A Software Craftsman's approach to Data Structures

Matti Luukkainen, Arto Vihavainen, Thomas Vikberg
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
Fi-00014 University of Helsinki
{ mluukkai, avihavai, tvikberg }@cs.helsinki.fi

Final draft

Originally appeared as: M. Luukkainen, A. Vihavainen and T. Vikberg: Software Craftsman's Approach to Data Structures. In SIGCSE'12: Proceedings of the 43nd ACM Technical Symposium on Computer Science Education. ACM Press, 2012.

Abstract

Data Structures (CS2) courses and course books do not usually put much emphasis in the process of how a data structure is engineered or invented. Instead, algorithms are readily given, and the main focus is in the mathematical complexity analysis of the algorithms. We present an alternative approach on presenting data structures using worked examples, i.e., by explicitly displaying the process that leads to the invention and creation of a data stucture and its algorithms. Our approach is heavily backed up by some of the best programming practices advocated by the Agile and Software Craftsmanship communities. It brings the often mathematically oriented CS2 course closer to modern software engineering and practical problem solving, without a need for compromise in proofs and analysis.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education *Computer Science Education*

General Terms Algorithms, Design, Experimentation, Verification

Keywords data structures, software engineering best practices, worked example, instructional design

1 Introduction

Data Structures (CS2) is the cornerstone course of most computer science curriculums. It usually covers principles of algorithm analysis, stacks, queues, linked lists, set implementations (binary search trees and hash tables), sorting algorithms and graphs. The course is usually taken at a later part of the first year or during the second year after students have acquired basic programming and mathematics knowledge.

Most CS2 course books, such as the books by Aho et. al. [2], Cormen et. al. [9] and Sedgewick [16], take a traditional, procedural pseudocode based approach to presenting the algorithms. A growing number of books where pseudocode is replaced with a real programming language exists, see e.g. [11, 18], but the traditional pseudocode-based approach dominates most of the CS2 courses.

A typical CS2 course implementation introduces one data structure at a time. The data structure and its algorithms are usually covered step by step as follows. First, the need of a data structure is motivated and the algorithms for the data structure are given as pseudocode. Then, the behavior of the algorithms are visualized and clarified by simulations, and finally, the complexity and correctness of the algorithms are analyzed formally.

There are several problems in this approach. As the data structures and related algorithms are presented as given, the students do not have the opportunity to discover the data structures themselves. Instead of just showing what a data structure is and how it works, it would be more enlightening to see the actual process of how a data structure is engineered step by step from scratch. In literature this type of explicitly thought out teaching process is often referred to as *worked example* [7]. Worked examples are often used in the cognitive apprenticeship [8] based learning methods that have recently had many successful applications in teaching programming (see e.g. [5, 7, 13, 17]).

Another problem is related to the way in which pseudocode (or real programming language code) is used in course books. Algorithms are often written as big monolithical entities, where variables follow a single letter naming convention. This writing style is taxing to a beginner since bad variable naming does not give much of a hint on the purpose of the variables, and the algorithms do not bear any structure that would make their intention more explicit. Simply put, the traditional approach of presenting algorithms does not take into account the limitations of human' cognitive architecture, which sets limits on how many *concepts* the working memory can hold at a time [7].

Unnecessary cognitive load is not the only problem of the pseudocode style used in CS2 courses and course books. Many algorithms of data structures have some common logic, however given pseudocode usually repeats the common parts of the logic. Thus, the pseudocode used in CS2 breaks many of the long known best practices of software engineering such as *don't repeat yourself* (no copy-paste code), *use short methods*, *use simple focused classes*, and *use descriptive names* (see eg. [12, 14]).

The importance of these best practices have lately been heavily advocated by some of the Agile Software Engineering methods, especially Extreme Programming [4] and the Software Craftsmanship community [15, 12, 14].

At least in our university the courses with emphasis in software engineering such as the introductory programming (CS1) are also putting more and more emphasis on the best pracices. Our results indicate that instruction in CS2 benefits from taking good programming practices into account.

The rest of this paper is organized as follows. In section 2, we describe Software Craftsman's approach to data structures, an approach based on worked examples backed heavily by some of the best practices advocated by the Software Craftsmanship and Agile Software Engineering communities. Section 3 gives excerpts of a worked example where a hash table is developed from scratch. Section 4 evaluates the effectiveness of the approach from the students' point of view in contrast to traditional lectures, and section 5 concludes the paper.

2 A Software Craftsman's approach to data structures

In our approach a data structure and its related algorithms are presented in the worked example manner where the actual data structure is engineered step by step from scratch, starting by giving a motivation for the data structure, e.g. a problem, and finishing in a detailed implementation. Instead of pseudocode, a concrete programming language is used (in our case we utilize Java as it is the language in our CS1 courses).

Data structures are engineered using *Test-driven development* [3], a discipline where tests are written before the actual code. Test-driven development is well suited for implementing data structures as it puts lots of emphasis on the interface design of the actual code to be written [1].

Engineering a data structure proceeds in small steps. First an intermediate step is identified and a few tests are written for it. Then the actual code that passes the tests is written. When tests pass, the next step is to write another test that leads one step closer to the solution.

However, if one notices that the code that was written is not clean, i.e., it breaks some of the best programming practices, the code is *refactored* to a clean form. Refactoring means the process where the internal structure of the code is altered but its interface and behavior remain unaltered [10]. In order to efficiently do refactoring, it is essential that the tests are automated, for example, by using a test framework. As we use Java, a natural choice for the test framework has been JUnit [3].

By using clean code principles in expressing algorithms we believe that the understandability of the algorithms is also increased. Using meaningful variable names and splitting algorithms into manageable pieces is expected to ease the students' task of grasping the abstract ideas behind the algorithms.

3 An example: A Hash Table

In this example, we demonstrate our approach by implementing a simple student register, where the information of a student (student number, list of course registrations and grades, etc.) can be quickly retrieved by using the students' name. Information on a student is encapsulated to class `Student` (details omitted), and the name used as a search key is assumed to be a `String`.

We start by defining an interface that the student register data structure is supposed to implement:

```
public interface StudentRegister {
  void add(Student student);
  Student find(String name);
}
```

There are two operations; one for adding a student and one for finding a student based on a name.

Next we define the basic behavior of what is expected from a student register. The behavior is defined in terms of JUnit tests:

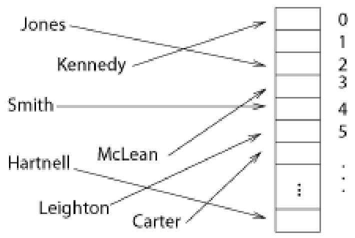
```
@Test
public void addedStudentIsFound() {
  Student jones = new Student("Jones");
  studentReg.add(jones);
  assertEquals(jones, studentReg.find("Jones") );
}

@Test
public void nonExistingStudentIsNotInRegister() {
  assertEquals(null, studentReg.find("Kennedy") );
}
```

In JUnit, each test is defined as a method annotated with `@Test`. Each test is designed to test one simple functionality, and the name of the test method documents its purpose. Execution of a test starts by setting up the state that is expected to hold before the testable behavior is executed. After the state has been set, the testable behavior is executed and the result is verified, typically with the `assert`-family of methods. Tests are run and results can be analyzed automatically, either by "pressing a button" in an IDE or by running a command line script.

In the tests above it is assumed that the variable `studentReg` refers to the hash table under test and that it is empty before the test methods are executed. The first test, `addedStudentIsFound` starts by creating a student object and adding it to the hash table. Then the `find`-method, that is being tested, is called and the result is verified. The second test, `nonExistingStudentIsNotInRegister`, verifies that the register does not attempt to return a student that is not in the register.

Starting the implementation

As the University of Helsinki has roughly 39000 students, it could be possible to store all student-objects in an array of roughly equal size. An idea is to define a mapping from each student to an array index:



Since the data structure is a hash table, we will name and define our initial implementation of the interface as follows:

```
public class RegisterHashTable
               implements StudentRegister {
  private Student[] students;

  public RegisterHashTable(int size) {
    students = new Student[size];
  }

  public void add(Student student) {
  }

  public Student find(String name) {
    return null;
  }
}
```

We define the size of the hash table to be a constant `SIZE` for testing. In JUnit the test setup is done in a method annotated with `@Before`. The method is executed before each test is run, so all the tests start with an empty hash table:

```
@Before
public void setUp() {
  studentReg = new RegisterHashTable(SIZE);
}
```

The idea of the hash table is to map a key (now a `String` representing the student's name) to an index of an array. A simple but not very efficient mapping can be defined first:

```
public void add(Student student) {
  students[0] = student;
}

public Student find(String name) {
  if ( students[0]==null ||
       !students[0].getName().equals(name) )
    return null;

  return students[0];
}
```

This simple implementation that maps each student to the position 0 of the table is enough to get the current tests to pass. In test-driven development, it is a very common technique to first do a hardcoded or "quick and dirty" implementation to make the tests pass and only after that to sit back and think about a more elaborate solution.

It is ugly to first use the method `getName` and then `equals` to compare a students' name to a given name. Following the *tell, don't ask* -principle [12], we simplify this by adding a method `nameEquals` to class Student:

```
public Student find(String name) {
  if ( students[0]==null ||
       !students[0].nameEquals(name) )
    return null;

  return students[0];
}
```

A better hash function

Next step is to introduce a test which adds two students to the hash table. That test forces us to consider the mapping of students to different indices:

```
@Test
public void ifTwoAddedBothAreFound(){
  Student jones = new Student("Jones");
  Student smith = new Student("Smith");

  studentReg.add(jones);
  studentReg.add(smith);

  assertEquals(jones, studentReg.find("Jones") );
  assertEquals(smith, studentReg.find("Smith") );
}
```

A simple approach for calculating a hash value is to sum up the ascii codes of the characters in the name string, and then take modulo base the hash table size of the sum:

```
public void add(Student student) {
  int index = 0;
  for (int i = 0; i < student.getName().length(); i++) {
    index += student.getName().charAt(i);
  }
  index = index % students.length;

  students[index] = student;
}

public Student find(String name) {
  int index = 0;
  for (int i = 0; i < name.length(); i++) {
    index += name.charAt(i);
  }
  index = index % students.length;

  if ( students[index]==null ||
    !students[index].nameEquals(name)) )
  return null;

  return students[index];
}
```

The implementation passes the tests. However, one notices that the hash function is calculated (copy-pasted!) in two different places. This violates the *do not repeat yourself* principle and, if left in the code, most probably will cause all sorts of problems later, for example if the logic of the hash function calculation needs to be changed. So we decide to refactor the copy-paste code to a separate method:

```
public void add(Student student) {
  int index = hashCode(student.getName());
  students[index] = student;
}

public Student find(String name) {
  int index = hashCode(name);

  if ( students[index]==null ||
      !students[index].nameEquals(name) )
    return null;

  return students[index];
}

private int hashCode(String name) {
  int index = 0;
  for (int i = 0; i < name.length(); i++) {
    index += name.charAt(i);
  }
  return index % students.length;
}
```

Tests are run immediately to ensure that refactoring has not broken the code.

Method `add` can be further simplified by replacing the temporary `index` value by the method call:

```
public void add(Student student) {
   students[ hashCode(student.getName()) ] = student;
}
```

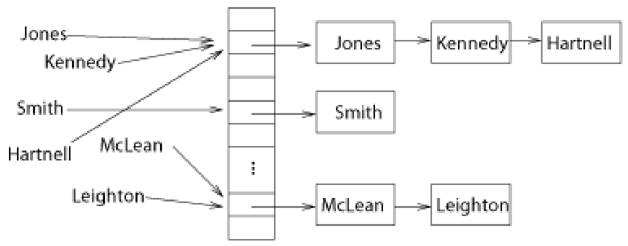
Hash collisions

Next we create a test for a boundary case where more students are added to the register than there is space in the backing table. This makes it evident that in the case of *hash collisions*, an information loss prevention strategy is needed:

```
@Test
public void ifManyAddedEachIsFound(){
  List<Student> students = buildAListOf(SIZE + 1);
  for (Student student : students)
    studentReg.add(student);

  for (Student student : students) {
    assertEquals(student,
                 studentReg.find(student.getName()));
  }
}
```

The test uses a method `buildAListOf(int numberOfStudents)`, which generates a given number of student objects and returns those in a List. In the above example, students in the generated list are added to the hash table, after which the existence of all students is tested.

The test above does not pass which forces us to think of another solution. The idea of chaining the objects to linked lists is introduced. All objects that have the same hash value are entered into a linked list in the corresponding position of the hash table:



Collision chains are implemented with Java's `LinkedList` data structure, which has been covered in an earlier part of the course, and the hash table is declared to hold lists. Methods `add` and `find` are changed correspondingly. The hash function remains the same as it is simply used to select the right list in each case.

```
private LinkedList<Student>[] students;

public RegisterHashTable(int size) {
   students = new LinkedList[size];
   for (int i = 0; i < students.length; i++)
      students[i] = new LinkedList<Student>();
}

public void add(Student student) {
   students[ hashCode(student.getName()) ].add(student);
}

public Student find(String name) {
  for (Student student : students[ hashCode(name) ]) {
    if ( student.getName().equals(name) )
      return student;
  }
  return null;
}
```

After doing the modifications, tests are run and we discover that they now pass.

Separating hash function calculation from hash table

Our hash function implementation uses the well known *division method* [9] for calculating the hash value. There exists also other types of hash functions. In our solution the hash table operations (adding and finding elements), and the calculation of the hash value, are tightly coupled since both are defined in the same class. In other words, the class is not very focused, and has too many responsibilities. This can be problematic if we would like to have alternative approaches for calculating the hash value. The modifiability of our current solution is not optimal due to a class with too many responsibilities.

A good solution to the problem is again refactoring. Calculating the hash value should be extracted to a class of its own. We start the process by writing tests to the hash function:

```
@Before
public void setUp() {
  hasher = new Hasher(SIZE);
}

@Test
public void stringGetsAlwaysSameHashValue() {
  assertEquals( hasher.hash("Jones"),
                hasher.hash("Jones") );
}
```

The class implementing the hash function is called `Hasher`, and the method to calculate the hash value is `hash`.

The next step is to create a class that passes the test. We simply take the method of our previous approach and perform some renaming:

```
public class Hasher {
  private int sizeOfHashTable;

  public Hasher(int size) {
    this.sizeOfHashTable = size;
  }

  public int hash(String string) {
    int value = 0;
    for (int i = 0; i < string.length(); i++)
      value += string.charAt(i);

    return value % sizeOfHashTable;
  }
}
```

The simple test passes. The next step is to plug a hash-ohject into the hash table:

```
public class RegisterHashTable
               implements StudentRegister {

  private LinkedList<Student>[] students;
  private Hasher hasher;

  public RegisterHashTable(int size) {
    hasher = new Hasher(size);
    students = new LinkedList[size];
    for (int i = 0; i < students.length; i++)
      students[i] = new LinkedList<Student>();
  }

  public void add(Student student) {
    students[ hasher.hash(student.getName()) ].add(student);
  }

  public Student find(String name) {
    for (Student student : students[ hasher.hash(name) ] ) {
      if ( student.nameEquals(name) )
        return student;
    }
    return null;
  }
}
```

Strengthening the hash function

All tests pass, and the basic functionality of the hash table is now ready. However, we have some doubts about our hash function. Let us first clarify the structure of the function. We notice that two separate tasks, mapping a string to integer and calculating the hash value based on an integer are currently mixed within one method. Let us refactor the string to integer mapping and hash value calculation to separate methods:

```
public int hash(String string) {
  return hashValue( stringToInteger(string) );
}

protected int hashValue(int value) {
  return value % sizeOfHashTable;
}

protected int stringToInteger(String string) {
  int value = 0;
  for (int i = 0; i < string.length(); i++)
    value += string.charAt(i);

  return value;
}
```

Tests pass after the refactoring, so we can go on. The integer that represents a string is calculated as a sum of ascii codes. This is not a good strategy if the hash table is big as only a small range of hash table indices are used when hashed strings are relatively short. A better schema is to scale up character ascii values based on their position in the string by multiplying them with a suitable prime [6]:

```
private int stringToInteger(String string) {
  int value = 0;
  int multiplier = 1;
  for (int i = 0; i < string.length(); i++) {
     value += string.charAt(i)*multiplier;
     multiplier *= PRIME;
  }
  return value;
}
```

Interestingly this change breaks some of our tests! We quickly discover that the problem is that in some situations the array index becomes negative. With a little bit of analysis, for example by using a debugger, we notice that the cause of the error is integer overflow in method `stringToInteger`.

The problem is fixed by returning the absolute value of the string to integer mapping:

```
private int stringToInteger(String string) {
   // ...
   return Math.abs(value);
}
```

This fixes the problems and all tests pass. We want to prevent this type of accidents by adding a test that checks that calculated hash values are within the expected range[1]:

```
@Test
public void valueWithinRange() {
  String string = "";
  for (int i = 0; i < 1000; i++) {
    string += "a";
    assertTrue( hasher.hash(string) >= 0 &&
                hasher.hash(string) < hashTableSize);
  }
}
```

Further modificationas

Due to size limitations we can't show the complete example here. The hash table example that was done in the lecture was continued with defining another

[1]More representative input sets can be easily generated but are omitted due to space limitations.

type of hash function, *the multiplication method* [9]. The extension was done by extending the class `Hasher`. The only change needed was to override the method that calculates the hash value:

```
public class HashMultiply extends Hasher {
  protected int hashValue(int value) {
    return (int) ( sizeOfHashTable *
                   fractionPart(value * GOLDEN_RATIO) );
  }
}
```

Thanks to the good structure of the solution, it was extremely easy to implement an alternative version of the hash table that uses a different type of hash function.

The lecture was continued by reflecting on what was done from an algorithmical perspective and by performing a complexity analysis of the algorithm. The implementation was also polished, e.g. user friendly creation patterns were created by introducing static factory methods.

4 The use and effectiveness of the approach

The worked example style of presenting data structures and algorithms was applied at the Data Structures course at the University of Helsinki for the first time in Spring 2011. The participants of the course were mainly first year Computer Science majors. The course lasts one term and consists of 4 hours of lectures each week with both programming and non-programming take-home assignments.

The following data structures were covered using the worked example style: stack, singly linked list, binary search tree, hash tables with chaining, open hashing, merge sort and quick sort. Besides just showing the program code, an overhead projector was heavily used in the worked example style of lectures to illustrate and reflect on what was being done. Traditional pseudocode representation of the data structures was also available in handouts, but was not explicitly covered during the worked example lectures.

The rest of the data structures in the course (queue, double linked lists, heap and graphs) were covered in a traditional way. Besides presenting various algorithms and data structures, the course included basics of algorithm analysis. Each of these three parts took roughly one third of the lecturing time of the course.

In order to measure the effectiveness of worked example style of lectures, the students were asked to fill an anonymous survey that asked which lecturing approach the students prefer from a learning point of view. The results are shown in the table below (total number of answers was 52).

Opinion	Percentage
Worked examples much better	44.2 %
Worked examples slightly better	34.6 %
Both equal	5.8%
Traditional lectures slightly better	3.8 %
Traditional lectures much better	5.8 %
No opinion	5.8 %

Altogether 78.8 % of the students thought that the worked example style supports learning better and only 9.6 % preferred the traditional format. Thus students clearly preferred the renewed lecturing approach.

It is somewhat hard to estimate the effect of our approach to the overall learning. Results from both the mid-term exam and the final exam were somewhat better than a year ago when all the lectures were traditional[2]. Pass rate of the course in Spring 2011 was 79 %, while in Spring 2010 it was 71 % – a clear improvement. However, other factors may also have been involved.

Many students gave direct verbal and textual feedback about the lectures. The majority of the feedback was positive, but some semi-critical opinions were also received, eg.:

Programming in the lectures helps to grasp the ideas of the algorithms a bit better compared to pseudocode or readily made program code. However, there is not much difference since programming in lectures proceeds quite quickly and there is not too much time to discuss why the particular solutions were selected in implementing the algorithms. To the lecturer the selections may seem obvious, but for a novice it is not clear why some choices are favoured over some others since the "blind alleys" where a novice programmer often finds herself are not explored.

This is clearly a good comment, and it would certainly be beneficial to discuss more alternative implementation strategies during the lectures e.g. by contrasting the "well engineered" solutions with the more problematic approaches that some students end up with.

5 Conclusions

Traditional pseudocode-oriented CS2-courses have quite often switched students from "software engineering mode", promoted in programming and software engineering courses, to "algorithmics mode". In the algorithmics mode, all good programming practices are often forgotten, and the focus in programming is getting correct end results. Because of the focus in end results, and not in the process, the code written in CS2 courses is often quality-wise poor from software engineering point of view.

We proposed an alternative approach to conducting CS2 lectures. The approach builds on the ideas of worked examples, i.e., instead of just presenting the solutions, the process that leads to the solution is shown in a step-by-step

[2]The lecturer and course content was same in both course instances.

manner. Instead of using pseudocode, the approach uses a real programming language with best programming practices to illustrate the engineering process of a data structure.

Our aim is to bridge the gap between software engineering practices and the more theory-oriented field of data structures and algorithms, and to demonstrate students that they actually benefit from writing well structured, automatically tested, and maintainable code.

The following anonymous feedback summarizes the purpose and benefits of our approach quite well:

Programming during the lectures is a good idea. Not only because of the illustrative power, but also the "hidden agenda" of repeatedly showing the benefits of good programming habits such as automatic tests, refactoring and programming against interfaces. The more students get inspiration and examples of good practices, the more likely it is that they will also pursue those in the future.

During the lectures we guide the students through problem-solving sessions, from motivating the need of a data structure all the way up to a concrete solution. By doing this several times for different data structures, we hope to give a student the means to act herself in similar problem solving contexts. Best programming practices are utilized in the process since those promote the understandability of the solutions by good naming and separation of concerns.

6 Acknowledgments

We thank Jaakko “gndi” Kurhila and Matti “mpa” Paksula for their invaluable contributions.

References

[1] J. Adams. Test-driven data structures: revitalizing CS2. In *Proc. of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, 2009.

[2] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1983.

[3] K. Beck. *Test Driven Development: By Example.* Addison-Wesley, 2002.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004.

[5] T. R. Black. Helping novice programming students succeed. *J. Comput. Small Coll.*, 22(2):109–114, 2006.

[6] J. Bloch. *Effective Java (2nd Edition) (The Java Series).* Prentice Hall, 2008.

[7] M. E. Caspersen and J. Bennedsen. Instructional design of a programming course: a learning theoretic approach. In *ICER '07: Proc. of the 3rd international workshop on Computing education research*, 2007.

[8] A. Collins, J. S. Brown, and A. Holum. Cognitive apprenticeship: making thinking visible. *American Educator*, 6:38–46, 1991.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, 3rd edition.* The MIT Press, 2009.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[11] M. T. Goodrich and R. Tamassia. *Data structures and algorithm analysis in C++, 5th edition.* Wiley, 2010.

[12] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley Professional, 1999.

[13] M. Kölling and D. J. Barnes. Enhancing apprentice-based learning of Java. In *SIGCSE '04: Proc. of the 35th technical symposium on Computer science education*, 2004.

[14] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008.

[15] P. McBreen. *Software Craftsmanship: The New Imperative.* Prentice Hall, 2001.

[16] R. Sedgewick and K. Wayne. *Algorithms, 4th edition.* Addison-Wesley, 2011.

[17] A. Vihavainen, M. Paksula, and M. Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *Proc. of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, 2011.

[18] M. A. Weiss. *Data Structures & Problem Solving Using Java, 4 th edition.* Addison-Wesley, 2009.