

Analysis by Noun Lists and Use Cases : A Case Study

Introduction

In a previous article, I presented an overview of the analysis and design of a simple payroll application. This article, and several others which will follow in this column, come from the third chapter of my book: “Designing Object Oriented C++ Applications using the Booch Method”; which will be published by Prentice-Hall later this year. In that chapter, and in this column we will reexamine the payroll application in far more depth. The purpose of this reexamination will be to answer the question: How do you create a “good object-oriented design”? That is, a design which is “easy” to maintain; has a “long” lifetime; and is comprised of components which can be reused in other applications.

Analysis and Design

Many traditional software methods separate analysis and design into distinct phases of the software lifecycle. Often there are different notations for the two activities; thus analysis documents must be translated into design documents. During design, problems are discovered within the analysis; but repairing the analysis documentation is often considered to be too difficult; so it is left in its deficient form. Eventually, as more problems are found with the analysis, it gets so far out of synch with the design that it may as well be discarded. Often, it is.

But the translation barrier between analysis and design is a symptom of a larger problem. Methodologies that separate analysis and design presume that designs are based upon analyses; that a design cannot begin until an analysis has been completed. This then presumes that the analysis *does not* depend upon the design in any way. In most cases, this is incorrect.

I believe that analysis and design should be concurrent. That knowledge from the analysis should flow into design; and that knowledge from the design should flow back into analysis. I view the analysis and design effort as a continuum of activity; beginning with analysis activities and ending with design activities; but with no clear separation of these activities in any part of the process. By analogy, consider a spectrum. There is definitely red light and definitely yellow light; but no clear separation between them; just a gradual change from red to yellow.

The best way to understand the synergy between analysis and design is to experience it first hand. Therefore this series of articles will present a single, relatively significant, case study.

Notation

I will be using Booch’s notation to document analysis and design decisions. For those of you who are unfamiliar with this notation, a lexicon of its major components is provided in Figure 0.

Why use any notation at all? Because we need some way to see the larger picture of the analysis and design. C++ header files are excellent tools for documenting individual classes, but they are not useful for conveying the relationships between classes. A design notation allows us

to document design decisions in an environment which is free of the constraints of source code. I can, for example, document several different classes on a single diagram. I can also use diagrams to focus on a particular aspect of a group of classes, whereas a header file must describe classes in entirety.

Why choose Booch's notation over Rumbaugh's or Coad's? In fact, the spirit of all three notations is similar, they differ only in degrees of detail. All these notations are successful at communicating essential design decisions. However, I find Booch's notation to be more complete than the others because it has more adornments and icons that I can use to document my designs. It also has a wide range of diagrams allowing me to make design decisions at very high, and very low levels.

Some designers believe that notation should not be complex. That a complex notation interferes with the design process. But I say that it takes complexity to manage complexity. One cannot discuss complex topics without a complex language to discuss them in.

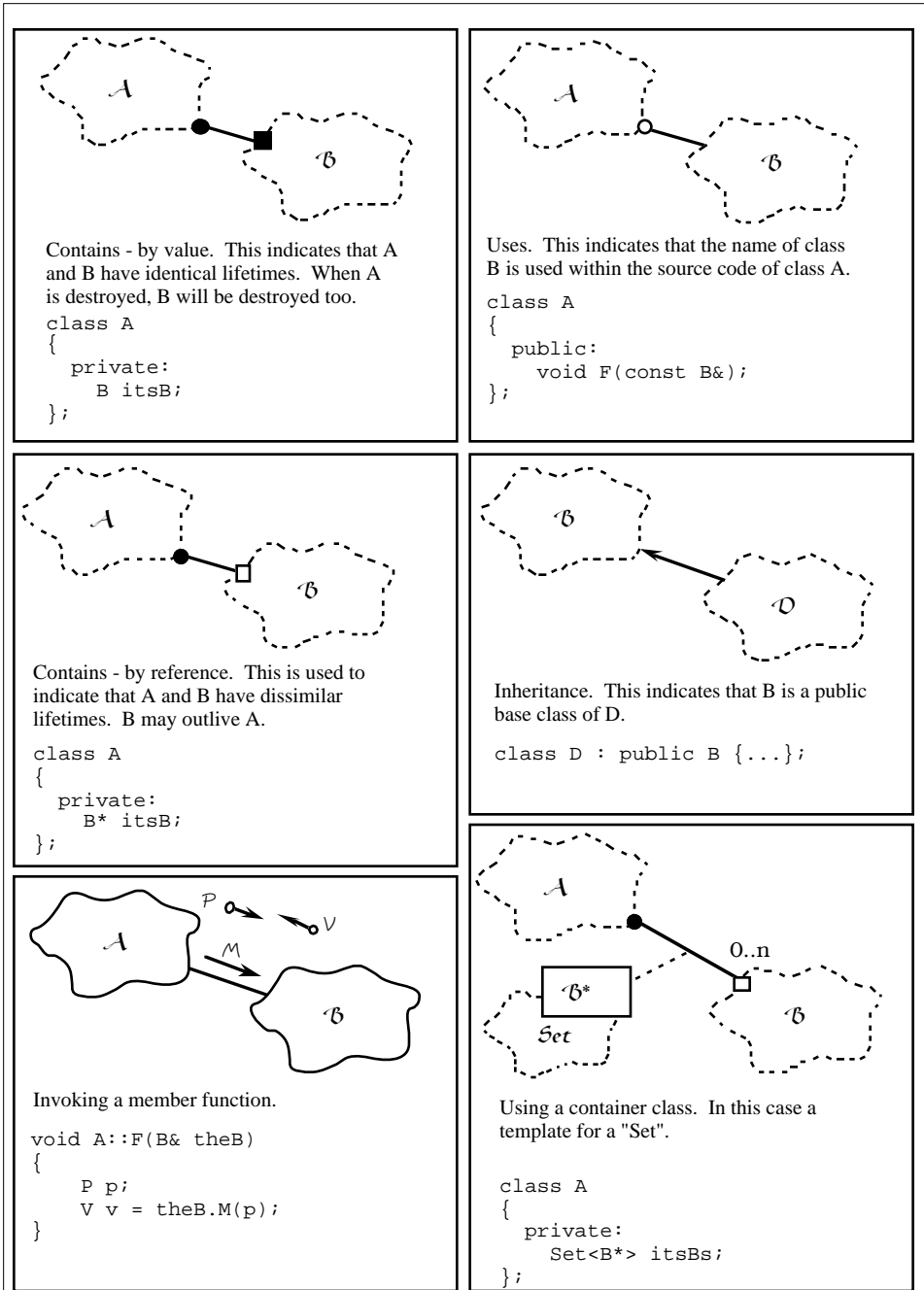


Figure 0
Booch - C++ Lexicon

Case Study: A Batch Payroll Application

There is a prevalent concern in the industry that object-orientation is inappropriate for certain kinds of applications. The following case study should dispel this notion; at least as far as batch business applications are concerned. The application is a simplified model of a payroll system. It has enough complexity to be interesting, but not so much as to be daunting.

Specification

A Simple Batch Payroll System

This system consists of a database of the employees in the company, and their associated data such as timecards. The system must pay each employee. Employees must be paid the correct amount, on time, by the method that they specify. Also, various deductions must be taken from their pay.

- Some employees work by the hour. They are paid an hourly rate which is one of the fields in their employee record. They submit daily time cards which record the date and the number of hours worked. If they work more than 8 hour per day, they are paid 1.5 times their normal rate for those extra hours. They are paid every Friday.
- Some employees are paid a flat salary. They are paid on the last working day of the month. Their monthly salary is one of the fields in their employee record.
- Some of the salaried employees are also paid a commission based on their sales. They submit sales receipts which record the date and the amount of the sale. Their commission rate is a field in their employee record. They are paid every other Friday.
- Employees can select their method of payment. They may have their paychecks mailed to the postal address of their choice; they may have their paycheck held for pickup by the Paymaster; or they can request that their paychecks be directly deposited into the bank account of their choice.
- Some employees belong to the union. Their employee record has a field which holds the weekly dues rate. Their dues must be deducted from their pay. Also, the union may assess service charges against individual union members from time to time. These service charges are submitted by the union on a weekly basis. Any such service charges must be deducted from the appropriate employee's next pay amount.
- The payroll application will run once each working day, and will pay the appropriate employees on that day. However the system will be told what date that the employees are to be paid to. So the system will generate payments for records between the last time the employee was paid, up to the specified date.

How do we begin?

We could begin by generating the database schema. Clearly this problem calls for some kind of relational database, and the requirements give us a very good idea of what the tables and fields should be. It would be easy to design a workable schema and then start building some queries. However, this approach will generate an application for which the database is the central concern.

Databases are *implementation details*! They should be absent from the initial object model. Far too many applications are inextricably tied to their databases because they were designed with the database in mind. Remember the definition of abstraction: “the amplification of the essential and the elimination of the irrelevant”. The database is irrelevant at this stage of the model. It is merely a technique used for storing and accessing data; nothing more.

Analysis by “noun lists”

Then how should we begin? Many texts suggest that we consider the nouns in the requirements list as candidates for objects in our model. Figure 1 shows the nouns that can be extracted from the requirements.

<i>amount</i>	<i>bankAccountOfTheirChoice</i>	<i>comissionRate</i>	<i>commission</i>
<i>company</i>	<i>database</i>	<i>date</i>	<i>day</i>
<i>deduction</i>	<i>employee</i>	<i>employeeRecord</i>	<i>field</i>
<i>hour</i>	<i>hourlyrate</i>	<i>method</i>	<i>methodOfPayment</i>
<i>month</i>	<i>pay</i>	<i>paycheck</i>	<i>paymaster</i>
<i>postalAddressOfTheirChoice</i>	<i>salary</i>	<i>sales</i>	<i>salesReceipts</i>
<i>serviceCharge</i>	<i>system</i>	<i>timecards</i>	<i>union</i>
<i>unionMember</i>	<i>weeklyDuesRate</i>		

Figure 1
Noun list from the Payroll System requirements

Compiling a list of nouns as a first step gives us something to do; and that may make us feel “warm and fuzzy”. At least we don’t have to ask how to begin. However, once the list is compiled we are still faced with the question: “What do we do next?” The list in Figure 1 does not go very far in helping us answer this question; it is reasonably impenetrable. Imagine the list that might accrue from 50 pages of requirements!

Some texts suggest that you prune the list by eliminating anything that does not correspond to a physical thing, role, event, process, etc. This can be difficult since the matching criteria are subject to interpretation. One analyst might determine that *paycheck* is a physical thing, and another might determine that it is outside of the model.

As an example of the variability of such prunings, take a moment to prune the above list. Remove obvious synonyms, and any noun or noun phrase which does not relate to a physical thing, a role, an event, or a process. Then compare your list to the list in Figure 2. The two lists should be somewhat different; perhaps remarkably so. This demonstrates the crudeness of compiling noun lists.

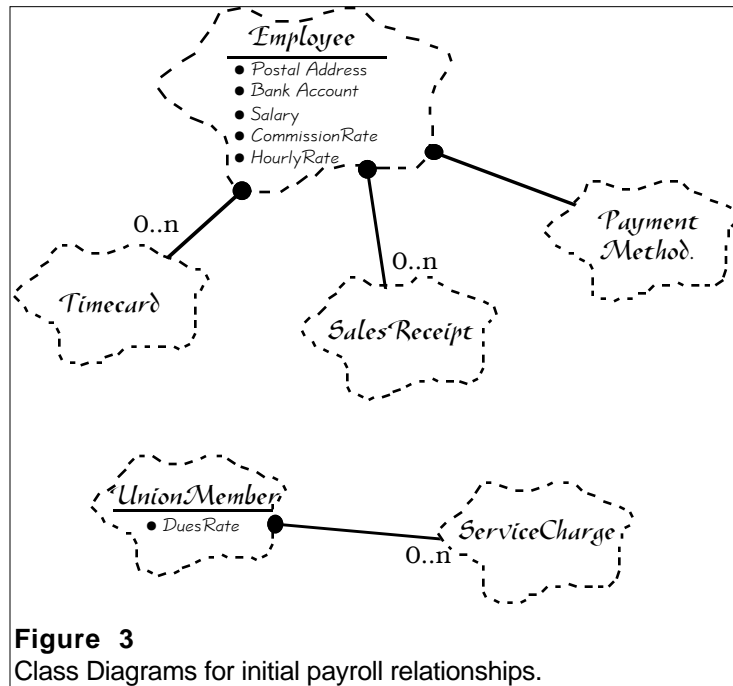
<i>bankAccountoftheirChoice</i>	<i>comissionRate</i>	<i>date</i>	<i>employee</i>
<i>hourlyrate</i>	<i>methodOfPayment</i>	<i>pay</i>	<i>paymaster</i>
<i>postalAddressOfTheirChoice</i>	<i>salary</i>	<i>salesReceipts</i>	<i>serviceCharge</i>
<i>timecards</i>	<i>unionMember</i>	<i>weeklyDuesRate</i>	

Figure 2
Pruned noun list from the Payroll System requirements

However crude, noun lists have their place. The pruned list in Figure 2 is a little easier to come to terms with than the unabridged list from Figure 1. Although some of the elements of this list may remain puzzling; e.g. what do we do with *paymaster*; others present some clear relationships. Here are a few:

1. Employees may have time cards or sales receipts.
2. All Employees have a method of payment.
3. Employees have either a salary and/or a commission rate, or they have an hourly rate.
4. Union members have weekly dues rates.
5. Union members also have a list of service charges associated with them.
6. Depending upon their method of payment, an employee may have a bank account or postal address of their choice.

The class diagram which represents these relationships is shown in Figure 3.



There are many things missing from these diagrams. For example, how are *UnionMember* and *Employee* related? How do *Employee* objects know whether they are paid hourly or by salary or by commission? Before we arbitrarily try to determine the answers to these questions, lets use

Jacobson's technique of "use-cases¹" to see what illumination they may provide.

Analysis by "Use Cases"

When we perform "use-case" analysis, we ask ourselves: "What kind of things can the users of this system do, and how does the system respond." We then enumerate each of these scenarios and describe them in detail.

For example, what can users of our payroll system do?

1. Add a new employee
2. Delete an employee
3. Post a time card
4. Post a sales receipt
5. Post a union service charge
6. Change employee details. (e.g. hourly rate, dues rate, etc.)
7. Run the payroll for today.

Lets examine each of these cases in detail.

Adding Employees

Use Case: 1

Add New Employee

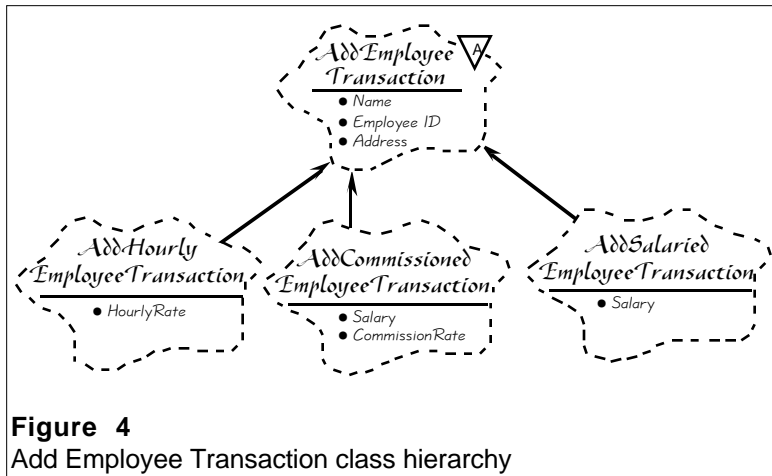
A new employee is added by the receipt of an Add Employee Transaction. This transaction contains the employee's name, address and assigned employee number. The transaction has three forms:

```
AddEmp <empid> "<name>" "<address>" H <hourly-rate>  
AddEmp <empid> "<name>" "<address>" S <monthly-salary>  
AddEmp <empid> "<name>" "<address>" C <monthly-salary> <commission-rate>
```

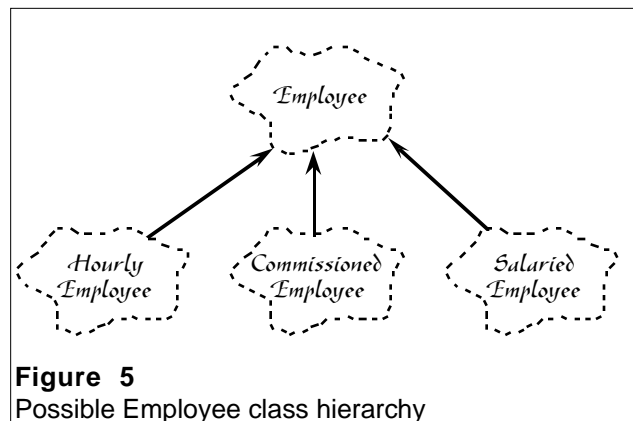
The employee record is created with its fields assigned appropriately

Use case 1 hints at an abstraction. There are three forms of the AddEmp transaction, yet all three forms share the <empid>, <name> and <address> fields. This leads us to the conclusion that there is an *AddEmployeeTransaction* abstract base class, with three derivatives: *AddHourlyEmployeeTransaction*, *AddSalariedEmployeeTransaction* and *AddCommissionedEmployeeTransaction*. See Figure 4.

¹ *Object Oriented Software Engineering, A Use Case Driven Approach*, Ivar Jacobson, Addison Wesley, 1992
Copyright © 1991, 1992, 1993 by R. C. M. Consulting Inc. All rights reserved



Use case 1 specifically talks about an employee record. This implies some sort of database. Again our predisposition to databases may tempt us into thinking about record layouts or the field structure in a relational database table; but we should resist these urges. What the use case is really asking us to do is to create an employee. What is the object model of an employee? A better question might be: what do the three different transactions create? To my mind, they create three different kinds of employee, mimicking the three different kinds of Add transactions. Figure 5 shows this possible structure.



An astute analyst might have inferred this structure from the requirements; but certainly not from the noun list. Still, it is perhaps too soon to be sure that this structure is the most optimal for the payroll application. So let's continue with the analysis.

We are already making Design Decisions.

The above analysis has suggested to us that there are some abstract classes in our model. Making a class abstract is a design decision. It pertains to the sharing of code and the closure² of the class. It has very little to do with the analysis of the model.

Remember that analysis describes “what” the application does, but not how it does it. An

² “Closure” refers to the Open/Closed principle as expounded in “Object Oriented Software Construction” by Bertrand Meyer, Prentice Hall, 1988

equally valid description of “what” would have been to state that there were three utterly independent classes: *SalariedEmployee*, *HourlyEmployee* and *CommissionedEmployee*. However, the very names of these classes cry out for some kind of unifying relationship; and so a design decision is made to create the abstract class *Employee*. This is just a small example of how design and analysis begin to intermix very early in the analysis of an application.

Deleting Employees

Use Case 2

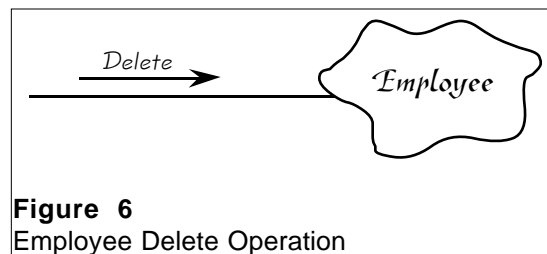
Deleting an Employee

Employees are deleted when a Delete Employee transaction is received. The form of this transaction is as follows:

DelEmp <empid>

When this transaction is received, the appropriate employee record is deleted.

Use Case 2 implies that every employee object should have a ‘delete’ operation as in Figure 3–6.



Posting Time Cards

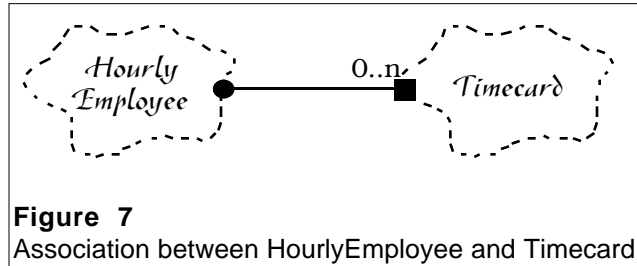
Use Case 3

Post a time card

Upon receipt of a Timecard transaction, the system will create a timecard record and associated it with the appropriate employee record

Timecard <empid> <date> <hours>

This use case points out that some transactions apply only to certain kinds of employee; strengthening the idea that the different kinds of employee should be represented with different classes. In this case there is also an association implied between timecards and hourly employees. Figure 7 shows a possible static model for this association.



Posting Sales Receipts

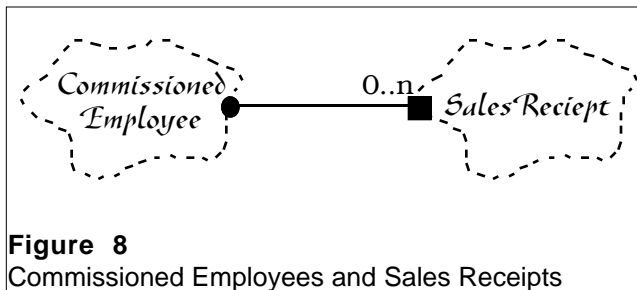
Use Case 4

Posting a Sales Receipt

Upon receipt of the Salesreceipt transaction, the system will create a new sales receipt record and associate it with the appropriate commissioned employee.

Salesreceipt <empid> <date> <amount>

This use case is very similar to Use Case 3. It implies the structure shown in Figure 8.



Posting a Union Service Charge

Use Case 5

Posting a Union Service Charge

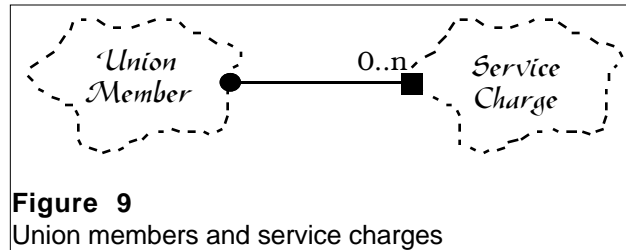
Upon receipt of this transaction the system will create a service charge record and associate it with the appropriate union member.

Servicecharge <memberID> <amount>

This use case shows that union members are not accessed through employee IDs. The union maintains its own identification numbering scheme for union members. Thus the system must be able to somehow associate union members and employees. There are many different ways to

provide this kind of association. So to avoid being arbitrary, let's defer this decision until later. Perhaps constraints from other parts of the system will force our hand one way or another.

One thing is certain. There is a direct association between union members and their service charges. Figure 9 shows a possible static model for this association.



Reflection: What have we learned so far?

We have learned that compiling a list of nouns from the specification does not provide as much insight as analyzing use cases. This is not to say that noun-lists are a bad idea; it is to say that you should not expect too much from them. Noun lists can only provide the crudest of initial approximations for the model.

We have also learned that design decisions can be made very early in the analysis process, and that such decisions help us think more clearly about the analysis.

Conclusion

Although there is much left to be done to complete the analysis and design of the Payroll system, we have made a good beginning. We see the beginnings of a static representation for the employees, and also the beginnings of a transaction model. The next article in this series will continue the analysis and design of the Payroll system by completing the analysis of the use cases and focusing on the subject of *abstraction*: “The most important word in OOD”. We will hunt for, and find, many abstractions. And we will show how those abstractions can be implemented as abstract classes.

In subsequent articles in this series we will explore the concept of modeling “real-world” objects; we will study groups of classes called class categories; we will define what “cohesion” and “coupling” mean in an object oriented design; we will describe what is meant by the “inversion of source code dependencies”; we will develop metrics for measuring the quality of an object-oriented design; and we will discuss the benefits of “Object Factories”.