simple batch payroll system

simple batch payroll system will provide only a very small part of the business value the customers need.

we do the kind of quick **analysis** and **design session** that often takes place **at the start**

**of a** normal **iteration**.

The customer has **selected the stories for the iteration**, and now we have to figure out **how** we are going to **implement them**.

**Took some notes** while conversing with our customer **about the stories** that were

selected **for the** first **iteration**.

# Rudimentary Specification

**Employee Record** work by:

* **hourly rate** - input hourly rate, list of daily time card
* **a flat salary** - input monthly salary
* **Commission based** - input monthly salary, input commission rate, list of sales receipts
* any - if in union, input weekly dues rate, service charges

Some employees belong to the union. Their dues must be deducted from their pay. Also, the union may assess service charges against individual union members from time to time. Union service charges are submitted by the union on a weekly basis and must be deducted from the appropriate employee's next pay amount.

Employees can select their method of payment.

**Payment method** for:

* **hourly rate**:

If they work more than 8 hours per day, they are paid 1.5 times their normal rate for those extra hours.   
Daily time card: the date and number of hours worked.

They are paid every Friday.

* **a flat salary:**   
  paid on the last working day of the month.
* **Commission based** on their sales.  
  Sales receipts: the date and the amount of the sale

They are paid every other Friday.

They may have their paychecks:

* mailed to the postal address
* held for pickup by the paymaster
* directly deposited into the bank account.

The payroll **application** will run once each working day and pay the appropriate employees on that day.

The system will be told what date the employees are to be paid to, so it will generate payments for records from the last time the employee was paid up to the specified date.

## Notice

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 might 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!* Consideration of the database should be deferred as long as possible**.

**Remember the definition of abstraction**: "the amplification of the essential and the elimination of the irrelevant."

**At this stage of the project, the database is irrelevant; it is merely a technique used for storing and accessing data, nothing more.**

# Analysis by Use Cases

Instead of starting with the data of the system, let's **start by considering the behavior of the system**.

One way to capture and analyze the behavior of a system is to **create *use cases***. A use case is

like a user story that has been elaborated with a little more detail.

When we perform use case analysis, we **look to the user stories** and **acceptance tests** to find out the kinds of **stimuli that the users** of this system **provide**. (**means input**) Then we try to figure out **how the system responds to those stimuli**.

For example, here are the user stories that our customer has chosen for the next iteration:

**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

Let's **convert** each of these **user stories into** an elaborated **use case**. We **don't need** to go into **too much detail**: **just enough to** help us **think through the design of the code** that fulfills each story.

## Adding Employees

### Use Case 1: Add New Employee

A new employee is added by the AddEmp TRansaction.

The AddEmp transaction has three forms, all of which share the employee's **name**, **address**, and assigned **employee** **number**. fields.

The AddEmp transaction has three forms:

**1.** AddEmp <EmpID> "<name>" "<address>" H <hrly-rate>

**2.** AddEmp <EmpID> "<name>" "<address>" S <mtly-slry>

**3.** AddEmp <EmpID> "<name>" "<address>" C <mtly-slry> <com-rate>

**Alternative 1 – wrong transaction structure:** If the transaction structure is inappropriate, it is printed out in an error message, and no action is taken.

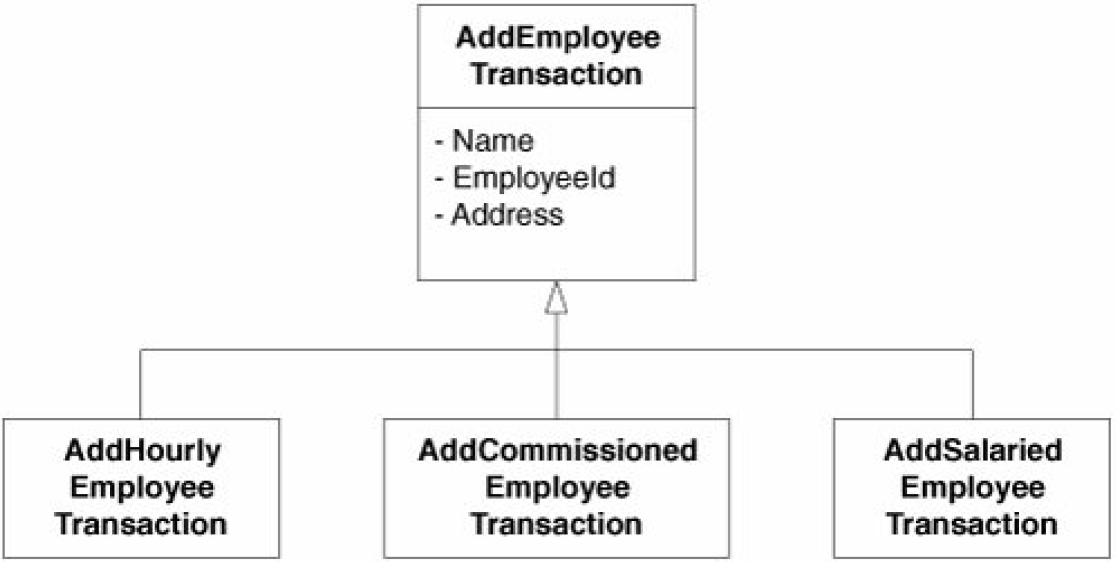
#### Notice

Use case 1 hints at an abstraction. The AddEmp transaction has three forms, all of which share the same 3 fields. **We can use the COMMAND pattern to create an abstract base class with three derivatives**. **This structure conforms nicely to the Single-Responsibility Principle** (SRP) by splitting **each job into its own class**.

AddEmployeeTransaction

* AddHourlyEmployeeTransaction,
* AddSalariedEmployeeTransaction,
* AddCommissionedEmployeeTransaction

**Figure 26-1. AddEmployeeTransaction class hierarchy**



**The alternative** would be to put all these jobs into a single module. Although doing so

might reduce the number of classes in the system and therefore make the system simpler, it would

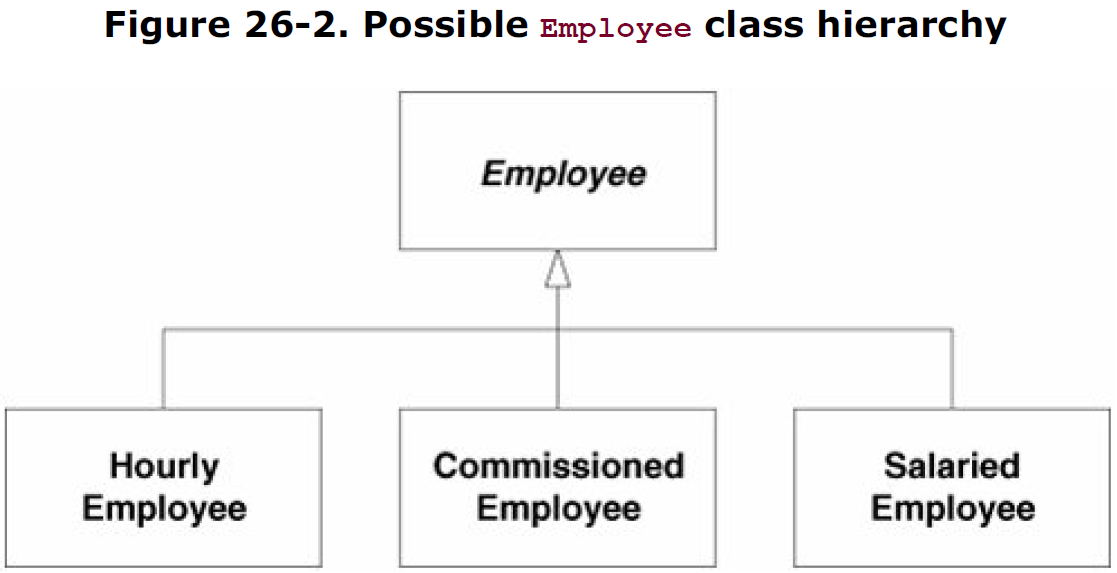
also **concentrate all the transaction-processing code in one place**, creating a **large and potentially error-prone module**.

#### Return object

We should resist talks about an employee record in terms of database.

What the use case is really asking us to do is create an employee. **What is the object model of an employee?** A better question might be:

**What do the three transactions create?** In my view**, they create three kinds of employee objects**, mimicking the three kinds of AddEmp transactions.



## Deleting Employees

### Use Case 2: Deleting an Employee

Employees are deleted when a DelEmp transaction is received. The form of this

transaction is as follows:

DelEmp <EmpID>

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

**Alternative 1 - Invalid or unknown EmpID:** If the <EmpID> field is not structured correctly or does not refer to a valid employee record, the transaction is printed with an error message, and no other action is taken.

Put this job to DeleteEmployeeTransaction class

## Posting Time Cards

### Use Case 3: Post a Time Card

On receipt of a TimeCard transaction, the system will **create a time card record and associate it with the appropriate employee record**.

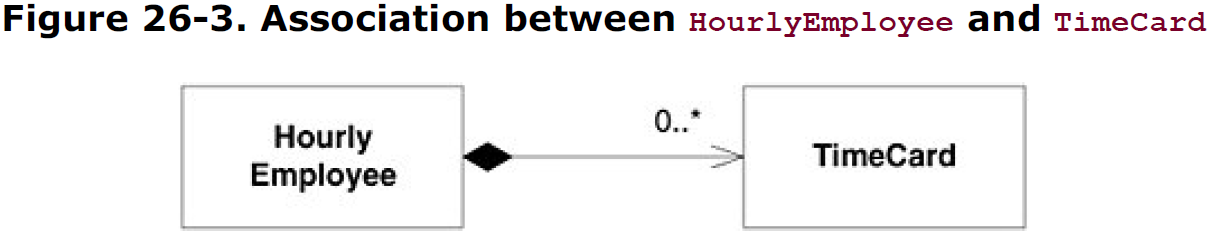
TimeCard <empid> <date> <hours>

**Alternative 1 - The selected employee is not hourly** - The system will print an appropriate error message and take no further action.

**Alternative 2 - An error in the transaction structure** - The system will print an appropriate error message and take no further action.

**This use case points out that some transactions apply only to certain kinds of employees**,

strengthening the idea that **each kind should be represented by different classes**. In this case, there is also **an association implied between time cards and hourly employees**.



## Posting Sales Receipts

### Use Case 4: Post a Sales Receipt

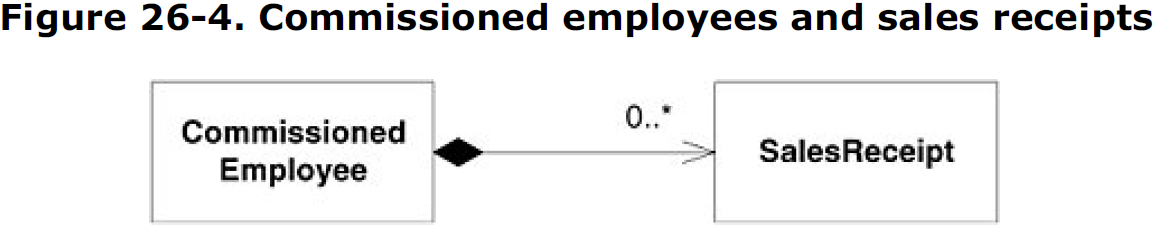
On receipt of the SalesReceipt TRansaction, the system will **create a** **new salesreceipt**

**record and associate it with the appropriate commissioned employee**.

SalesReceipt <EmpID> <date> <amount>

**Alternative 1 - The selected employee not commissioned** - The system will print an appropriate error message and take no further action.

**Alternative 2 - An error in the transaction structure** - The system will print an appropriate error message and take no further action.



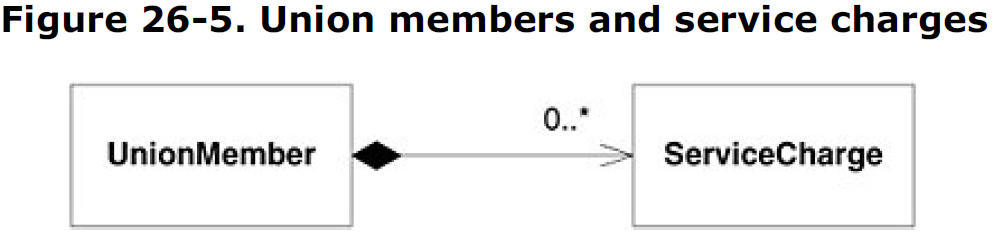
## Posting a Union Service Charge

### Use Case 5: Post a Union Service Charge

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

ServiceCharge <memberID> <amount>

**Alternative 1 - Poorly formed transaction -** If the transaction is not well formed or if the <memberID> does not refer to an existing union member, the transaction is printed with an appropriate error message.



#### Notice

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 associate union members and employees. There are many 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.

## Changing Employee Details

### Use Case 6: Changing Employee Details

Upon receipt of this transaction, the system will alter one of the details of the appropriate

employee record.

There are several possible variations to this transaction:

ChgEmp <EmpID> Name <name> Change employee name

ChgEmp <EmpID> Address <address> Change employee address

ChgEmp <EmpID> Hourly <hourlyRate> Change to hourly

ChgEmp <EmpID> Salaried <salary> Change to salaried

ChgEmp <EmpID> Commissioned <salary> <rate> Change to commissioned

ChgEmp <EmpID> Hold Hold paycheck

ChgEmp <EmpID> Direct <bank> <account> Direct deposit

ChgEmp <EmpID> Mail <address> Mail paycheck

ChgEmp <EmpID> Member <memberID> Dues <rate> Put employee in union

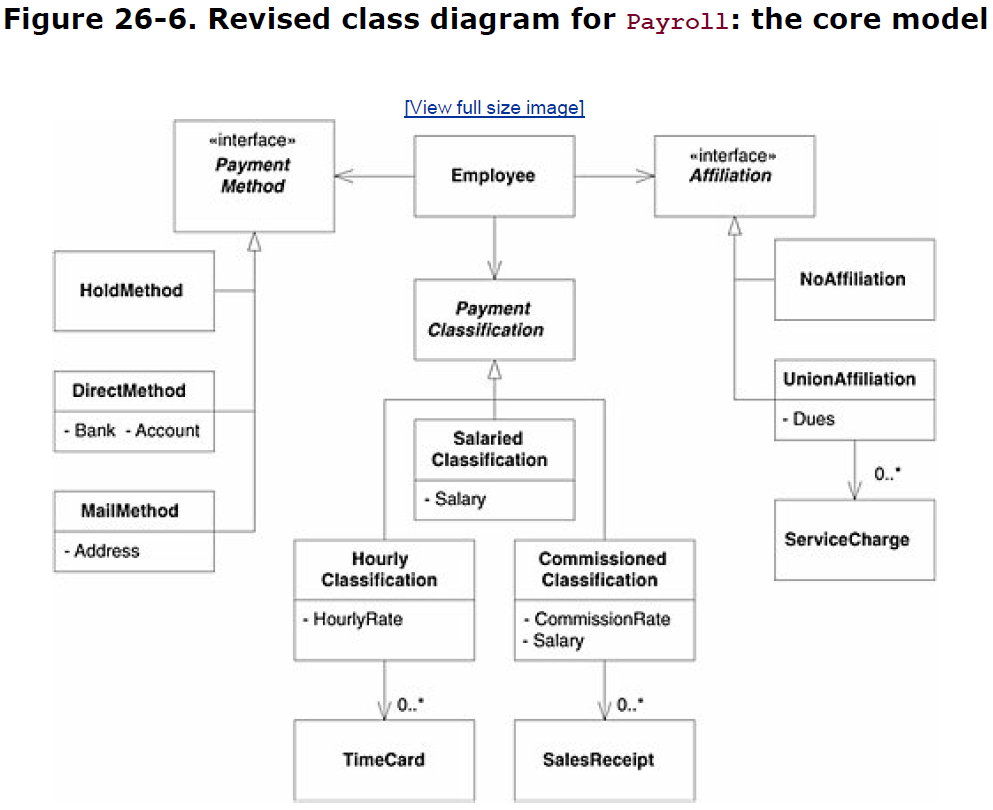
ChgEmp <EmpID> NoMember Cut employee from union

**Alternative 1 - Transaction errors** - If the structure of the transaction is improper, <EmpID> does not refer to a real employee, or <memberID> already refers to a member, the system will print a suitable error and take no further action.

It has told us all the employee aspects that must be changeable.

## **Payroll: the core model**

**The fact that we can change an employee from hourly to salaried** means that the diagram in Figure 26-2 is certainly invalid. Instead**, it would probably be more appropriate to use the STRATEGY pattern for calculating pay**.



### Notes

##### Payment classification

The Employee class could hold a strategy class named PaymentClassification.

This is an advantage because we can change the PaymentClassification object without changing any other part of the Employee object.

When an hourly employee is changed to a salaried employee, the HourlyClassification of the corresponding Employee object is replaced with a SalariedClassification object.

PaymentClassification objects come in 3 varieties:

* The HourlyClassification objects maintain the hourly rate and a list of TimeCard objects.
* The Salaried-Classification objects maintain the monthly salary figure.
* The Commissioned-Classification objects maintain a monthly salary, a commission rate, and a list of SalesReceipt objects.

##### Method of payment

The **method of payment** must also be changeable. This idea implemented by using the STRATEGY pattern and deriving three kinds of PaymentMethod classes:

* If the Employee object contains a MailMethod object, the corresponding employee will have paychecks mailed to the address recorded in the MailMethod object.
* If the Employee object contains a DirectMethod object, the corresponding employee's pay will be directly deposited into the bank account recorded in the DirectMethod object.
* If the Employee contains a HoldMethod object, the corresponding employee's paychecks will be sent to the paymaster to be held for pickup.

##### Union membership

Model applies the NULL OBJECT pattern to union membership. Each Employee object contains an Affiliation object, which has two forms:

* If the Employee contains a NoAffiliation object, the corresponding employee's pay is not adjusted by any organization other than the employer.
* If the Employee object contains a UnionAffiliation object, that employee must pay the dues and service charges that are recorded in that UnionAffiliation object.

### Conclusion

This use of these patterns (Strategy, Null Object) makes this system conform well to the **Open/Closed Principle** (OCP).

**The Employee class is closed against changes in payment method, payment classification, and union affiliation.**

New methods, classifications, and affiliations can be added to the system **without affecting Employee**.

Our *core model*, or architecture, it's at the heart of everything that the payroll system does. There will be many other classes and designs in the payroll application, but they

will all be secondary to this fundamental structure.

## Payday

### Use Case 7: Run the Payroll for Today

On receipt of the payday transaction, the system:

1. finds all those employees that should be paid on the specified date.
2. then determines how much they are owed
3. and pays them according to their selected payment method.
4. An audit-trail report is printed showing the action taken for each employee.

Payday <date>

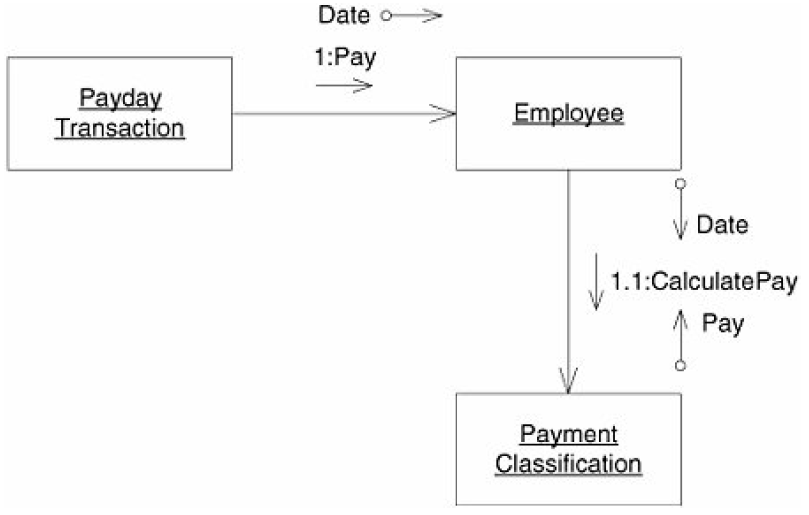
#### Notes

Although it is easy to understand the intent of this use case, **BUT** it is not so simple to determine what impact it has on the static structure of CORE MODEL. We need to answer several questions.

**First**, how does the Employee object know how to calculate its pay? Where does this get done?

The ideal place seems to be in the PaymentClassification derivatives. **These objects maintain the records needed to calculate pay**, so **they should probably have the methods for determining pay**.

**Figure 26-7. Calculating an employee's pay (**a collaboration diagram that describes howthis might work**)**

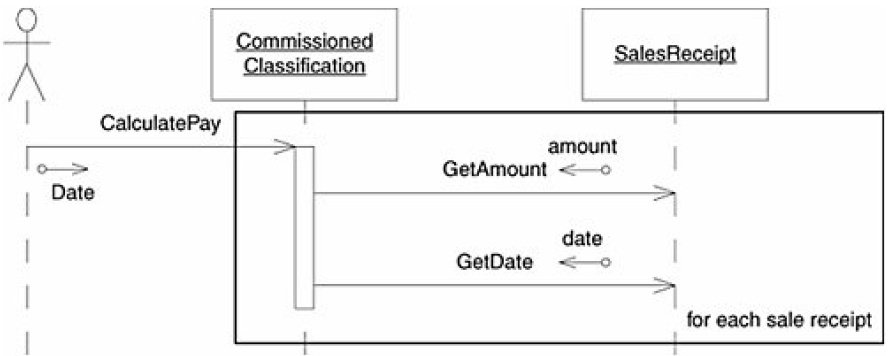


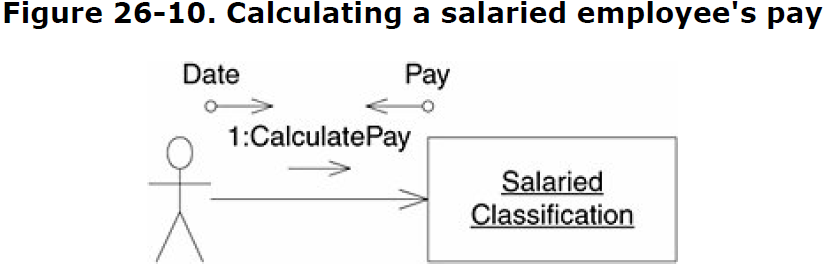
When asked to calculate pay, the Employee object refers this request to its PaymentClassification object. The algorithm used depends on the type of PaymentClassification that the Employee object contains.

Figures 26-8 through 26-10 show the three possible scenarios.

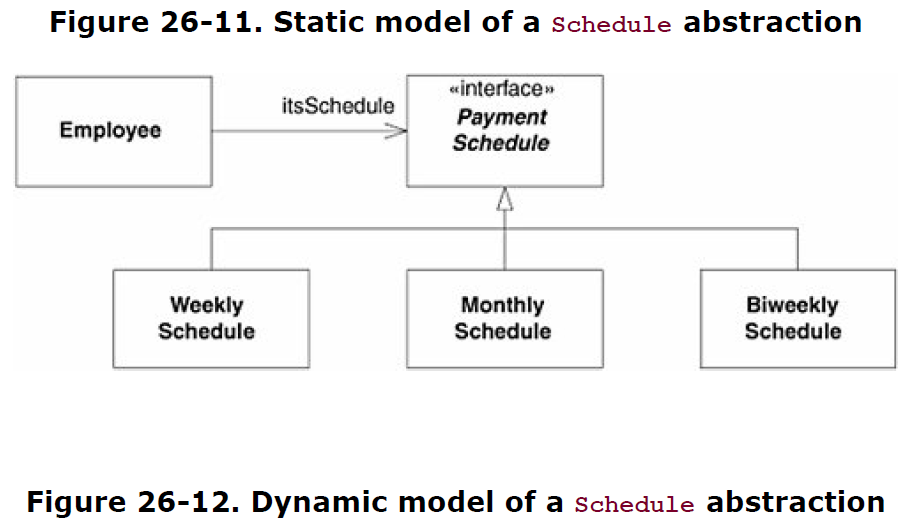


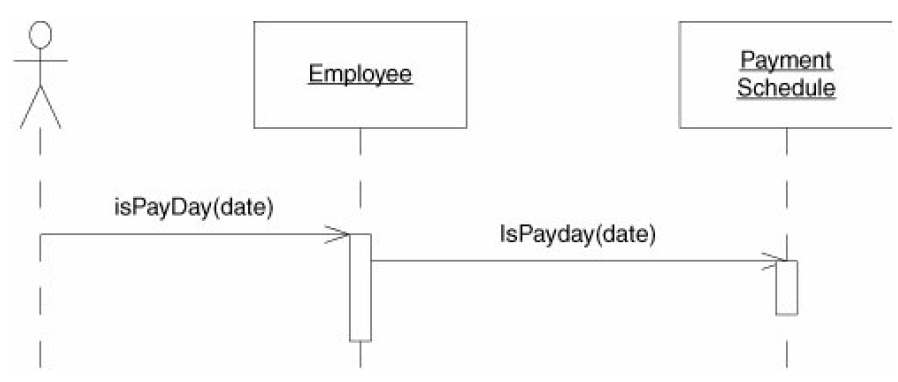
**Figure 26-9. Calculating a commissioned employee's pay**





### Payment Schedule



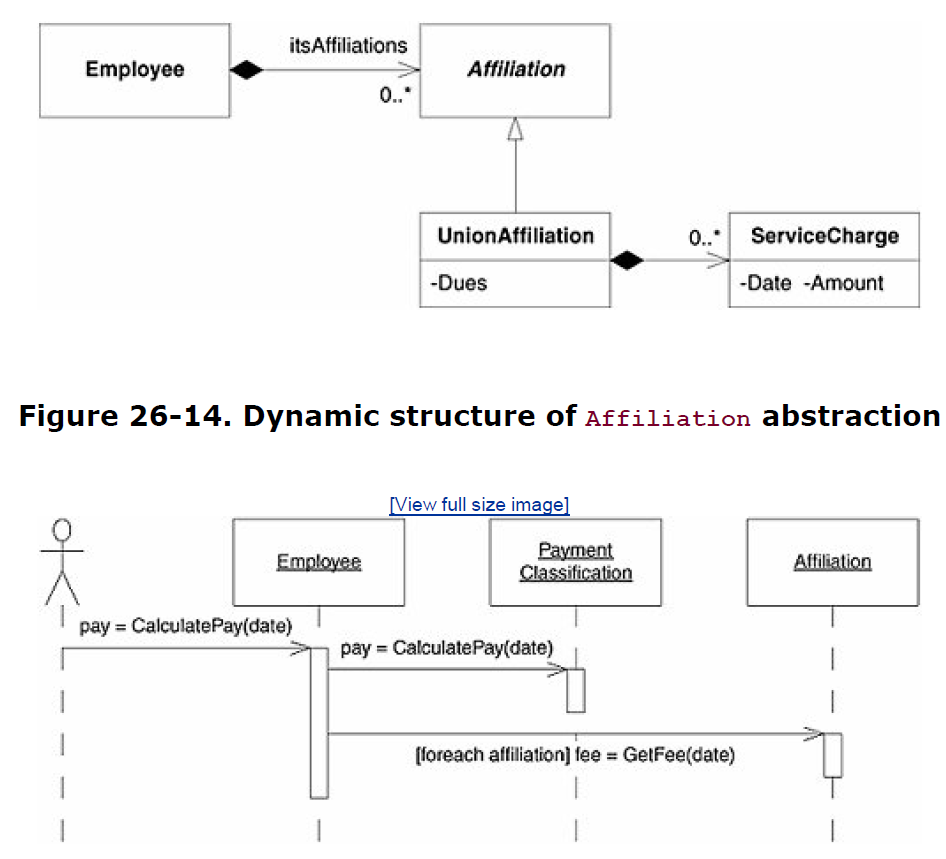


### Affiliations (Union)

The list of Affiliation objects has obviated the need to use the NULL OBJECT pattern for unaffiliated

employees. Now, the list of affiliations for an employee who has no affiliation will simply be empty.

**Figure 26-13. Static structure of Affiliation abstraction**



# 3. Reflection: Finding the Underlying Abstractions

A simple use case analysis can provide a wealth of information and insights into the design of a system. Payroll core model resulted from thinking about the use cases, that is, thinking about behavior.

**To use the OCP effectively, we must hunt for abstractions and find those that underlie the application.**

Often, these abstractions are not stated or even alluded to by the requirements of the application or even the use cases. Requirements and use cases may be too steeped in details to express the generalities of the underlying abstractions.

**Employee Payment - The abstraction here is that *all employees are paid*.**

Let's look again at the requirements. We see statements like this: "Some employees work by the hour" and "Some employees are paid a flat salary" and "Some . . . employees are paid a commission." This hints at the following generalization: All employees are paid, but they are paid by different schemes.

**Payment Schedule - *All employees are paid according to a schedule*.**

Looking for other abstractions, we find "They are paid every Friday," "They are paid on the last working day of the month," and "They are paid every other Friday."

If, as the requirements imply, we delegated the issue of schedule to the Payment-Classification class, our class could not be closed against issues of change in schedule. When we changed payment policy, we would also have to test schedule; when we changed schedules, we would also have to test payment policy. Both OCP and SRP would be violated.

**Payment Methods - *all employees receive their pay by some method*.**

**Affiliations - *the employee may be affiliated with many organizations that should be automatically paid from the employee's paycheck*.**

The requirements imply that employees may have affiliations with a union; however, the union may not be the only organization that has a claim to some of an employee's pay. Employees might want to make automatic contributions to certain charities or have their dues to professional associations paid automatically.

**By elaborating the user stories into use cases and hunting through those use cases for abstractions, we've created a *shape* for the system.**

Note, however, that this architecture has been created by looking at only the first few user stories. We did not do a comprehensive review of every requirement in the system. Nor did we demand that every user story and use case be perfect. We also did not do an exhaustive design of the system, complete with class and sequence diagrams for every jot and title that we could think of.

**Thinking about design is important. Thinking about design in small, incremental steps is *critical*.** Doing too much is worse than doing too little. In this chapter, the amount we did was just about right. It feels unfinished, but it's enough for us to understand and make progress with.

# 4. Implementation

## Architecture notes

### COMMAND pattern

The implementation of the transaction class is the COMMAND pattern

### FACADE pattern

PayrollDatabase is an example of the FACADE pattern.

### Deferring details about the database

In general, I consider database implementations to be details. Decisions about those details should be deferred as long as possible. Whether this particular database will be implemented with a relational database management system (RDBMS), or flat files, or an object-oriented database management system (OODBMS), is irrelevant at this point. Right now, I'm simply interested in creating the API that will provide database services to the rest of the application. I'll find appropriate implementations for the database later.

Deferring details about the database is an uncommon but very rewarding practice. Database decisions can usually wait until we have much more knowledge about the software and its needs. By waiting, we avoid the problem of putting too

much infrastructure into the database. Rather, we implement only enough database facility for the current needs of the application.

### Using Template Method to add employees

Note that the AddEmployeeTransaction object sends messages to *itself* in order to get the appropriate PaymentClassification and PaymentSchedule objects. These messages are implemented in the derivatives of the AddEmployeeTransaction class. This is an application of the TEMPLATE METHOD pattern.

Two things are of particular interest here.

First, **when the TEMPLATE METHOD pattern is applied**, as it is here, **for the sole purpose of creating objects**, it goes by the name **FACTORY METHOD**.

Second, **it is conventional for the creation methods in the FACTORY METHOD pattern to be named** MakeXXX().

I realized both of these issues while I was writing the code, and that is why the method names differ between the code and the diagram.

### Using Static fields, the SINGLETON or MONOSTATE patterns

By now, you have noticed that the PayrollDatabase provides static access to its fields. In effect, PayrollDatabase.employees is a global variable. For decades, textbooks and teachers have been discouraging the use of global variables, with good reason. Still, global variables are not intrinsically evil or harmful. This particular situation is an ideal choice for a global variable. There will ever be only one instance of the PayrollDatabase methods and variables, and it needs to be known by a wide audience.

You might think that this could be better accomplished by using the SINGLETON or MONOSTATE patterns.

It is true that these would serve the purpose. However, they do so by using global variables themselves. A SINGLETON or a MONOSTATE is, by definition, a global entity. In this case, I felt that a SINGLETON or a MONOSTATE would smell of needless complexity. It's easier to simply keep the database global.

### Using Standard exception message or **Meaningful exception classes**

Note the use of InvalidOperationExceptions. This is not particularly good long-term practice but suffices this early in development. After we get some idea of what the exceptions ought to be, we can come back and create **meaningful exception classes**.

### NULL OBJECT pattern

in place with the NoAffiliation class.

### Developers and business decisions

If something wasn't certainly mentioned in the user stories or the use cases and looks like business decision, like program should produce different results, I have to ask my customer or project manager about this, since they

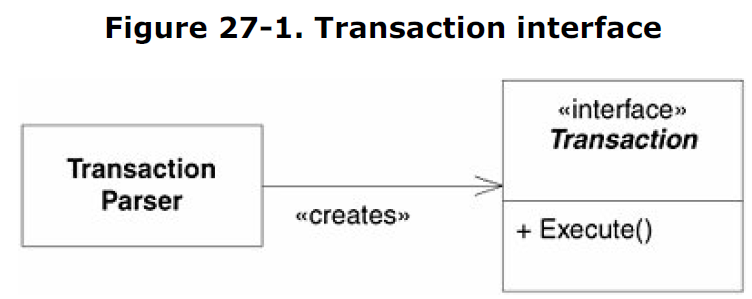
might have very different ideas.

### Use of the STRATEGY pattern.

All detailed paing calculations are deferred to the contained strategy classes: classification, affiliation, and method.

## Transactions

We begin by thinking about the transactions that represent the use cases.



### Add Employee Transaction

#### Static model

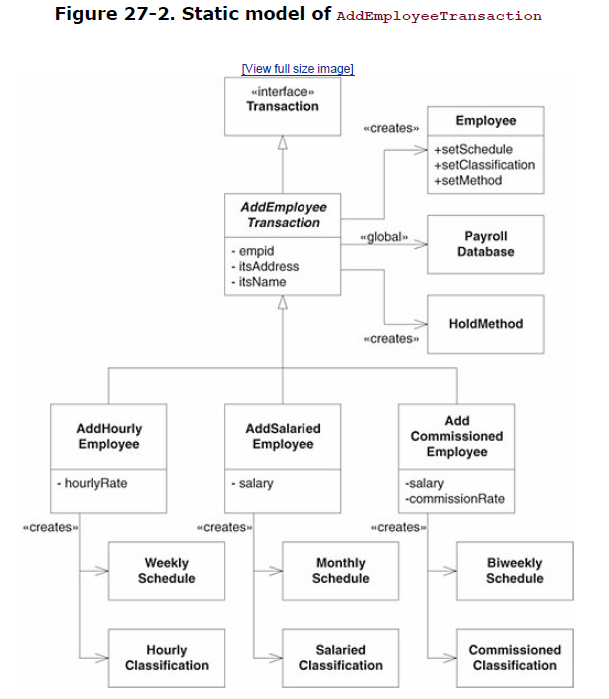
Below a potential structure for the transactions that add employees.

Note that it is within these transactions that the employees' payment schedule is associated with their payment classification.

This decision conforms nicely to OCP and SRP.

It is the responsibility of the transactions, not the core model, to specify the association between payment type and payment schedule.

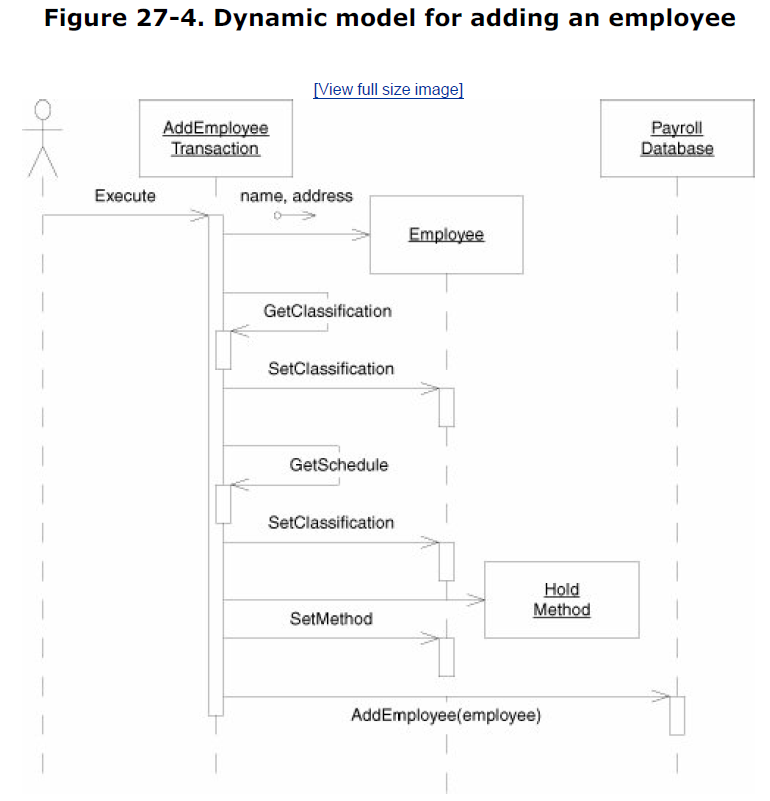
What's more, that association can be changed without changing the core model. Note, too, that the default payment method is to hold the paycheck with the paymaster.



#### Dynamic model

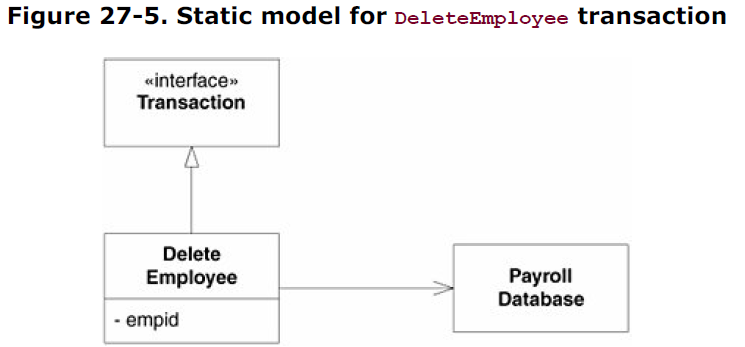
Tis is the implementation of the TEMPLATE METHOD pattern in the AddEmployeeTransaction class. This class implements the Execute() method to call two pure virtual functions that will be implemented by derivatives.

These functions, MakeSchedule() and MakeClassification(), return the PaymentSchedule and PaymentClassification objects that the newly created Employee needs. The Execute() method then binds these objects to the Employee and saves the Employee in the PayrollDatabase.

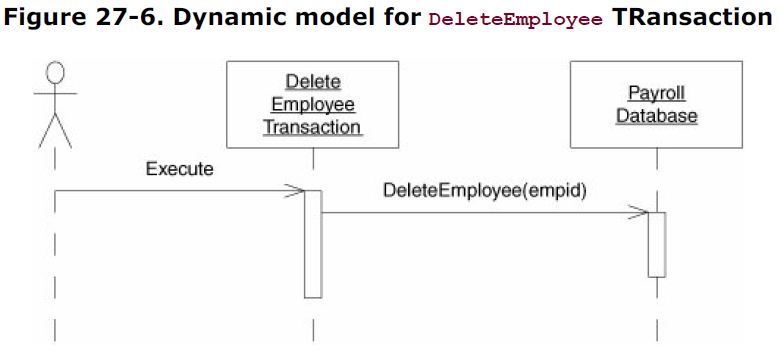


### Deleting Employees

#### Static model



#### Dynamic model

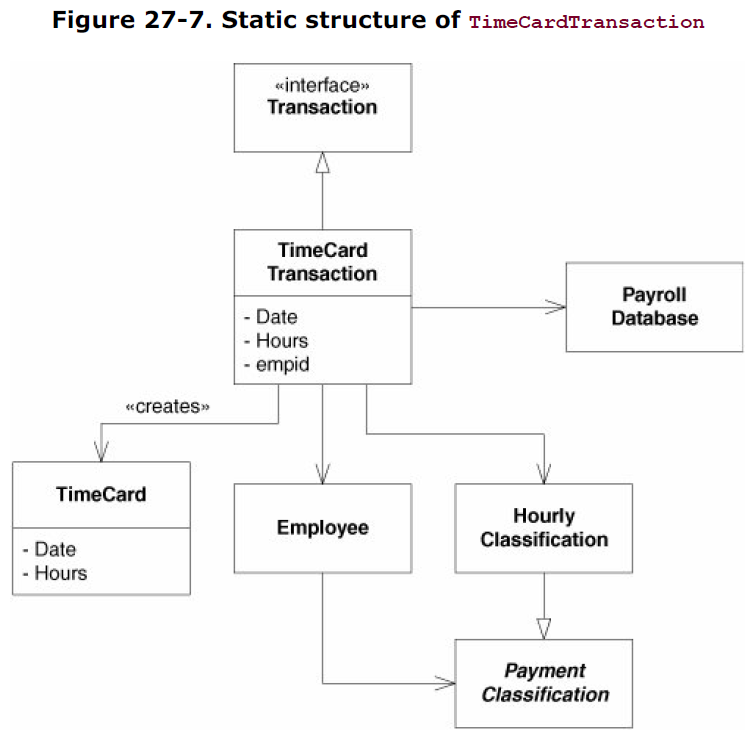


### Time Cards

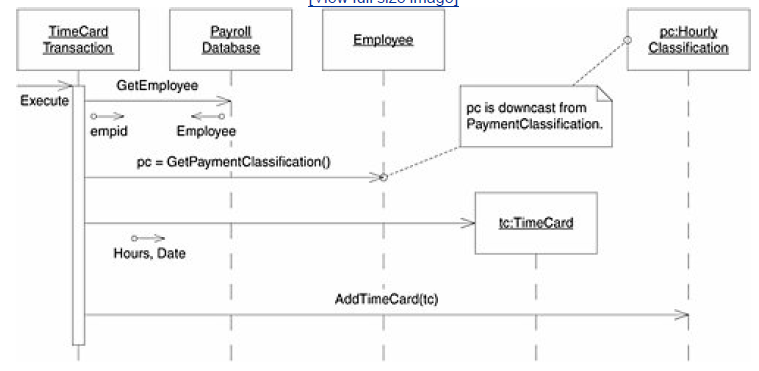
The basic idea is that the transaction gets the Employee object from the PayrollDatabase, asks the Employee for its PaymentClassification object, and then creates and adds a TimeCard object to that PaymentClassification.

Note that we cannot add TimeCard objects to general PaymentClassification objects; we can add them only to HourlyClassification objects. This implies that we must downcast the PaymentClassification object received from the Employee object to an HourlyClassification object.

#### Static model

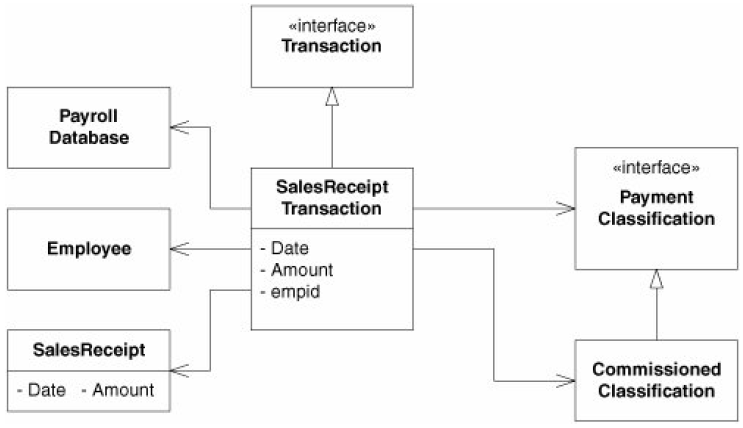


#### Dynamic model

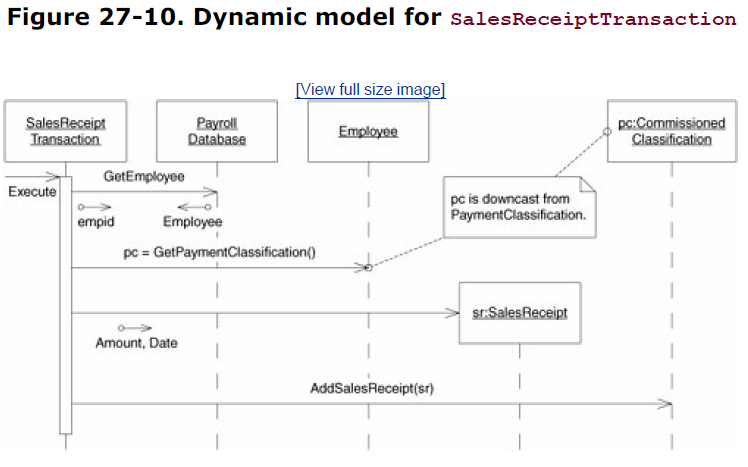


### Sales Receipts

#### Static model



#### Dynamic model



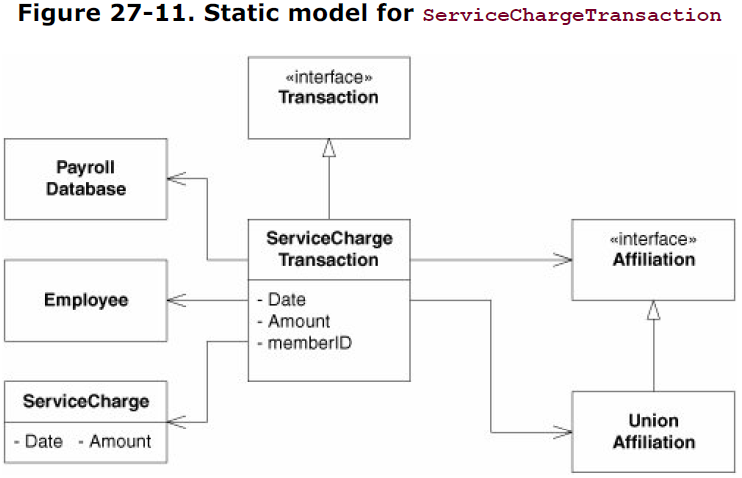
### Service Charges

This is the design for the transaction that posts service charges to union members.

These designs point out a mismatch between the transaction model and the core model that we have created.

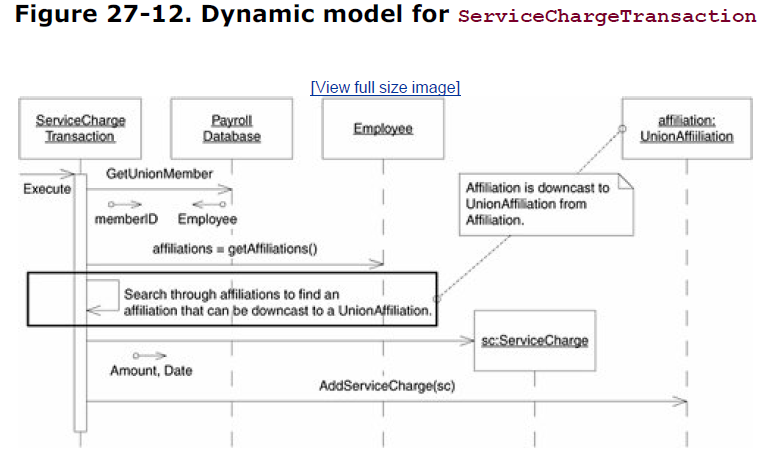
Our core Employee object can be affiliated with many different organizations, but the transaction model assumes that any affiliation must be a union affiliation. Thus, the transaction model provides no way to identify a particular kind of affiliation. Instead, it simply assumes that if we are posting a service charge, the employee has a union affiliation.

#### Static model



#### Dynamic model

The dynamic model addresses this dilemma by searching the set of Affiliation objects contained by the Employee object for a UnionAffiliation object. The model then adds the ServiceCharge object to that UnionAffiliation. It is indeed much simpler without the loop looking for UnionAffiliation objects. It simply gets the Employee from the database, downcasts its Affillation to a UnionAffilliation, and adds the ServiceCharge to it.

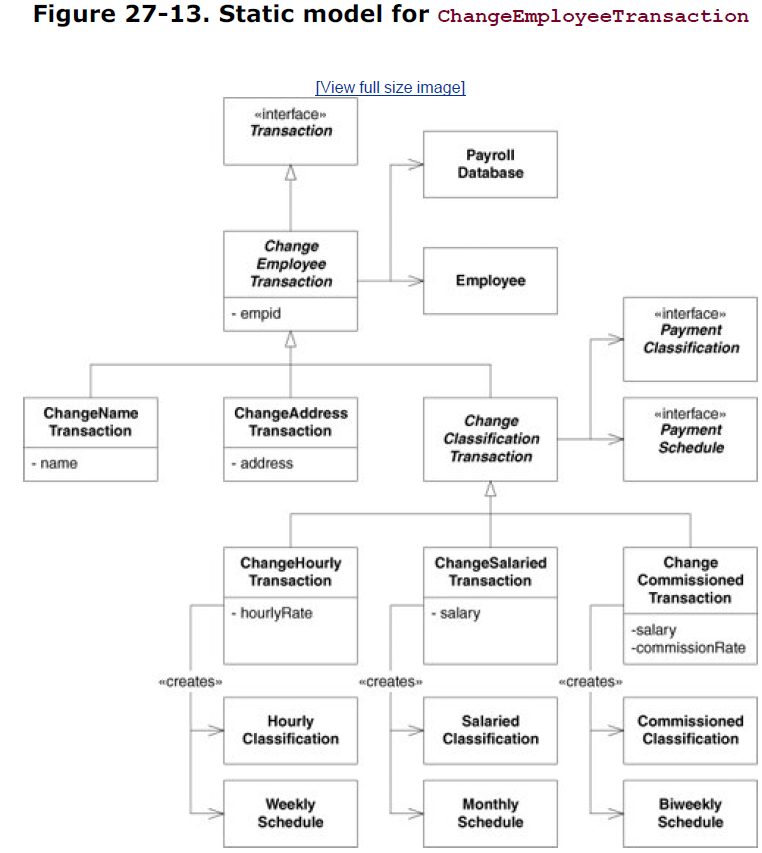
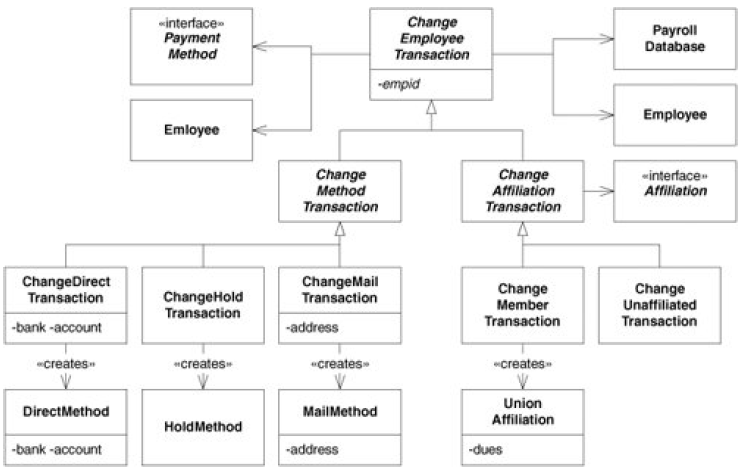


### Changing Employees

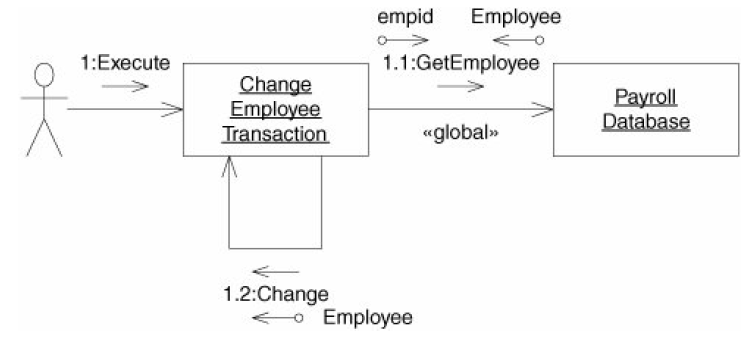
All the transactions take an EmpID argument, so we can create a top-level base class called Change-EmployeeTransaction.

Below this base class are the classes that change single attributes, such as ChangeNameTransaction and ChangeAddressTransaction. The transactions that change classifications have a commonality of purpose in that they all modify the same field of the Employee object. Thus, they can be grouped together under an abstract base, ChangeClassificationTransaction. The same is true of the transactions that change the payment and the affiliations. This can be seen by the structure of ChangeMethodTransaction and ChangeAffiliationTransaction.

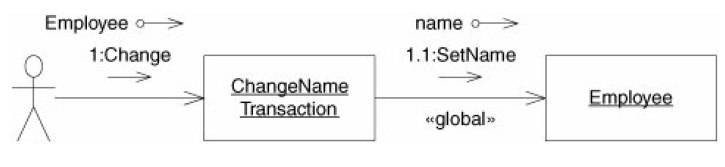
#### Static model

#### Dynamic model for ChangeEmployeeTransaction



#### Dynamic model for ChangeNameTransaction

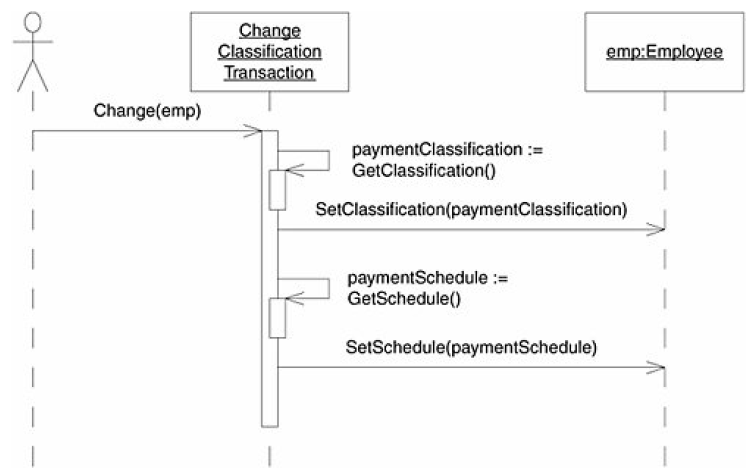


#### Dynamic model for ChangeAddressTransaction

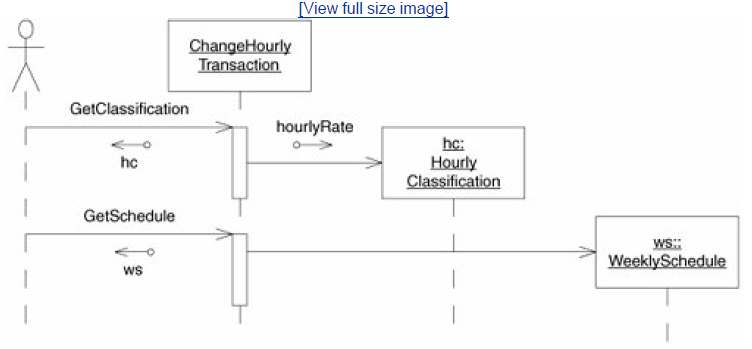


### Changing Payment Classification

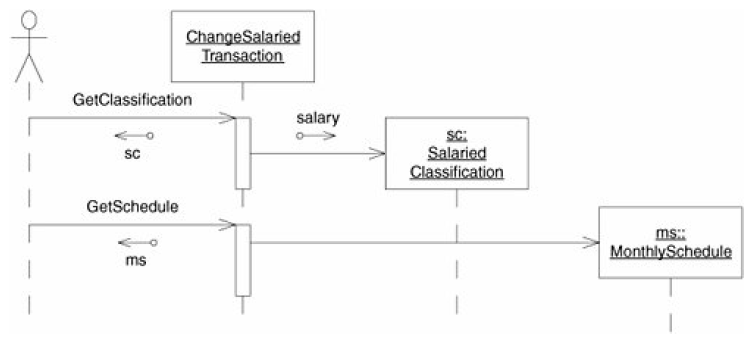
#### Dynamic model for ChangeClassificationTransaction



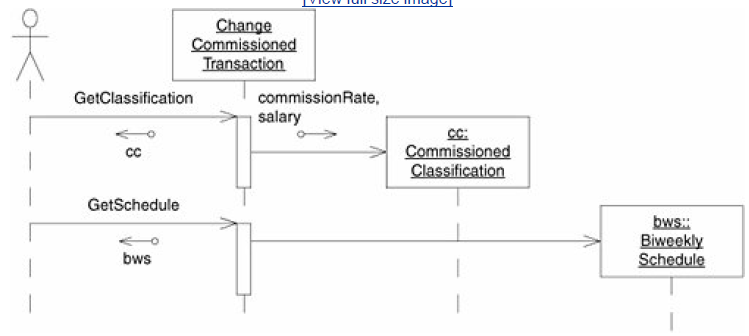
#### Dynamic model for ChangeHourlyTransaction



#### Dynamic model for ChangeSalariedTransaction

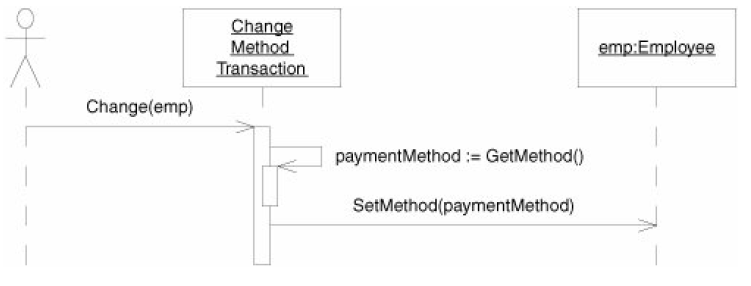


#### Dynamic model for ChangeCommissionedTransaction

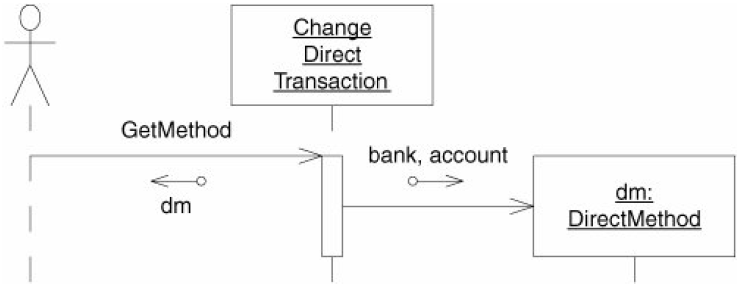


### Changing Payment Method

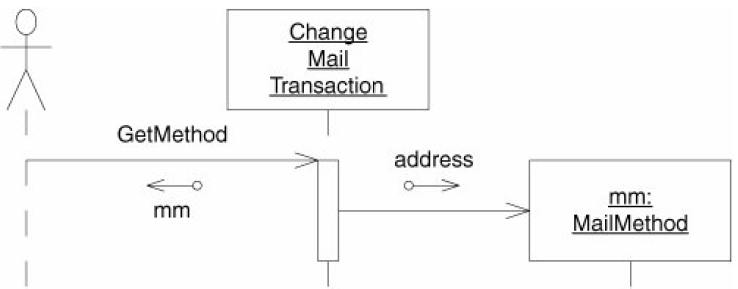
#### Dynamic model for ChangeMethodTransaction



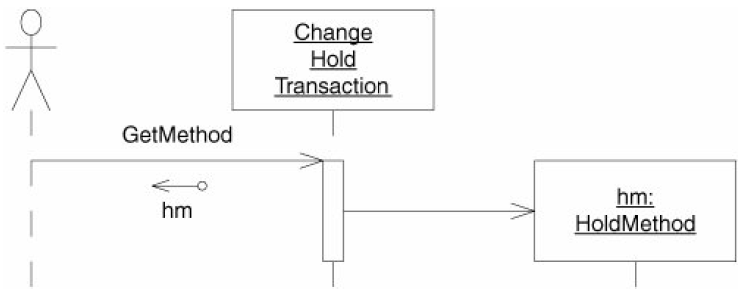
#### Dynamic model for ChangeDirectTransaction



#### Dynamic model for ChangeMailTransaction



#### Dynamic model for ChangeHoldTransaction



### Change Affiliation

The surprise is hidden in the last few lines of the test case. Those lines make sure that the PayrollDatabase has recorded Bill's membership in the union.

Nothing in the existing UML diagrams makes sure that this happens. The UML is concerned only with the appropriate Affiliation derivative being bound to the Employee.

I didn't notice the deficit at all. Did you?

How do I get the membership to be recorded by ChangeMemberTransaction but erased by ChangeUnaffiliatedTransaction?

The answer was to add to ChangeAffiliationTransaction another abstract method, named RecordMembership(Employee).

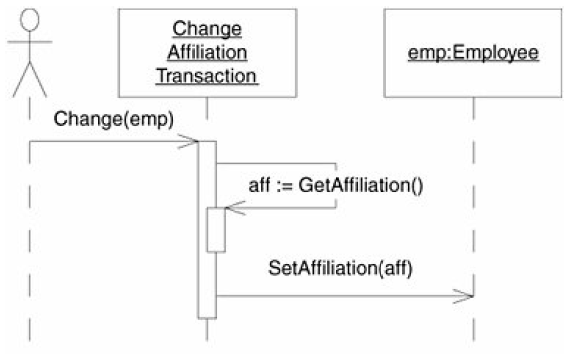
This function is implemented in ChangeMemberTransaction to bind the memberId to the Employee instance. In the ChangeUnaffiliatedTransaction, it is implemented to erase the membership record.

The RecordMembership function has to decide whether the current employee is a union member. If so, it gets the memberId from the UnionAffiliation and erases the membership record.

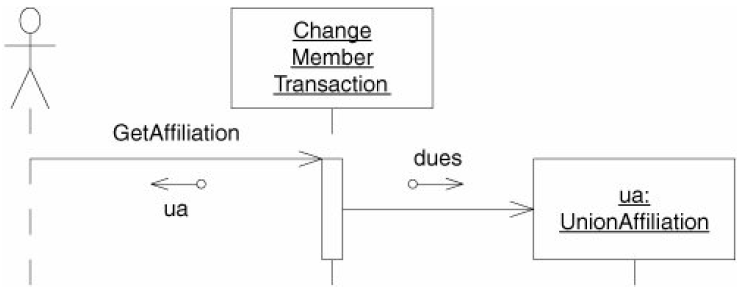
I can't say that I'm very pleased with this design. It bothers me that the ChangeUnaffiliatedTransaction must know about UnionAffiliation. I could solve this by putting RecordMembership and EraseMembership abstract methods in the Affiliation class. However, this would force UnionAffiliation and NoAffiliation to know about the PayrollDatabase. And I'm not very happy about that, either. (**Here can be used the VISITOR pattern to solve this problem, but that would probably be way overengineered**)

Still, the implementation as it stands is pretty simple and violates OCP only slightly. The nice thing is that very few modules in the system know about ChangeUnaffiliatedTransaction, so its extra dependencies aren't doing very much harm.

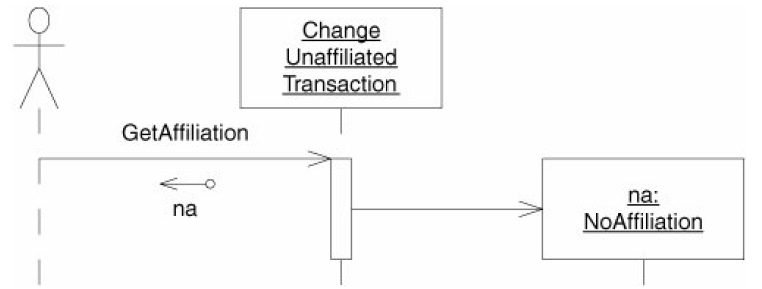
#### Dynamic model for ChangeAffiliationTransaction



#### Dynamic model for ChangeMemberTransaction

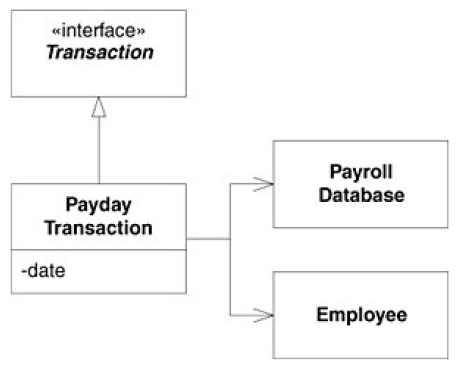


#### Dynamic model for ChangeUnaffiliatedTransaction

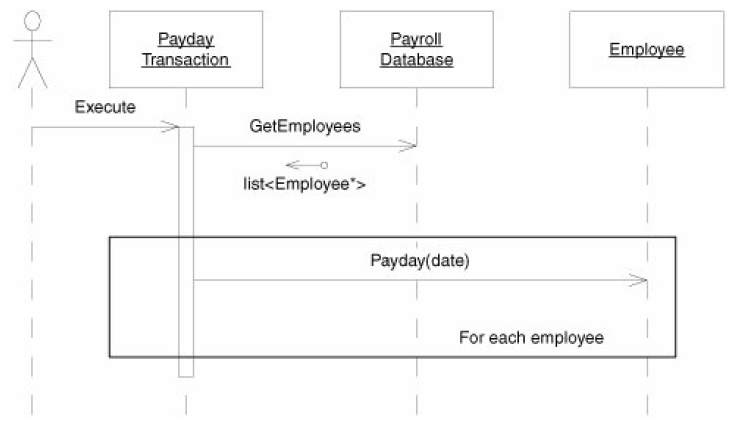


### Paying Employees

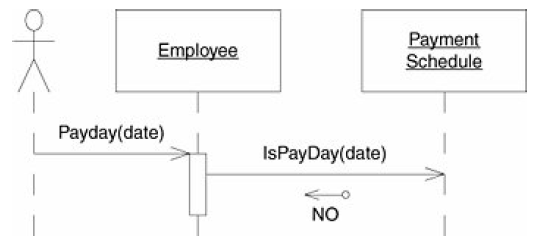
#### Static model for PaydayTransaction



#### Dynamic model for PaydayTransaction



##### Dynamic model scenario: "Payday is not today."



The dynamic models express a great deal of polymorphic behavior.

The algorithm used by the CalculatePay message depends on the kind of PaymentClassification that the Employee object contains. The algorithm used to determine whether a date is a payday depends on the kind of PaymentSchedule that the Employee contains. The algorithm used to send the payment to the Employee depends on the type of the PaymentMethod object.

**This high degree of abstraction allows the algorithms to be closed against the addition of new kinds of payment classifications, schedules, affiliations, or payment methods**.

##### Dynamic model scenario: "Payday is today."

The algorithms depicted in Figure 27-31 and Figure 27-32 introduce **the concept of *posting***. After the correct pay amount has been calculated and sent to the Employee, the payment is posted; that is, the

records involved in the payment are updated. Thus, we can define the CalculatePay method as calculating the pay from the last posting until the specified date.

Where did this notion of posting come from?

**It certainly wasn't mentioned in the user stories or the use cases**. As it happens, I cooked it up as a way to solve a problem that I perceived. I was concerned that the Payday method might be called multiple times with the same date or with a date in the same pay period, so I wanted to make sure that the employee was not paid more than once. I did this on my own initiative, without asking my customer. **It just seemed the right thing to do**.

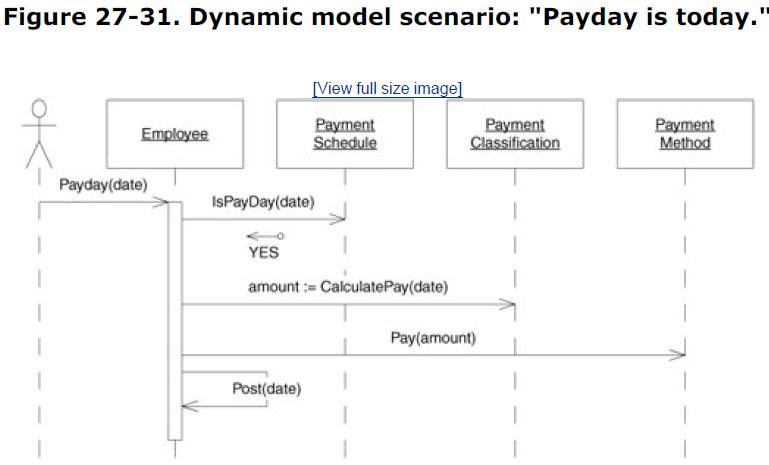
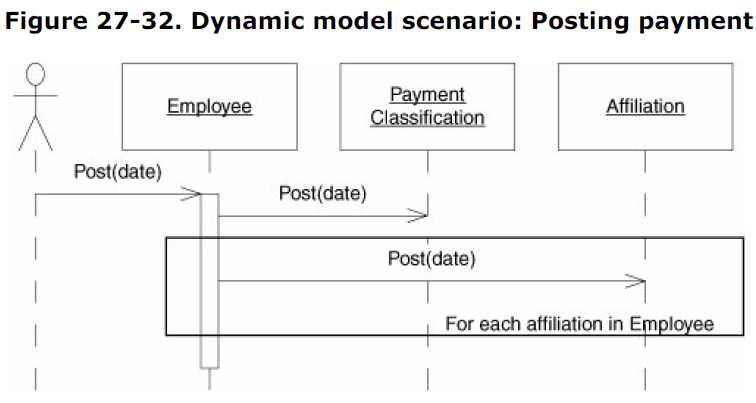
**In effect, I made a business decision**, deciding that multiple runs of the payroll program should produce different results. I should have asked my customer or project manager about this, since they might have very different ideas. In checking with the customer, I find that the idea of posting goes against his intent.

**The customer wants to be able to run the payroll system and then review the paychecks.**

**If any of them are wrong, the customer wants to correct the payroll information and run the payroll program again.**

**The customer tells me that I should never consider time cards or sales receipts for dates outside the current pay period.**

**So, we have to ditch the posting scheme. It seemed like a good idea at the time, but it was not what the customer wanted.**

#### Paying Salaried Employees

Test whether a salaried employee is being paid appropriately:

* The first test case makes sure that the employee is paid on the last day of the month.
* The second test case makes sure that the employee is not paid if it is not the last day of the month.

The PaydayTransaction. iterates through all the Employee objects in the database, asking each employee if the date on this transaction is its pay date. If so, it creates a new paycheck for the employee and tells the employee to fill in its fields.

#### Paying Hourly Employees

The first tests whether we can pay an employee after adding a single time card. The second tests whether we can pay overtime for a card that has more than 8 hours on it.

# 5. The Database

In previous chapters, we implemented all the business logic for the payroll application.

That implementation had a class, PayrollDatabase, that stored all the payroll data in RAM.

This chapter 37 explains how to provide that persistence by storing the data in a relational database.

## Building the Database

The choice of database technology is usually made more for political reasons than for technical reasons. So you should not read too much into our choice of Microsoft SQL Server to persist the data for our application.

The schema that we'll be using is shown in Figure 37-1.

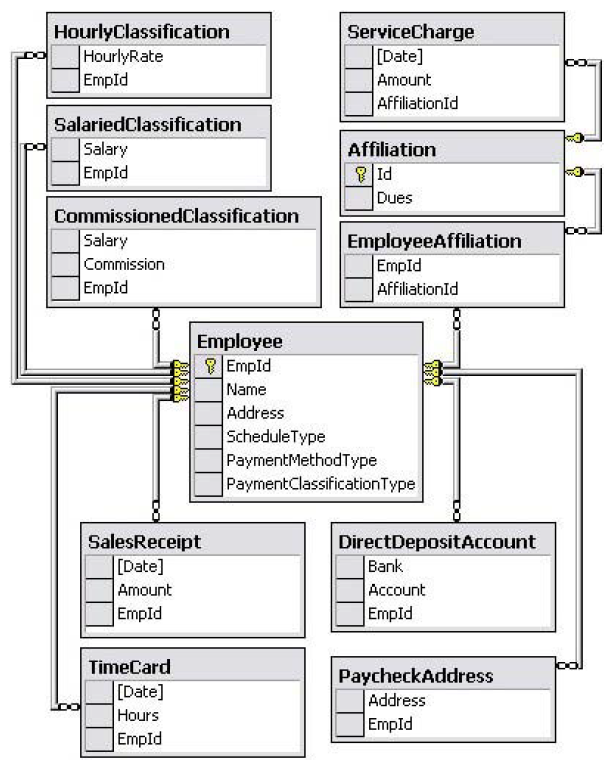
The Employee table is central. It stores the immediate data for an employee, along with string constants that determine the PaymentSchedule, PaymentMethod, and PaymentClassification.

PaymentClassifications have data of their own that will be persisted in the corresponding HourlyClassification, SalariedClassification, and CommissionedClassification tables. Each references the Employee it belongs to, via the EmpId column. This column has a constraint to make sure that an Employee record with the given EmpId exists in the Employee table.

DirectDepositAccount and PaycheckAddress hold the data appropriate to their PaymentMethod and are likewise constrained by the EmpId column. SalesReceipt and TimeCard are straightforward.

The Affiliation table holds such data as the union members and is linked to the Employee table via EmpoyeeAffiliation.

### Figure 37-1. Payroll schema



## Code Design

How do we start using a real database in the code without breaking all the tests that use the static methods? We don't want to overwrite the PayrollDatabase class to use a real database. That would force all our existing unit tests to use the real database.

It would be nice if PayrollDatabase were an interface so we could easily swap out different implementations. One implementation would store data in memory like it does now, so that our tests can continue to run quickly. Another implementation would store data in a real database.

# Packaging and Manage component dependency structure (+6 OOD Principles)

Large programs can become opaque masses of source files without some kind of partitioning structure. The principles and metrics described in this chapter have helped

me, and many other development teams, manage their component dependency structures.

## Principles of Component Cohesion: Granularity

In the past, our view of cohesion was much simpler. We used to think that cohesion was simply the attribute of a module to perform one, and only one, function. However, the three principles of component cohesion describe a much more complex kind of cohesion. In choosing the classes to group together into a component, we must consider the opposing forces involved in reusability and developability.

Balancing these forces with the needs of the application is nontrivial. Moreover, the balance is almost always dynamic. That is, the partitioning that is appropriate today might not be appropriate next year. Thus, the composition of the component will likely jitter and evolve with time as the focus of the project changes from developability to reusability.

* **Reuse/Release Equivalence Principle (REP)** - The granule of reuse is the granule of release.
* **Common Reuse Principle (CRP)** - The classes in a component are reused together. If you reuse one of the classes in a component, you reuse them all.
* **Common Closure Principle (CCP)** - The classes in a component should be closed together against the same kinds of changes. A change that affects a component affects all the classes in that component and no other components.

## Principles of Component Coupling: Stability

* **Stable-Dependencies Principle (SDP)** - Depend in the direction of stability.
* **Stable-Abstractions Principle (SAP)** - component should be as abstract as it is stable.
* **Acyclic Dependencies Principle (ADP)** - Allow no cycles in the component dependency graph.

## Metrics Spreadsheet

As we showed in Chapter 28, we can quantify the attributes of cohesion, coupling, stability.

But why should we want to? You can't manage what you can't control, and you can't control what you don't measure.[1] To be effective software engineers or software managers, we must be able to control software development practice. If we don't measure it, however, we will never have that control.

### Stability metrics

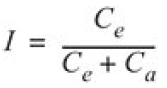
***Ca* (*afferent coupling*)** can be calculated as the number of classes from other components that depend on the classes within the subject component. These dependencies are class

relationships, such as inheritance and association.

***Ce* (*efferent coupling*)** can be calculated as the number of classes in other components that the classes in the subject component depend on. As before, these dependencies are class relationships.

***I* (*instability*)** can be calculated as the ratio of efferent coupling to total coupling. This metric

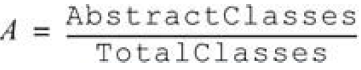
also ranges from 0 to 1.



### Measuring abstraction

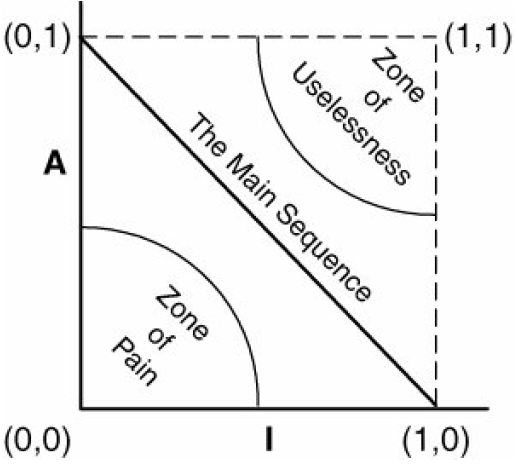
***A* (*abstractness*, or *generality*)** can be calculated as the ratio of the number of abstract classes or interfaces in the component to the total number of classes and interfaces in the

component. This metric ranges from 0 to 1.



We are now in a position to define the relationship between stability (*I*) and abstractness (*A*). We can create a graph with *A* on the vertical axis and *I* on the horizontal axis. If we plot the two "good" kinds of components on this graph, we will find the components that are maximally stable and abstract at the upper left at (0,1). The components that are maximally instable and concrete are at the lower right at (1,0). We can infer what that locus is by finding the areas where components should *not* be, that is, zones of *exclusion*.

**Zones of exclusion**

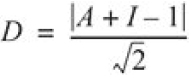


### Distance from the main sequence

If it is desirable for components to be on or close to the main sequence, we can create a metric that measures how far away a component is from this ideal. Given this metric, a design can be analyzed for its overall conformance to the main sequence. The *D* metric for each component can be calculated. Any component that has a *D* value that is not near 0 can be reexamined and restructured. In fact, this kind of analysis has been a great aid to me in

helping to define components that are more maintainable and less sensitive to change.

***D*****(distance from the main sequence)** = |(*A* + *I* -1) ÷ D2|**.** The main sequence is idealized by the line *A* + *I* = 1. The formula calculates the distance of any particular component from the main sequence. It ranges from ~.7 to 0; the closer to 0, the better.[



***D'* (normalized distance from the main sequence)** represents the *D* metric normalized to the range [0,1]. It is perhaps a little more convenient to calculate and to interpret. The value 0

represents a component that is coincident with the main sequence. The value 1 represents a

component that is as far from the main sequence as is possible.

*D'*= |*A* + *I* - 1|

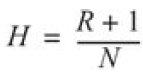
***H* (*relational cohesion*)** can be represented as the average number of internal relationships per

class in a component. Let *R* be the number of class relationships that are internal to the

component (i.e., that do not connect to classes outside the component. Let *N* be the number of

classes within the component). The extra 1 in the formula prevents *H* = 0 when *N* = 1 and

represents the relationship that the package has to all its classes.



## Class Allocation to Component

|  |  |  |  |
| --- | --- | --- | --- |
| **Component** | **Classes in Component** | | |
| **AbstractTransactions** | AddEmployeeTransaction ChangeClassificationTransaction | ChangeAffiliationTransaction | ChangeMethodTransaction |
| **Application** | Application |  |  |
| **PayrollApplication** | PayrollApplication |  |  |
| **PayrollDatabase** | PayrollDatabase |  |  |
| **PayrollDatabase-Implementation** | PayrollDatabase-Implementation |  |  |
| **PayrollDomains** | Affiliation PaymentMethod | Employee PaymentSchedule | PaymentClassification |
| **PayrollFactory** | **PayrollFactory** |  |  |
| **Payrollimplementation** | BiweeklySchedule MonthlySchedule WeeklySchedule  **PayrollFactoryImplementation** | DirectMethod HoldMethod MailMethod UnionAffiliation ServiceCharge | SalesReceipt TimeCard SalariedClassification HourlyClassification CommissionedClassification |
| **TextParser-TransactionSource** | TextParserTransactionSource |  |  |
| **Transaction-Application** | TransactionApplication | Transaction | TransactionSource |
| **TransactionFactory** | **TransactionFactory** |  |  |
| **TransactionImplementation** | AddCommisionedEmployee AddSalariedEmployee ChangeNameTransaction  ChangeAddressTransaction DeleteEmployeeTransaction  AddHourlyTransaction ChangeEmployeeTransaction PaydayTransaction | ChangeSalariedTransaction  ChangeCommisionedTransaction SalesReceiptTransaction  ChangeHourlyTransaction TimeCardTransaction | ChangeDirectTransaction  ChangeHoldTransaction  ChangeMailTransaction ServiceChangeTransaction ChangeUnaffiliatedTransaction ChangeMemberTransaction |

Merge the transactions into a single TRansactionImplementation component. We will also merge the Classifications, Schedules, Methods, and Affiliations components into a single

PayrollImplementation package.

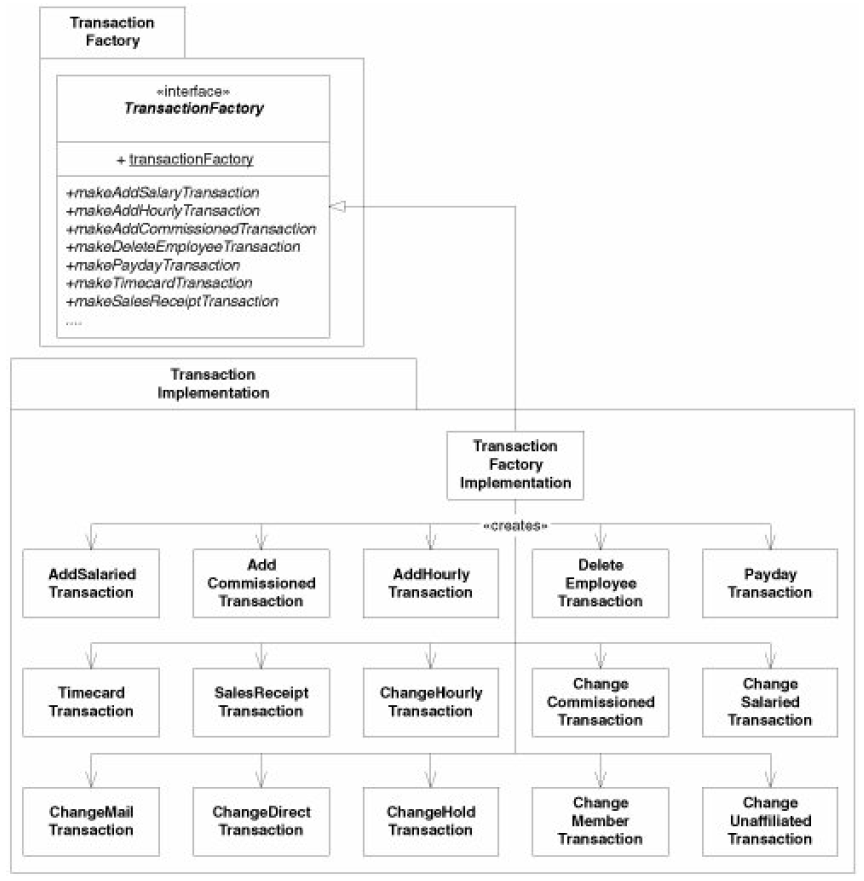
### Object Factories

Classifications and ClassificationTransaction are so heavily depended on because the classes within them must be instantiated. This problem can be significantly mitigated by using the FACTORY pattern. Each component provides an object factory that is responsible for creating all the public objects within that package.

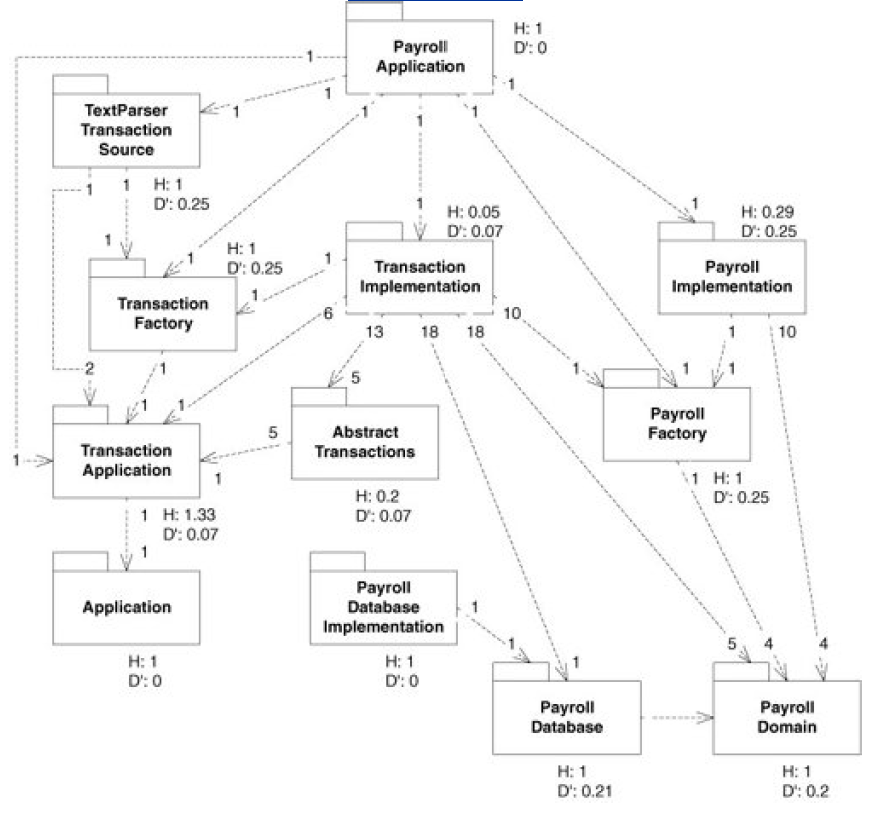
The transactionFactory class has a static member declared as a transactionFactory pointer. This member must be initialized by the main program to point to an instance of the concrete transactionFactoryImplementation object.

If other factories are to create objects using the object factories, the static members of the abstract object factories must be initialized to point to the appropriate concrete factory. This must be done before any user attempts to use the factory. The best place to do this is usually the main program, which means that the main program depends on all the factories *and* on all the concrete packages.

#### Object factory for transactions

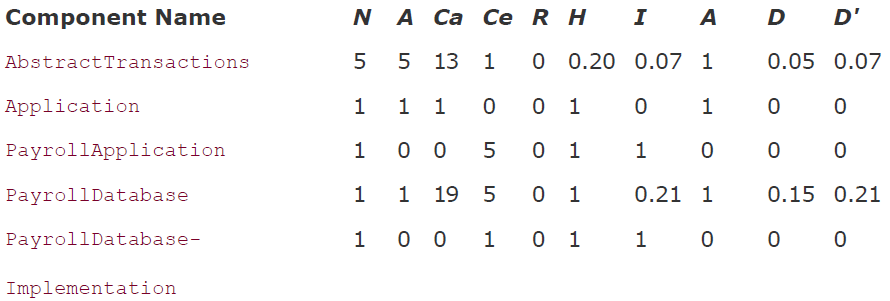


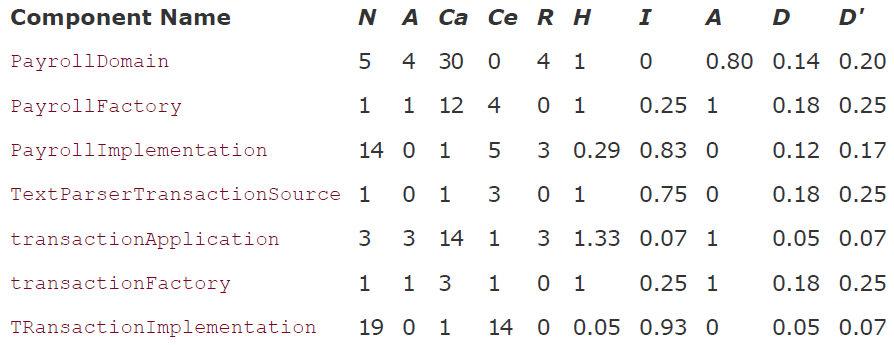
### Final payroll component structure



Our abstract components are closed, reusable, and heavily depended on but have few dependencies of their own. Our concrete components are segregated on the basis of reuse, are heavily dependent on the abstract components, and are not heavily depended on themselves.

### Applying the Metrics to the Payroll Application





# Agile

Manifesto for Agile Software Development

Principles behind the Agile Manifesto

Practices of Extreme Programming

The Principles of Object Oriented Design (11)

# OOD Principles Chapters (first 5 from 11)

1. Chapter 8. The Single-Responsibility
2. Chapter 9. The Open/Closed Principle
3. Chapter 10. The Liskov Substitution
4. Chapter 11. The Dependency-Inversion
5. Chapter 12. The Interface Segregation

# Using UML

Chapter 13. Overview of UML for C#

Chapter 14. Working with Diagrams

# Design Pattern Chapters

Chapter 21. COMMAND and ACTIVE OBJECT

Chapter 22. TEMPLATE METHOD and STRATEGY

Chapter 23. Facade and Mediator

Chapter 24. Singleton and Monostate

Chapter 25. Null Object

Chapter 29. Factory

Chapter 31. Composite

Chapter 32. Observer: Evolving into a Pattern

Chapter 33. Abstract Server, Adapter, and Bridge

Chapter 34. PROXY and GATEWAY: Managing Third-Party APIs

Chapter 35. Visitor

Chapter 36. State

Chapter 38. The Payroll User Interface

# Notes

In order to test a module, you have to be able to isolate it from the other modules in

the system, just as we have isolated the ClockDriver from the Clock and DigitalClock. Considering tests first helps us to minimize the coupling in our designs.

CPU hog problem – occurs when code consumes all available CPU cycles to repeatedly display the time. E.g.

public void DisplayTime()

{

while (true)

{

int sec = clock.Seconds;

int min = clock.Minutes;

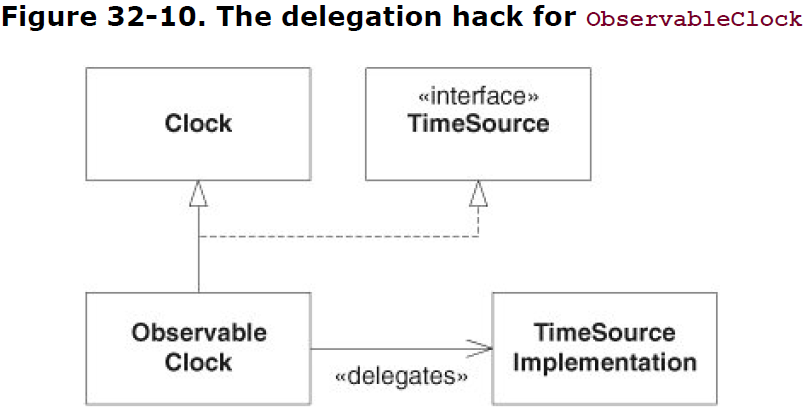
int hour = clock.Hours;

ShowTime(hour, min, sec);

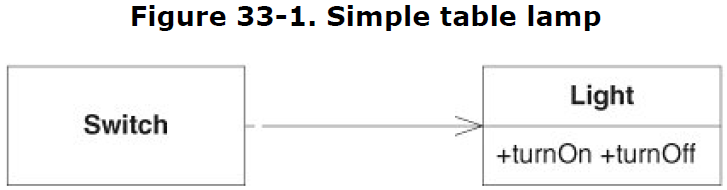
}

}

**The delegation hack** – is a delegation of functionality from base class to reduce code dependency in child class. This solves the problem of New-child class depending on the registration and update code part but does so at a nontrivial price.



## ABSTRACT SERVER pattern



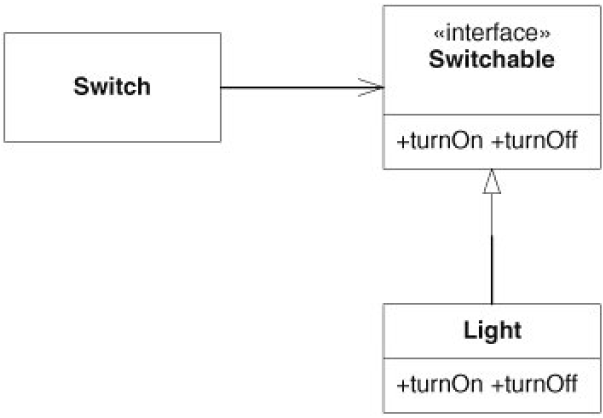
The violation of DIP is easy to see; the dependency from Switch to Light is a dependency on a concrete class. DIP tells us to prefer dependencies on abstract classes.

The violation of OCP is a little less direct but is more to the point. We don't like this design, because it forces us to drag a Light along everywhere we need a Switch. Switch cannot be easily extended to control objects other than Light.

By introducing an interface between the Switch and the Light, we have made it possible for Switch to control anything that implements that interface. This immediately satisfies both DIP and OCP.

**Figure 33-3. ABSTRACT SERVER solution to the table lamp problem**

As an interesting aside, note that the interface is named for its client. It is called Switchable rather than Light. **Clients tend to be packaged with the interfaces they control.**



What if we can't add the inheritance relationship to Light?

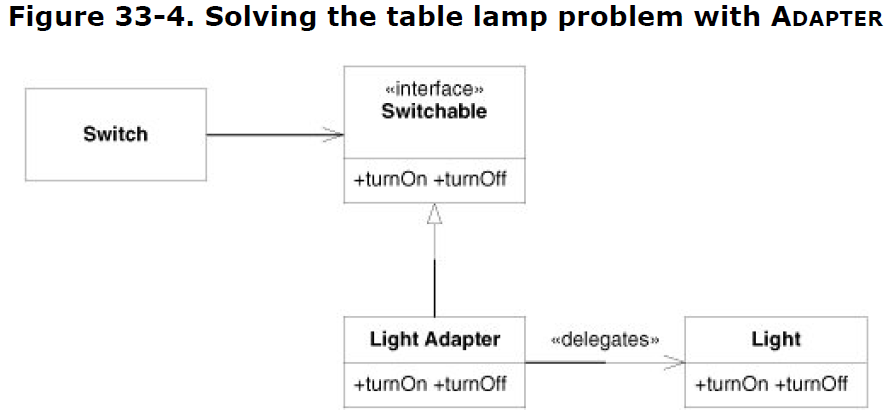
What if we purchased Light from a third party and don't have the source code?

What if we want a Switch to control a class that we can't derive from Switchable?

Enter the ADAPTER.

## ADAPTER pattern

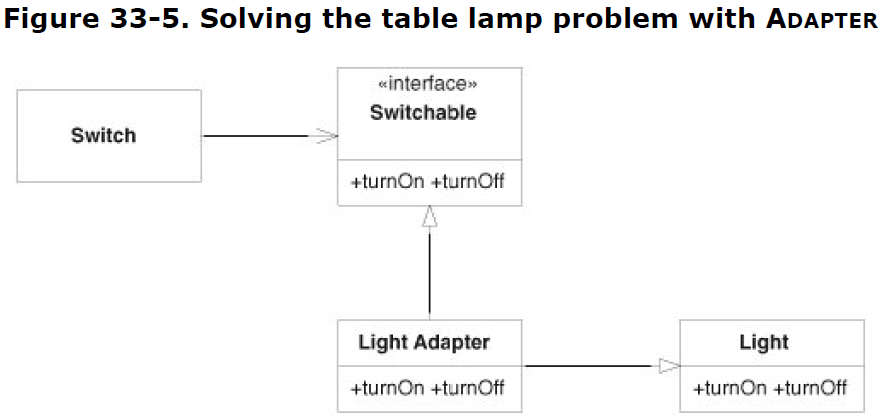
The adapter derives from Switchable and delegates to Light. Now we can have any object that can be turned on or off controlled by a Switch. All we need to do is create the appropriate adapter. Indeed, the object need not even have the same turnOn and turnOff methods that Switchable has. The adapter can be *adapted* to the interface of the object.



**Adapters don't come cheap.** You need to write the new class, and you need to instantiate the adapter and bind the adapted object to it. Then, every time you invoke the adapter, you have to pay for the time and space required for the delegation. **So clearly, you don't want to use adapters all the time.**

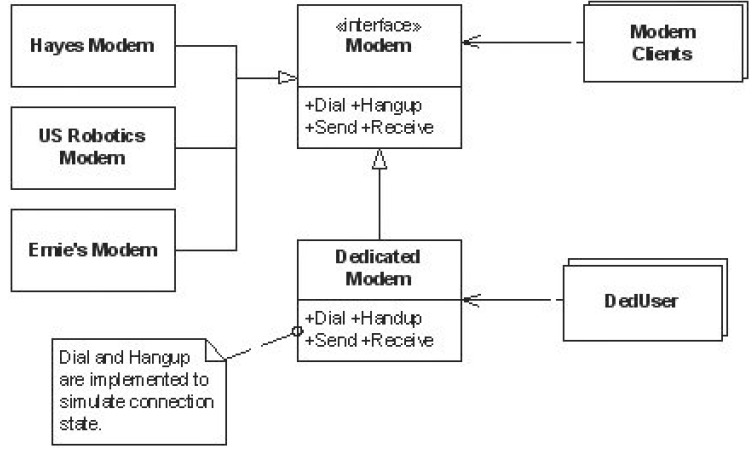
**The ABSTRACT SERVER solution is quite appropriate for most situations.** In fact, even the initial solution in Figure 33-1 is pretty good **unless you happen to *know* that there are other objects** for Switch to control.

**Another approach, known as the *class-form adapter***, is shown in Figure 33-5. In this form, the adapter object inherits from both the Switchable interface and the Light class. This form is a tiny bit more efficient than the object form and is a bit easier to use but at the expense of using the high coupling of inheritance.



### Modem problem

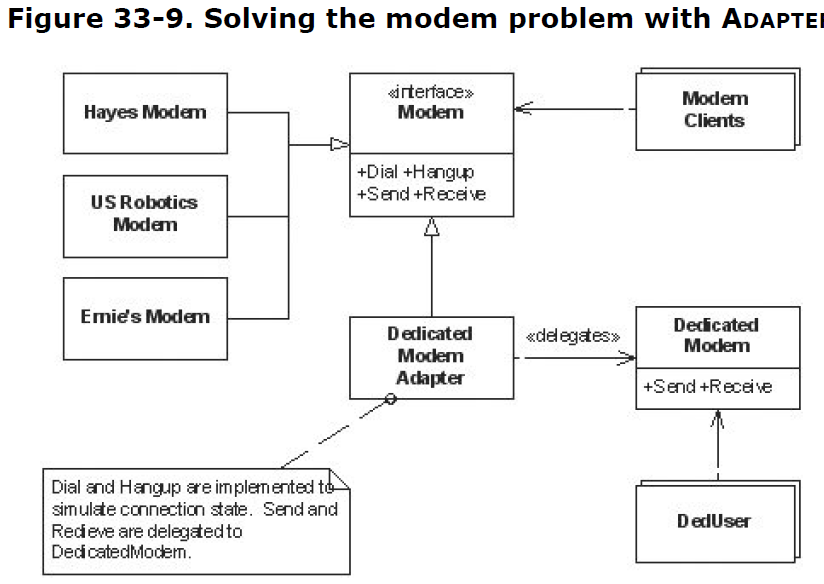
**Figure 33-8. Kludging DedicatedModem to simulate connection state**



Clearly, all the modem clients must be changed. They were written to expect char[10] for the phone number. Our customers authorize this change because they have no choice, and hordes of programmers are put to the task. Just as clearly, the classes in the modem hierarchy must change to accommodate the new phone number size. Our little team can deal with that. *Unfortunately, we now have to go to the authors of the DedUsers and tell them that they have to change their code!*

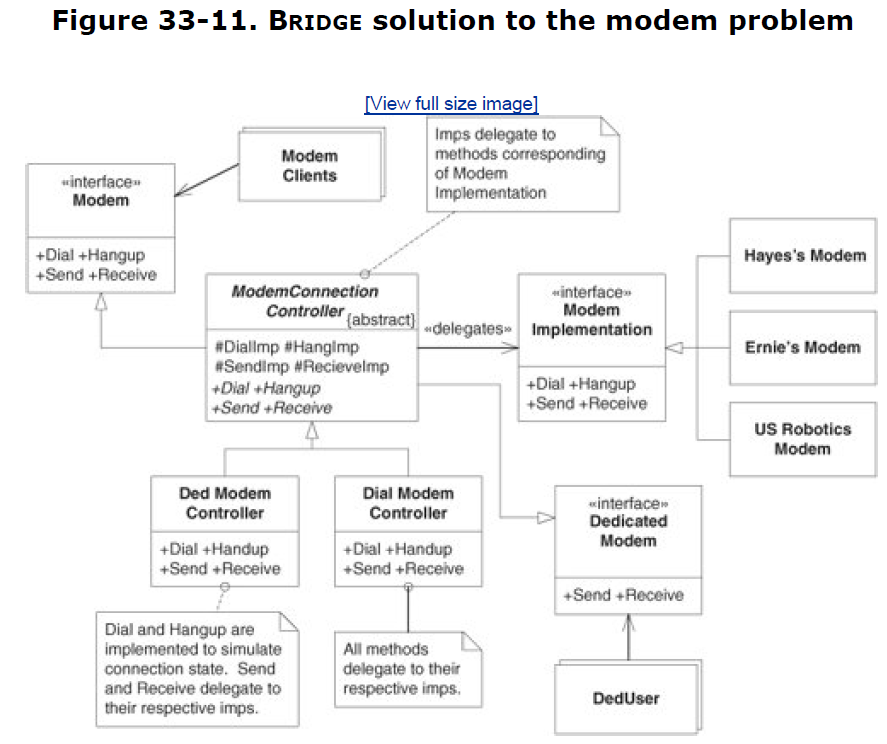
**Adding adapter that simulating connection state.**

Note that this eliminates all the difficulties we had before. Modem clients are seeing the connection behavior that they expect, and DedUsers don't have to fiddle with dial or hangup. When the phone number requirement changes, the DedUsers will be unaffected. Thus, by putting the adapter in place, we have fixed both LSP and OCP violations.



## Bridge

We split the modem hierarchy into two hierarchies. One represents the connection method, and the other represents the hardware. The Bridge pattern attempts to solve this problem by switching from inheritance to composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.



There is no escape from situation when customer introduces a change

that violates perfect software structure.. There are no perfect structures. There are only structures that try to balance the current costs and benefits. Over time, those structures must change as the requirements of the system change. The trick to managing that change is to keep the system as simple and as flexible as possible.

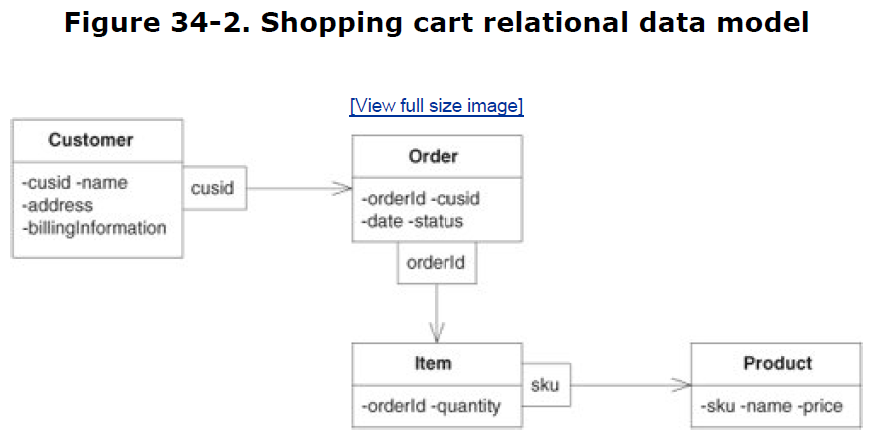
The ADAPTER solution is simple and direct. It keeps all the dependencies pointing in the right direction, and it's very simple to implement. The BRIDGE solution is quite a bit more complex.

I would not suggest embarking down that road until you had very strong evidence that you needed to completely separate the connection and communication policies and that you needed to add new connection policies.

The lesson here, as always, is that **a pattern is something that comes with both costs and benefits**. You should find yourself using the ones that best fit the problem at hand.

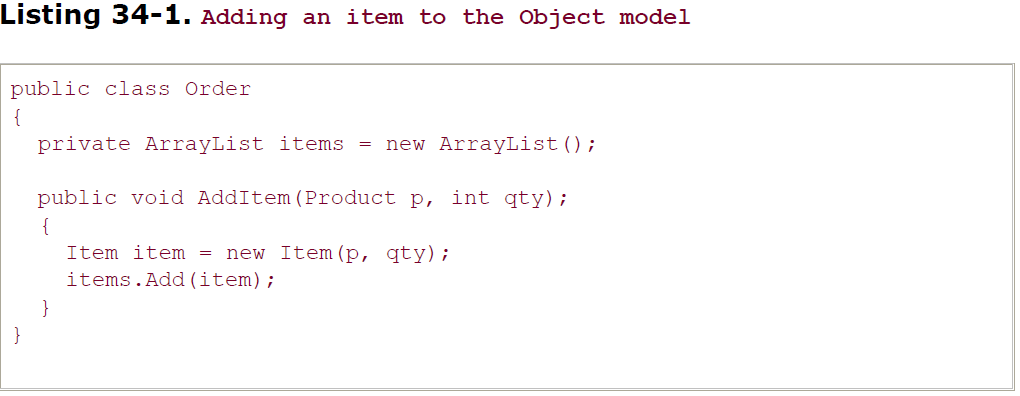
## Chapter 34. PROXY and GATEWAY: Managing Third-Party APIs



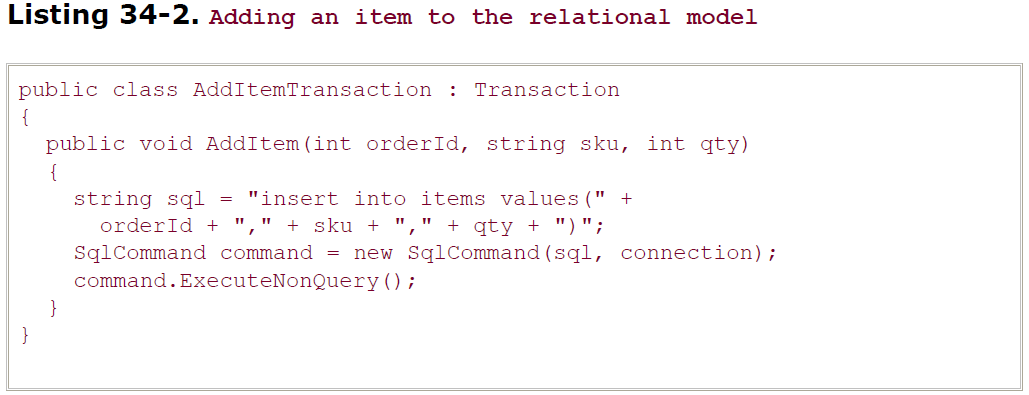


Clearly, the shopping cart program is all about orders, items, and products.

If we consider the problem of adding a new item to an order, we might come up with the code in Listing 34-1.



If we want to add an item row for a particular order, we'd use something like Listing 34-2. This code makes ADO.NET calls to directly manipulate the relational data model.



These two code snippets are very different, but they perform the same logical function. The first ignores the existence of a database, and the second glories in it.

**This is a significant violation of SRP and possibly CCP.** It mixes the concept of the items and orders with the concept of relational schemas and SQL. If either concept must change for any reason, the other concept will be affected.

**Аlso violates DIP**, since the policy of the program depends on the details of the storage mechanism.

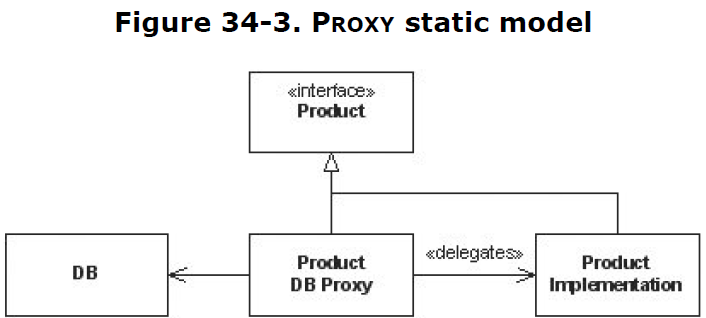
### Proxy

The PROXY pattern is a way to cure these ills.

Each object to be proxied is split into three

parts.

* The first **is an interface** that declares all the methods that clients will want to invoke.
* The second is **an implementation** that implements those methods without knowledge of the database.
* The third is **the proxy** that knows about the database.



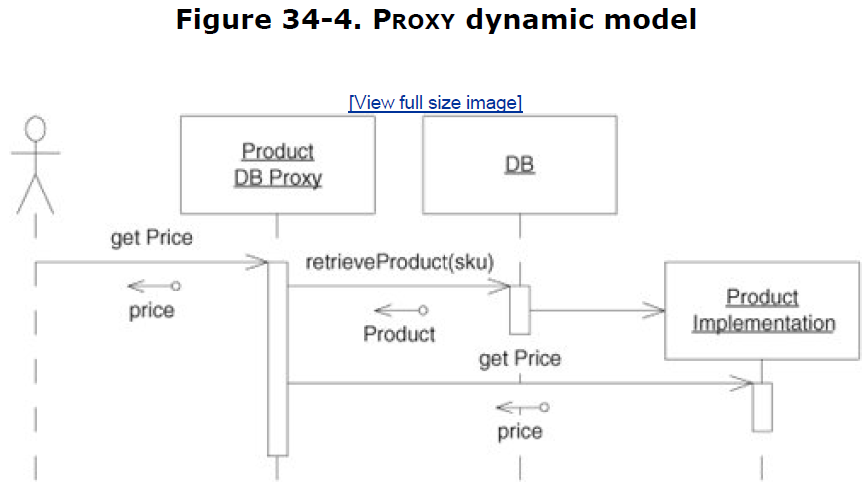
Consider the Product class. We have proxied it by replacing it with an interface.

This interface has all the same methods that Product has.

The ProductImplementation class implements the interface almost exactly as before.

The ProductDBProxy implements all the methods of Product to fetch the

product from the database, create an instance of ProductImplementation, and then delegate the message to it.



The client sends the Price message to what it thinks is a Product but what is in fact a ProductDBProxy. The ProductDBProxy fetches the ProductImplementation from the database and then delegates the Price property to it.

Neither the client nor the ProductImplementation knows that this has happened. The database has been inserted into the application without either party knowing about it. **That's the beauty of the PROXY pattern.** In theory, it can be inserted in between two collaborating objects without their having to know about it. Thus, it can be used to cross a barrier, such as a database or a network, without either participant knowing about it.

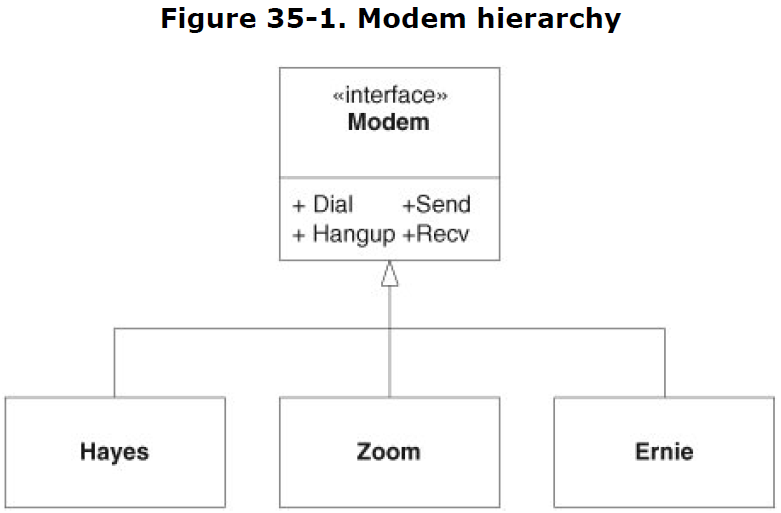
**The whole point of building proxies is to separate database implementation from business rules.** In reality, using proxies is nontrivial. To get an idea what some of the problems are, let's try to add the PROXY pattern to the simple shopping cart application.

## Chapter 35. Visitor

The VISITOR pattern gave us a way to add methods to existing hierarchies without changing those hierarchies.

The patterns[1] in this family are:

* ACYCLIC VISITOR
* DECORATOR
* EXTENSION OBJECT



The Modem interface contains the generic methods that all modems can implement.

How can we configure these modems for UNIX without putting the ConfigureForUnix

method in the Modem interface? We can use a technique called *dual dispatch*, the mechanism at the heart of the VISITOR pattern.

Having built this structure, new operating system configuration functions can be added by adding new derivatives of ModemVisitor without altering the Modem hierarchy in any way. So the VISITOR pattern substitutes derivatives of ModemVisitor for methods in the Modem hierarchy.

The two dispatches of VISITOR form a matrix of functions. In our modem example, one axis of the matrix is the various types of modems; the other axis, the various types of operating systems. Every cell in this matrix is filled in with **a function that describes how to initialize the particular modem for the particular operating system**.

### Uses of Visitor

#### Report generation

The VISITOR pattern is commonly used to walk large data structures and to generate reports.

**The value of the VISITOR in this case** is that the data structure objects do not have to have any reportgeneration code. **New reports can be added by adding new VISITORs** rather than by changing the code in the data structures. This means that reports can be placed in separate components and individually deployed only to those customers needing them.

#### Other uses

In general, the VISITOR pattern can be used **in any application having a data structure that needs to be interpreted in various ways**.

Many applications make use of configuration data structures. One could imagine the various

subsystems of the application initializing themselves from the configuration data by walking it with their own particular visitors.

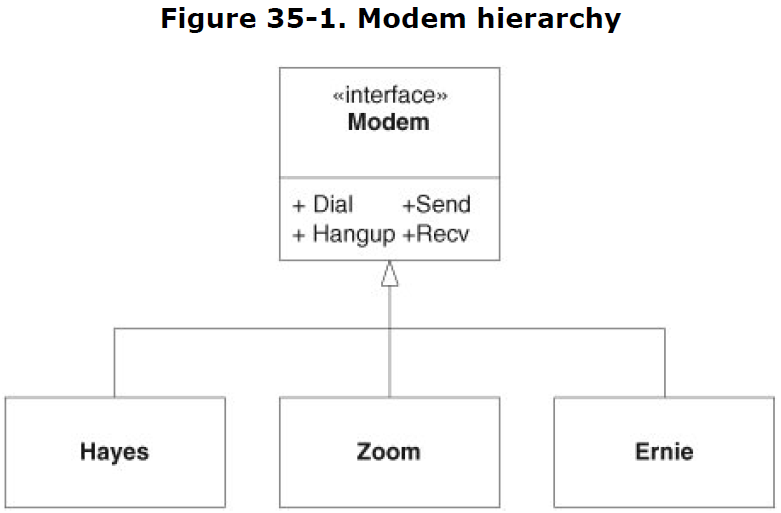
Whatever visitors are used, **the data structure being used is independent of the uses** to which it is being put. New visitors can be created, existing visitors can be changed, and all can be redeployed to installed sites **without the recompilation or redeployment of the existing data structures**. **This is the power of the VISITOR**.

### Decorator

The VISITOR pattern gave us a way to add methods to existing hierarchies without changing those hierarchies. Another pattern that accomplishes this is DECORATOR.

#### Problem description

Consider once again the Modem hierarchy in Figure 35-1.



Imagine that we have an application that has many users. Sitting at a computer, each user can ask the system to call out to another computer, using the computer's modem. Some of the users like to hear their modem's dial. Others like their modems to be silent.

We could implement this by querying the user preferences at every location in the code where the modem is dialed. If the user wants to hear the modem, we set the speaker volume high; otherwise, we turn it off:

...

Modem m = user.Modem;

if (user.WantsLoudDial())

m.Volume = 11; // it's one more than 10, isn't it?

m.Dial(...);

...

The specter of seeing this stretch of code duplicated hundreds of times throughout the application conjures images of 80-hour weeks and heinous debugging sessions. It is something to be avoided.

Another option would be to set a flag in the modem object itself and have the Dial method inspect it and set the volume accordingly:

...

public class HayesModem : Modem

{

private bool wantsLoudDial = false;

public void Dial(...)

{

if (wantsLoudDial)

{

Volume = 11;

}

...

}

...

}

This is better but must still be duplicated for every derivative of Modem. Authors of new derivatives of Modem must remember to replicate this code. Depending on programmers' memories is pretty risky business.

We could resolve this with the TEMPLATE METHOD[3] pattern by changing Modem from an interface to a class, having it hold the wantsLoudDial variable, and having it test that variable in the dial function before it calls the DialForReal function:

...

public abstract class Modem

{

private bool wantsLoudDial = false;

public void Dial(...)

{

if (wantsLoudDial)

{

Volume = 11;

}

DialForReal(...)

}

public abstract void DialForReal(...);

}

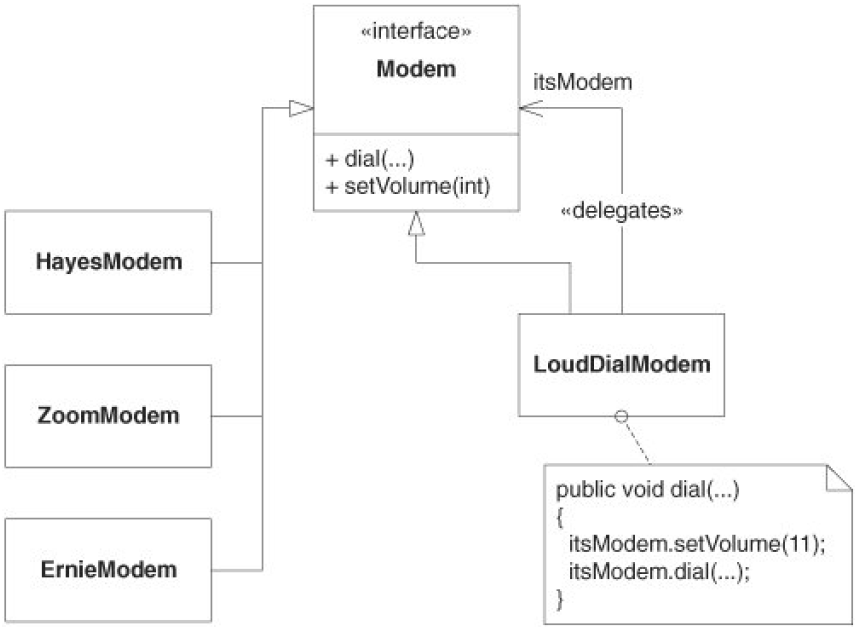
This is better still, but why should Modem be affected by the whims of the user in this way? Why should Modem know about loud dialing? Must it then be modified every time the user has some other odd request, such as logging out before hangup?

Once again, the **Common Closure Principle (CCP)** comes into play. We want to separate those things that change for different reasons. We can also invoke the **Single-Responsibility Principle (SRP)**, since the need to dial loudly has nothing to do with the intrinsic functions of Modem and should therefore not be part of Modem.

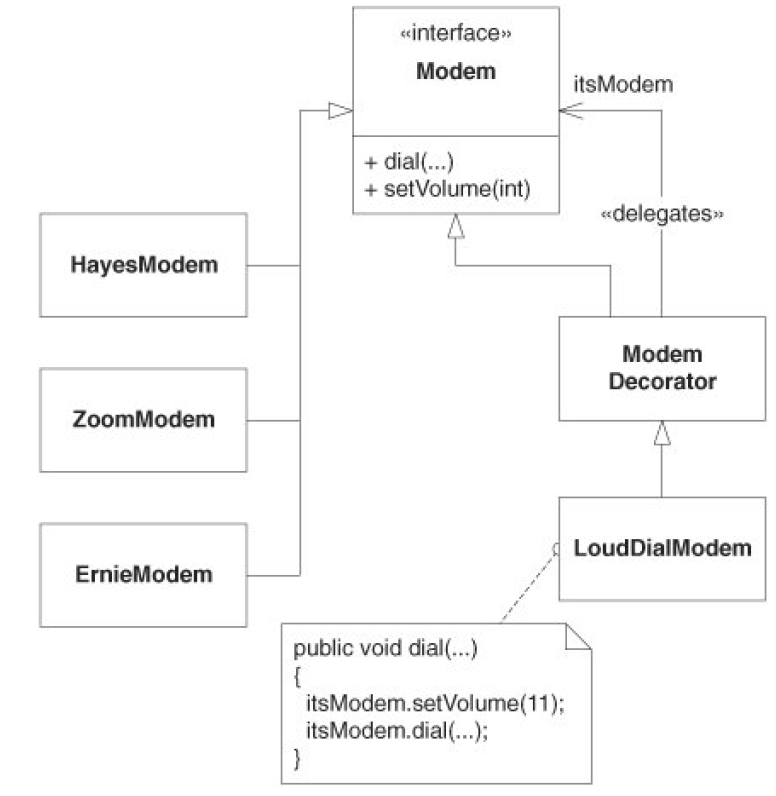
#### The problem solution

DECORATOR solves the issue by creating a completely new class: LoudDialModem. LoudDialModem derives from Modem and delegates to a contained instance of Modem, catching the Dial function and setting the volume high before delegating. Figure 35-5 shows the structure.

Now the decision to dial loudly can be made in one place. At the place in the code where the user sets preferences, a LoudDialModem can be created if loud dialing is requested, and the user's modem can be passed into it. LoudDialModem will delegate all calls made to it to the user's modem, so the user won't notice any difference. The Dial method, however, will first set the volume high before delegating to the user's modem. The LoudDialModem can then become the user's modem without anybody else in the system being affected.



Sometimes, two or more decorators may exist for the same hierarchy. For example, we may wish to decorate the Modem hierarchy with LogoutExitModem, which sends the string 'exit' whenever the Hangup method is called. This second decorator will have to duplicate all the delegation code that we have already written in LoudDialModem. We can eliminate this duplicate code by creating a new class, ModemDecorator, that supplies all the delegation code. Then the actual decorators can simply derive from ModemDecorator and override only those methods that they need to.



### Extension Object

Still another way to add functionality to a hierarchy without changing it is to **use the EXTENSION OBJECT pattern**. This pattern is more complex than the others but is also much more powerful and flexible.

Each object in the hierarchy maintains a list of special extension objects. Each object also provides a method that allows the extension object to be looked up by name. The extension object provides methods that manipulate the original hierarchy object.

**For example**, let's assume that we have a BOM system again. We need to develop the ability for each object in this hierarchy to create an XML representation of itself.

We could put toXML methods in the hierarchy, but this would violate CCP. It may be that we don't want BOM stuff and XML stuff in the same class. We could create XML by using a VISITOR, but that doesn't allow us to separate the XMLgenerating code for each type of BOM object. In a VISITOR, all the XML-generating code for each BOM class would be in the same VISITOR object. What if we want to separate the XML generation for each different BOM object into its own class?



Note that the extension objects are loaded into each BOM object by that object's constructor. This means that, to some extent, the BOM objects still depend on the XML and CSV classes. If **even this tenuous dependency needs to be broken, we could create a FACTORY[4] object that creates the BOM objects and loads their extensions**.

The fact that the extension objects can be loaded into the object creates a great deal of flexibility. Certain extension objects can be inserted or deleted from objects depending upon the state of the system. It would be easy to get carried away with this flexibility.

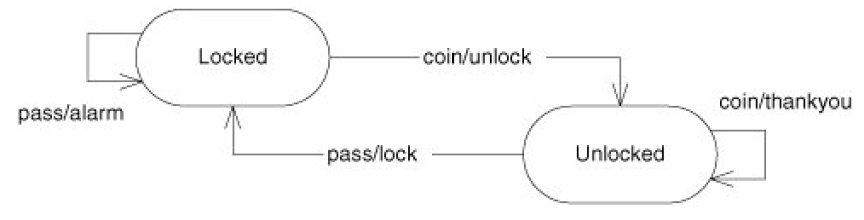
### Conclusion

The VISITOR family of patterns provides us with **a number of ways to modify the behavior of a hierarchy of classes without having to change them**. Thus, they **help us maintain the Open/Closed Principle**. They also provide mechanisms for segregating various kinds of functionality, keeping classes from getting cluttered with many different functions. As such, they **help us maintain the Common Closure Principle**. It should be clear that LSP and DIP are also applied to the structure of the VISITOR family.

The VISITOR patterns are seductive. It is easy to get carried away with them. Use them when they help, but maintain a healthy skepticism about their necessity. Often, something that can be solved with a VISITOR can also be solved by something simpler.[5]

## Chapter 36. State

Subway turnstile example.



### Switch/Case statements

There are many different strategies for implementing an FSM. The first, and most direct, is through nested switch/case statements.

The need to create test code that verifies each unit in isolation forces us to decouple the code in ways we might not otherwise think of. Thus, testability is a force that drives the design to a less coupled state.

For simple state machines, the nested switch/case implementation is both elegant and efficient. All the states and events are visible on one or two pages of code. However, for larger FSMs, the situation changes. In a state machine with dozens of states and events, the code devolves into page after page of case statements. There are no convenient locators to help you see where, in the state machine, you are reading. Maintaining long, nested switch/case statements can be a very difficult and error-prone job.

**Another cost of the nested switch/case is that there is no good separation between the logic of the FSM and the code that implements the actions.**

### Transitions table

A common technique for implementing FSMs is to create a data table that describes the transitions. This table is interpreted by an engine that handles the events.

The engine looks up the transition that matches the event, invokes the appropriate action, and changes the state.

One powerful benefit is that the code that builds the transition table reads like a canonical state transition table. The four AddTransition lines can be very easily understood. The logic of the state machine is all in one place and is not contaminated with the implementation of the actions.

Maintaining an FSM like this is very easy compared to the nested switch/case implementation. To add a new transition, one simply adds a new AddTransition line to the Turnstile constructor.

Another benefit of this approach is that the table can easily be changed at runtime. This allows for dynamic alteration of the logic of the state machine.

Still another benefit is that multiple tables can be created, each representing a different FSM logic. These tables can be selected at runtime, based on starting conditions.

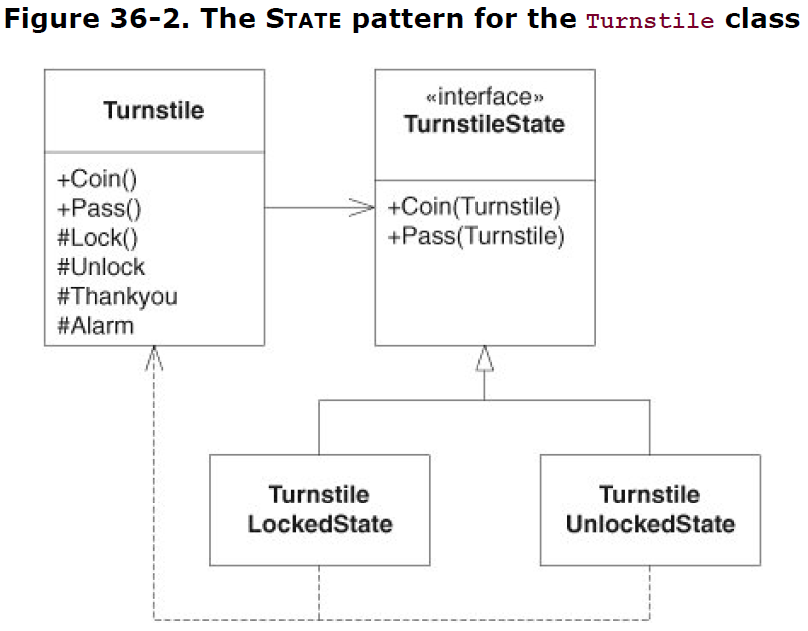
**The cost of the approach is primarily speed. It takes time to search through the transition table. For large state machines, that time may become significant.**

### The State Pattern

Another technique for implementing FSMs is the STATE pattern.[1] This pattern combines much of the efficiency of the nested switch/case statement with much of the flexibility of interpreting a transition table.

Figure 36-2 shows the structure of the solution:

* The Turnstile class has public methods for the events
* and protected methods for the actions.
* It holds a reference to an interface called TurnstileState.
* The two derivatives of TurnstileState represent the two states of the FSM.



When one of the two event methods of Turnstile is invoked, it delegates that event to the

TurnstileState object. The methods of TurnstileLockedState implement the appropriate actions for the Locked state. The methods of TurnstileUnlocked-State implement the appropriate actions for the Unlocked state. To change the state of the FSM, the reference in the Turnstile object is assigned to an instance of one of these derivatives.

For example, the Coin method of LockedTurnstileState tells the Turnstile object to change state to the unlocked state and then invokes the Unlock action function of Turnstile.

The Turnstile class is shown in Listing 36-8. Note the static variables that hold the derivatives of TurnstileState. These classes have no variables and therefore never need to have more than one instance. Holding the instances of the TurnstileState derivatives in variables obviates the need to create a new instance every time the state changes. Making those variables static obviates the need to create new instances of the derivatives in the event that we need more than one instance of Turnstile.

### State versus Strategy

Figure 36-2 is strongly reminiscent of the STRATEGY pattern.[2]

Both have a context class, and both delegate to a polymorphic base class that has several derivatives.

**The difference** (see Figure 36-3) is that in STATE, the derivatives hold a reference back to the context class. The primary function of the derivatives is to select and invoke methods of the context class through that reference.

**In the STRATEGY pattern, no such constraint or intent exists.** The STRATEGY derivatives are not required to hold a reference to the context and are not required to call methods on the context. **Thus, all instances of the STATE pattern are also instances of the STRATEGY pattern, but not all instances of STRATEGY are STATE.**

The STATE pattern provides a **strong separation between the actions and the logic of the state machine**. The actions are implemented in the Context class, and the logic is distributed through the derivatives of the State class. This makes it very simple to change one without affecting the other.

For example, it would be very easy to reuse the actions of the Context class with a different state logic by simply using a different set of derivatives of the State class. Alternatively, we could create Context subclasses that modify or replace the actions without affecting the logic of the State derivatives.

### SMC-generated state machine

Clearly, we've managed to maximize the benefits of the various approaches. The description of the FSM is contained in once place and is very easy to maintain. The logic of the FSM is strongly isolated from the implementation of the actions, enabling each to be changed without impact on the other.

The solution is efficient and elegant and requires a minimum of coding. The cost is in the use of SMC. You have to procure, and learn how to use, another tool. In this case, however, the tool is remarkably simple to install and use, and it's free.

Login.sm

Initial init

{

init

{

Start (event) logginIn (state) displayLoginScreen (action)

}

logginIn

{

enter checkingPassword checkPassword

cancel init clearScreen

}

checkingPassword

{

passwordGood loggedIn startUserProcess

passwordBad notifyingPasswordBad displayBadPasswordScreen

thirdBadPassword screenLocked displayLockScreen

}

notifyingPasswordBad

{

OK checkingPassword displayLoginScreen

cancel init clearScreen

}

screenLocked

{

enter checkingAdminPassword checkAdminPassword

}

checkingAdminPassword

{

passwordGood init clearScreen

passwordBad screenLocked displayLockScreen

}

}

States: init, loggedIn, checkingPassword, notifyingPasswordBad, screenLocked, checkingAdminPassword

Events: Start, enter, cancel, passwordGood, passwordBad, thirdBadPassword, OK

Actions: displayLoginScreen, checkPassword, clearScreen, startUserProcess, displayBadPasswordScreen, displayLockScreen, checkAdminPassword