its own specific view and behavior.

Figure 2 shows the general class structure and some example methods. Model, View and Control are the general classes for all participants like Guest — as shown — as well as Waiters, Tables, Cooks etc. The simulation itself is divided into a powerful general class Simulation (containing complicated code) and the simpler Restaurant class.

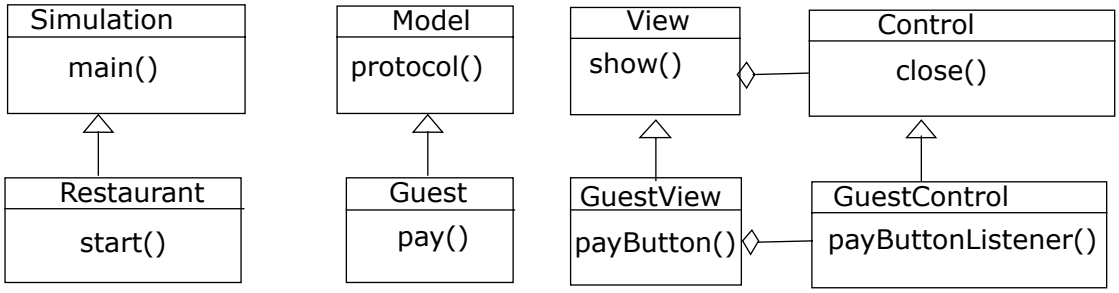


Figure 2 — General class structure

During the course's year we unveil the complexity of this pattern step by step.

### 3.1 No Code Visible

Students start with a version they can play with, but all the code is hidden. They see a small restaurant simulation with some tables, waiters, a kitchen and some guests. A GUI representation of a participant always contains a picture, some specific buttons and an individual protocol that tells only about the actions of that specific object. In addition, there is a general protocol, that represents the restaurant simulation itself and tells about object creation as well as a summary of all actions of all participants within the simulation.

Students learn how the simulation works, they understand both the independence and inter-dependence of participants by watching the game and reading the different protocols. They also have to draw interaction diagrams in order to follow the flow of services between participants.

### 3.2 Restaurant Visibility

Next, students see the restaurant code which starts the whole simulation. It mainly contains a start() method that creates the participants and invokes some example functions by calling the object's method. This is an alternative to pressing the GUI button. The code is carefully written without cryptic C or C++ legacies like the infamous main() method would show. It contains only object creations and method invocations without any parameters or return values.

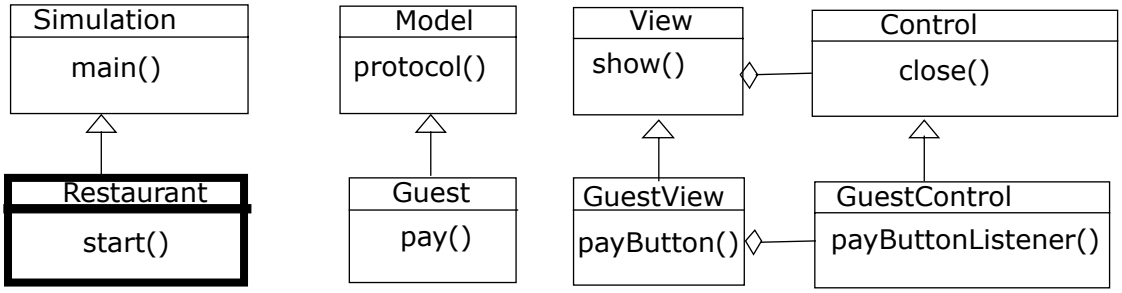


Figure 3 — Visible code in step 2

Students learn how to create objects, how to invoke a service by code instead of the GUI button. Since the method invocations in that version have neither parameters nor return values, they do not need to know about types or naming.

### 3.3 Model Visibility

In the third step we show the model code of guests, waiters, tables and so on.

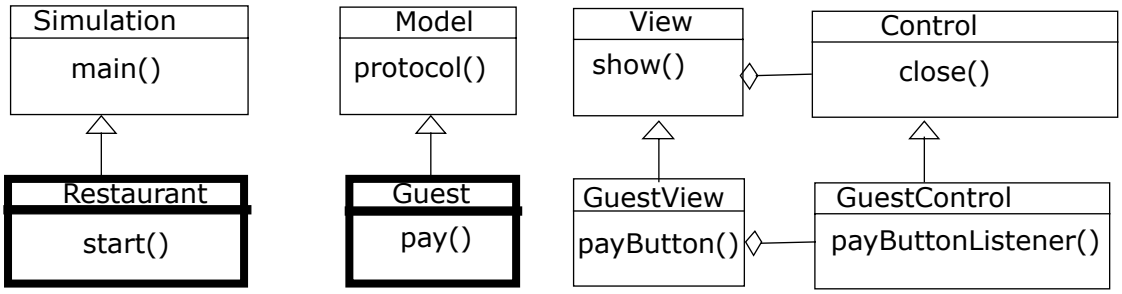


Figure 4 — Visible code in step 3

Here we start teaching how to program in the small. Students learn about names and types, method invocation and parameters, as well as exception handling. They gradually extend the given material by new functionality, have to find a set of useful services and provide information needed for these services. Interaction diagrams become more elaborate on details and force the student to first think about the interaction, then write the code.

This step is also the place where we teach the algorithmic aspects of methods. Many students already know about control statements in other languages, and since Java does not differ very much from those, we discuss syntactical problems rather than the underlying semantics.

Before we can reveal the next layer of our structure, students have to learn about object oriented concepts, especially inheritance and the resulting polymorphism. As an example, students extend the simulation class by special kinds of employees: Waiters, Managers, Cooks. All of them are also Persons, as Guests are too.

### 3.4 GUI Visibility

When the second semester starts, the students have a quite powerful version of the restaurant. Waiters, cooks and guests talk to each other, the dishes are cooked, the money flows and everything works almost natural. But the effects can be witnessed only in the protocols. We now open up the GUI part of our simulation and show the individual GUI representations. They are simple to read because the more sophisticated parts of a GUI are still hidden in the superclass.

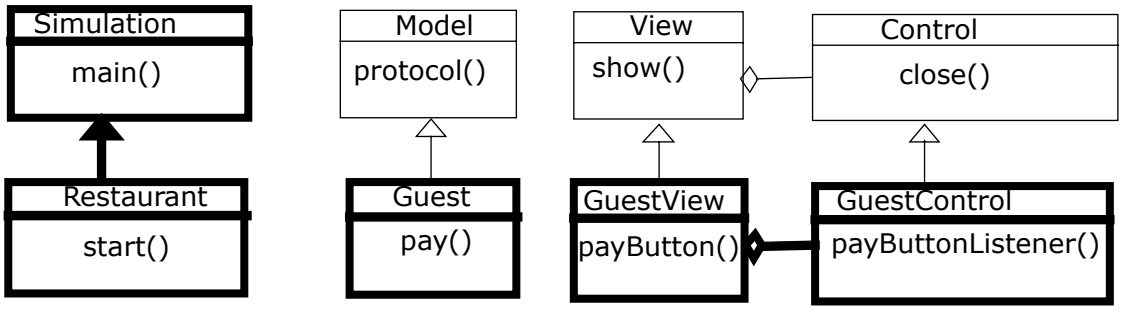


Figure 5 — Visible code in step 4

Students here learn how to create windows, add components like a picture, the buttons and a text field for protocols. They soon start to write extensions like a form to fill in certain parameters for a function, so their GUI representations adapt to the full functionality of step 3.

### 3.5 Full Visibility

Before the students start their final projects, we reveal the complete code including the superclasses. Students learn the power of interfaces, a set of useful patterns [1]. With the help of those we discuss the crucial point of connecting our simulation to the environment: how do we use files and databases? What once was a dreaded topic in the olden times of procedural programming, seems to come easier with patterns and object orientation. Most students have no problems to use a file for the menu list, or a database for recipes. We also have a set of projects that explore new areas of programming which we do not explicitly teach: concurrency and persistency are examples. Students receive some guidance to learning material but there is no lecture about it. The successful projects were cheered by the class during presentation.

## 4 LESSONS LEARNED

The rather luxury combination of instruction and practical work requires an environment which is not always available. In order to switch from classroom discussion, instruction to exercises and project work as we like, we need both a classroom and a lab reserved during the whole day. Next, reviews of the students solution takes a lot of time and is almost too much for just one instructor. We will experiment with a tutoring system where students of higher classes coach the teams and help to integrate the projects into a final version.

The code warriors were first easily frustrated since they did not have the control over the code. In order to pass the quizzes they have to stay in the restaurant example and they need the whole environment for exercises at home. They also had a hard time to accept the necessity of writing on paper, of drafting interactions and information flow before coding could start.

Absolute beginners however, were quite happy about the power they had in our environment. They did not miss the Hello World mantra because our restaurant provided enough visible effects.

Software engineering virtues are hard to teach to beginners. In order to understand, they must have some experience what can go wrong without a proper version control, without interfaces, and without release cycles. These topics are taught in a parallel course but our students were so absorbed by the small projects that they lost control of the engineering part. So they learned it the hard way and had a feeling of failure when the complete simulation did not work although their individual projects were really successful. The next year we will help them to keep an eye on project management and version control.

## ACKNOWLEDGEMENTS

We would like to thank Diana Schmidt who followed our approach in her introductory programming course at the Medical Informatics department in Heilbronn/Heidelberg. Her valuable suggestions and error messages contributed to the quality of slides and examples.

## REFERENCES

[1] GAMMA, E. HELM, R. JOHNSON, R. VLISSIDES, J. 1994. *Design Patterns*. Addison-Wesley, 1994 ISBN: 0201633612.

[2] KERNIGHAM, B. RITCHIE, D. 1978. *The C Programming Language.* Prentice-Hall, 1978

[3] STEIN, L. A.: *Interactive Programming in Java*; Morgan Kaufman, to appear. Available from web: <URL: http://www.cs101.org/ >

[4] WEBER-WULFF, D. 2000. Combating the Code Warrior - A Different Sort of Programming Instruction. *Proceedings of the ITiCSE 2000.* Helsinki Finland, July 11-13, 2000.