AN OBJECT ORIENTED FRAMEWORK FOR ACCOUNTING SYSTEMS

BY

PAUL DUSTIN KEEFER

B.S., SUNY College at Buffalo, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

Table of Contents

Chapter 1. Introduction

Many people believe that object oriented technology can greatly improve reuse [Meye88]. Framework design is an area with one of the greatest potentials for reuse because with frameworks, it is not functions or even applications that are being reused, but application domain definitions. System developers have been experiencing some success with horizontal frameworks, primarily for user interfaces, but very little has been done yet with vertical frameworks, which are developed for a particular application subject's domain. The domain of accounting systems is particularly well-suited for framework design. This thesis describes the Accounts framework developed using Smalltalk-80 [Gold83] at the University of Illinois at Urbana-Champaign.

1.1. Reusability

Some observers [Cox91] have pointed out one important difference between software engineering and other engineering fields. The products produced by most engineering disciplines (such as electrical engineering) are built by plugging together existing parts. Even when new parts must be custom-built for some products, the type of part is the same. For example, all refrigerators have one or two doors in front even though the actual parts used for the door may be specific to a particular model.

Software, however, is rarely this uniform. Most developers create new systems from scratch, sometimes without the benefit of seeing any other applications or designs in the same domain. This is one of the reasons software takes so long to develop. Often, a system has already been built to do something similar to what the developers want it to do, but they are unaware of its existence. And there are no books that one can study telling a designer how to build, for example, a payroll system.

To some degree, mass-produced software has solved some of these problems. Spreadsheets, word processors, and databases provide an environment in which one can develop an application, often without writing any traditional program code. In some situations, however, these products are not powerful enough and/or not specific enough.

The general nature of a spreadsheet, word processor, or database means that someone who wants to use these products to write a payroll system for their company may find they need to write a program to handle a situation specific to payroll, such as processing logic that distinguishes between salaried, hourly, and temporary employees.

Also, since this general purpose software knows nothing about payroll systems, two companies writing their own separate payroll systems will end up with major portions of their corresponding systems that are functionally equivalent to each other, since most payroll systems have a great deal in common with each other. In fact, most accounting systems (of which payroll systems are a subset) have a great deal in common with each other.

1.2. Frameworks

A framework is an abstract design that describes the interaction between interacting objects in a particular application domain [John88, Deut89, John91]. It incorporates knowledge that is common to applications in the domain. The low level details are left to the framework user, since these are the things which vary among systems in the application domain.

Frameworks can be thought of as reusable designs. In one sense, they are the opposite of traditional reuse. When most people think of reusable code, they think of subroutines that someone else wrote that they can call from a program they are writing - such as a sort routine or a C library function. With frameworks, the reuse occurs at the top level of design. The overall system design is reused and the application developer writes the low level details and plugs them into the framework. Note that the application developer need not be a programmer. In fact, the ideal framework does not need to have a traditional programmer involved in application development.

Good framework design requires good scope definition. If the scope is too broad, the framework will require a great deal of program code to create a specific application or it will require an enormous amount of code to build the framework to a level of complexity where it can handle all the cases without resorting to writing program code. If the scope is too narrow, the framework will not be applicable to enough systems to warrant interest. However, *prototype* frameworks can have a narrow scope, which can then be broadened when developing the actual framework.

My own interest in frameworks began while working for my previous employer, Software Enterprises, Inc. in Clearwater, Florida. SEI developed traditional MIS/relational database/IBM mainframe systems. One of the secrets of our success was the high priority we placed on writing reusable code. While working there, I developed a framework for program design that was used in almost all of our online programs and was transferable across all the systems we developed. Through that experience, I became interested in making generalizations about software systems and implementing those generalizations programmatically.

1.3. Accounting Systems

Some people claim that object oriented programming is good for modeling the "real world." A better statement would be that object oriented programming is good for modeling the worlds in the human mind. Throughout history, people have developed systems for representing the physical world (science and mathematics) and the social world (commerce, politics, history, and law). In each type of system, only information that is important to that system is represented. The rest is intentionally ignored. For example, an apple would have different properties of interest depending on whether the observer is an artist, a physicist, a chef, or a produce supplier. A computer system would contain different information and different behavior depending on whose view of the world it is representing.

Accounting systems represent a business person's (or business entity's) view of the world. In the narrowest sense of the term, they are used to represent the flow of money in an organization. Two things they all have in common are the existence of accounts and of transactions that post to those accounts. Some examples are general ledger, payroll, and sales. Using a broader definition, accounting systems can keep track of many different kinds of non-financial data - inventory supplies, student grades, census data, and some meteorological data. Although the users of these broader-sense accounting systems may not think of these systems in terms of accounts and transactions, these systems all provide a way to keep track of changes through time to categories of objects, and as such can be represented using accounts and transactions.

There are some other features that accounting systems are likely to have. First, an accounting system will have some means for accessing transactions according to the date of the transaction. For example, the user of an inventory system should be able to find out the quantity of a particular item purchased so far this month. In order to do that, there must be some way to access transactions by date.

Another feature that occurs in some accounting system is the ability to create groups of accounts. For example, there might be different accounts representing foreign sales and domestic sales. In order to find out the total sales amount, it is often convenient to have an account that simply consists of the foreign sales and domestic sales accounts. Typically, accounts that group other accounts together do not have low level details defined for them. Instead, they combine information from their component accounts. For example, transactions are not applied to group accounts. A detailed account must be specified. Then, when summary information is needed for the group account, the individual account values are calculated and usually added together.

Another feature that is common in accounting systems is the storage of the data in some sort of a database. This is necessary whenever concurrent access to the same data is required (i.e., almost always).

1.4. A Framework for Accounting Systems

This thesis describes the design and implementation of a framework for accounting systems that includes the common features of accounting systems mentioned in the previous section. The Accounts framework began as a class project in Ralph Johnson's seminar on Object Oriented Programming at the University of Illinois. Eleven people developed two different implementations of the Accounts framework. Since then, Ralph Johnson, Rebecca Evans, and I have extended and rewritten most of the framework. We demonstrated the framework at OOPSLA '93, and hope to publish a paper on the framework soon.

Chapter 2. The Accounts Framework

2.1. Accounts and Transactions

The two kinds of objects that are common to all accounting systems are accounts and transactions. So it follows that these are the two primary classes in the framework. An account keeps track of transactions posted to it, and it is able to calculate attribute values as of a certain date. “Transaction” is a term commonly used in accounting systems and is a record of a numerically significant event concerning data in a category of relevance to the system. For example, an invoice, which is a kind of transaction, contains header information, such as the name of the vendor and the date, and it also contains a list of merchandise containing the name, quantity, and price of inventory items which have been purchased.

These transactions should not be confused with database transactions, which define atomic units of work. A common method for entering accounting transactions is to start up a database transaction, enter a large number of accounting transactions, and commit the database transaction. However, there are some applications, such as an airline reservation system, where a single accounting transaction (ticket reservation) is also a database transaction. And there may be some situations where one accounting transaction could consist of multiple database transactions. So a database transaction could consist of database accesses involving many account transactions, one account transaction, or a portion of one account transaction (although the first is much more common).

2.2. Roles

One way in which the Accounts framework is similar to database systems is that database systems generally have multiple levels of access. The DBA is responsible for designing and creating the database schema. The application programmer writes programs that access the database. The end user views and modifies the data in the database. Each role can be filled by more than one person, and one person can fill more than one role.

In the same way, the Accounts framework provides different levels of access. The first thing to do when setting up an accounting system is to determine what kinds of accounts and transactions you are going to have. For example, an inventory system would have some accounts that represent inventory items and other accounts representing vendors. An investment brokerage would have accounts for the customers (i.e., portfolios) and accounts for the investments (e.g., stocks, bonds, mutual funds, etc.). For each of these kinds of accounts you then need to determine the kinds of transactions that get posted to the accounts and the information you want to store for each of these kinds of accounts. The person who performs these functions is called the Accounts programmer. The Accounts programmer does not normally write any traditional program code, but instead "programs" the accounting system using the Accounts framework.

The Accounts programmer performs several other tasks, including designing the transaction entry screens, defining the account and transaction attributes, and defining the field mappings between transactions and accounts (these tasks are described later in the paper).

Another role in the Accounts framework is the person who creates the accounts specifically needed for that use of the accounting system. For example, all inventory systems will have a kind of account called Inventory Account, but the specific inventory items that the system keeps track of will depend on the specific business (or kind of business). A music store will keep track of keyboards. An office supply store will keep track of copiers. The implementation section will show that these two roles are not as distinct as they seem.

The third role in the Accounts framework is the data entry role - the person who enters the transactions and corrects errors.

2.3. Accounts

There are actually several kinds of accounts used in the framework. Simple accounts are those that represent the lowest level accounts. Each transaction will be stored in one of these simple accounts. There are two kinds of group accounts. Group accounts with relatively homogeneous components are called composite accounts. "Relatively" homogeneous means that the components do not all have to be the exact same kind of account, but they should have a common set of attribute names (though they may have additional attribute names that are not common). Composite accounts are helpful when a group account's attribute values are calculated by simply adding up its components' attribute values. Group accounts with heterogeneous components are called aggregate accounts. With these three kinds of accounts, an accounting system can be built using a single account structure. Conceptually, this structure is an acyclic directed graph. However, in the current implementation, the structure is a forest, and in most implementations, it will just be a single tree.

Although there are many possible ways to set up an account structure, one aggregate account will typically be at the highest level in each account tree and will be the entry point into the accounting system. Composite accounts will typically be used to represent *journals*. In the old days of pencil and paper, a journal was a book that kept track of related accounts. The inventory journal kept track of inventory items, the accounts payable journal stored the accounts of people or businesses to whom the company owed money, etc. Now, a journal is the computerized version of that book. The leaves of the tree will be simple accounts.

Aggregate accounts can also exist at lower levels in the account structure when components need different information or when the transaction is sent to more than one component. An example of this occurs in the Accounts Payable journal of a business. The accountant wants to determine, for example, how much the business owes to its creditors. One way to design the account structure is to make the due date the important date in that account. However, it's also important to be able to ask, "How much does the company owe for the things it purchased this month?" In this case, it is the purchase date, not the due date, that is important. So there would be two different kinds of Accounts Payable Accounts - those that sort their transactions by purchase date, and those that sort by due date. Figure 1 shows an example of an account tree for a store. The letters in the upper left corner of each account indicate whether it's an aggregate, composite, or simple account.

Note that the Accounts framework could have been designed so that the Accounts Payable example would just use a single kind of account whose transactions have all the fields in it (including both purchase date and due date). This would simplify the account structure. However, this advantage is more than offset by the additional complication of storing multiple collections of data (or a single collection with multiple indices) in a single account.

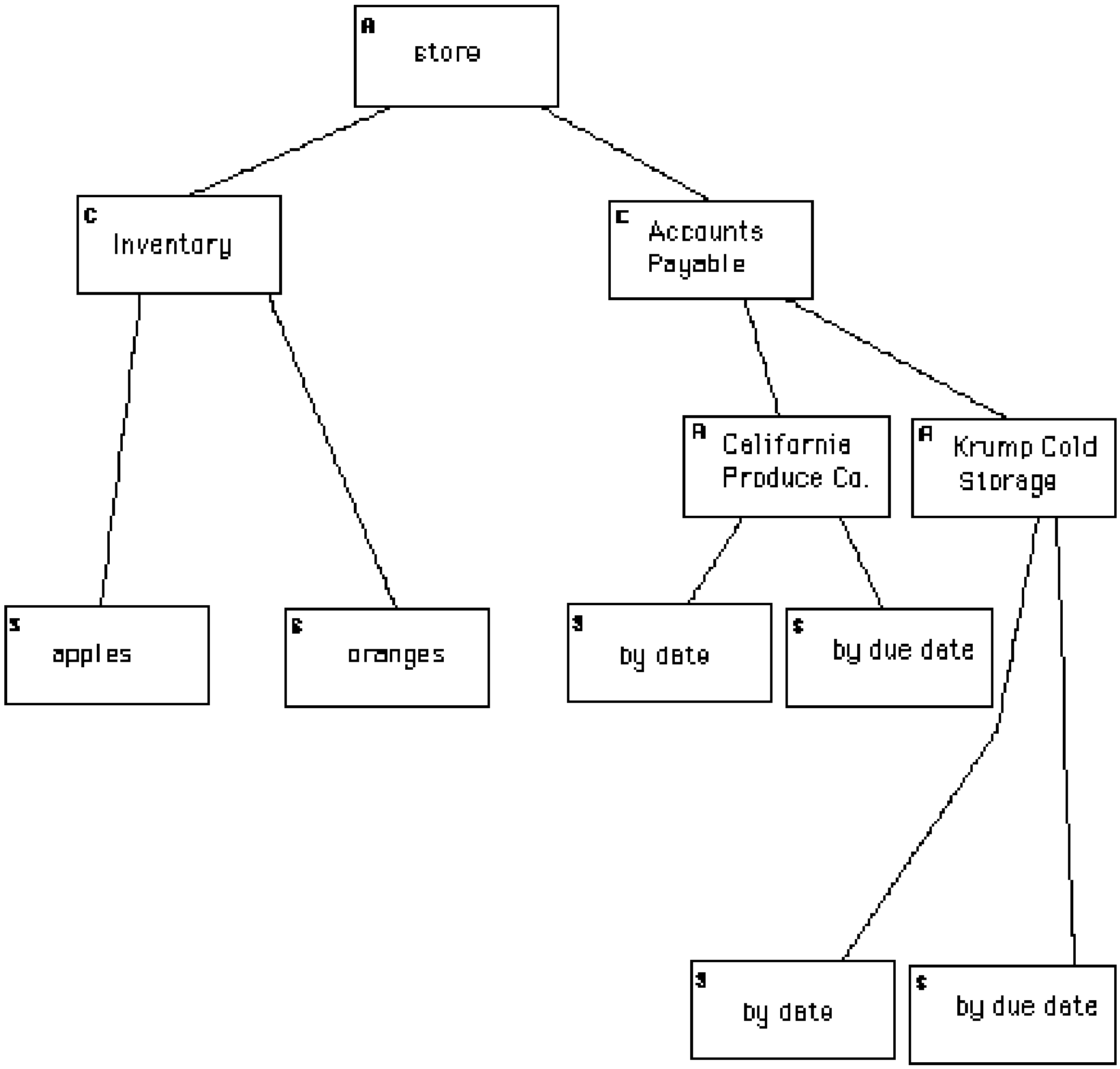

Figure 1

2.4. Transactions

When a transaction is posted, it starts at the root of an account tree (usually an aggregate account), which then passes it down through the graph until it reaches one or more leaves. Along the way, each account a transaction gets posted to will determine which of its component accounts to send it to, and will post the transaction to those components. If the account is a composite account, it will just send the transaction to one of its components. If it is an aggregate account, it will convert the transaction into one or more new transactions and will then post the new transactions to one or more of its components.

There are several reasons why transactions would be created for multiple components. The most common situation is when a transaction contains multiple sets or kinds of information. For example, if an invoice example has several inventory items, a separate transaction would be created for each inventory item and sent to the account representing that inventory item, and another transaction would be created for the vendor and would be sent to the account representing the vendor.

Another situation in which an account would create multiple transactions is in the Accounts Payable example in section 2.3. One transaction would be created for an account that uses the purchase date, and another would be created for an account that uses the due date.

There are two kinds of transactions in the framework. External transactions are those that are created and posted to an aggregate account by an entity outside of the aggregate accounts container hierarchy, such as a user or an automatic system process, or a sibling account. Internal transactions are transactions that an aggregate account creates from another transaction and sends to one of its components. The primary differences between the two are:

1) Part of defining the fields in an external transaction is specifying the type for each field. The type is needed for validating data that users enter later. Internal transactions are created by the system, so they should not contain type errors. Additional validation rules could be added to the framework.

2) An external transaction can have lists of entries (as in the invoice example). Therefore, the account to which the user posts an external transaction must break up the transaction into its individual components. Note that this is not a fundamental feature of the framework, so the framework could be changed so that internal transactions could also have lists of entries.

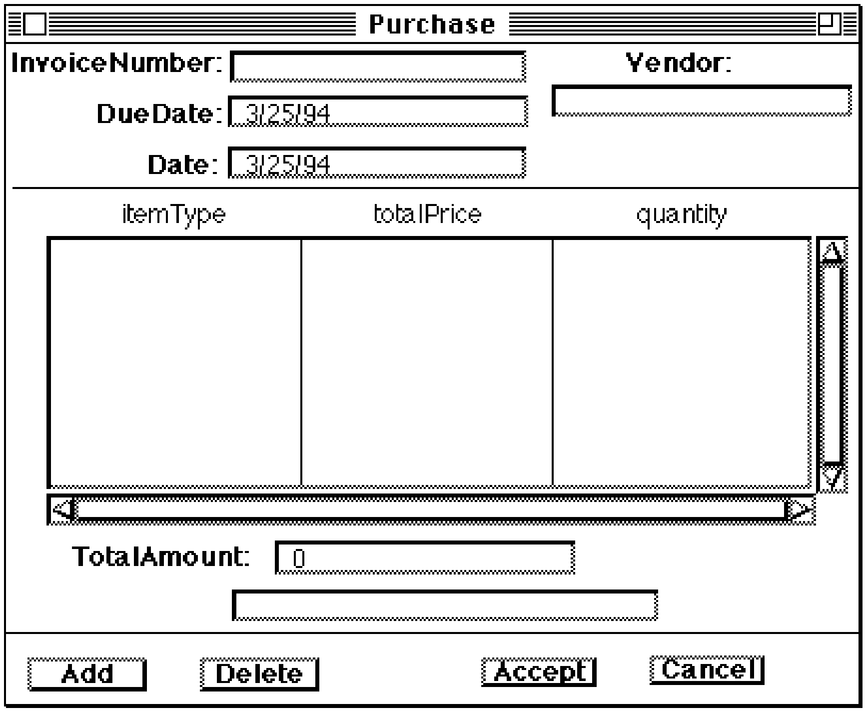

Figure 2

In the current implementation, an external transaction can have only one list. The fields in the list are called multi-value fields. Figure 2 shows an example of a purchase transaction in which the top part of the transaction shows the single-value fields, and the table shows the list. Each row in the table is a item in the list.

Creating new kinds of external transactions means defining the fields each kind of transaction will have, the fields within the lists multi-value fields, and the types of both kinds of fields. Figure 3 shows a tool that can define new types of external transactions.

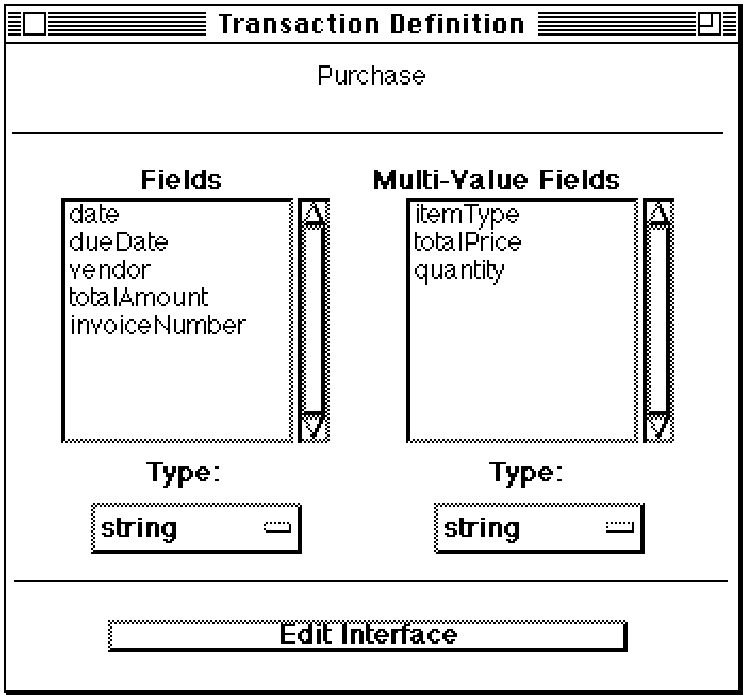

Figure 3

2.5. Posting Transactions

When the user posts an external transaction to an aggregate account (almost always the top level account) or when an account sends an internal transaction to one of its components, the account must determine how to handle it. This will depend on what kind of account it is.

Simple and composite accounts never alter a transaction. Aggregate accounts use a mapping from the external transaction fields to the internal transaction fields to create internal transactions and post them to component accounts. For each internal transaction field, the mapping indicates the source of its value. Currently, the source can either be an external transaction field or nil.

Figure 4 shows an example of this interface. On the left hand side is a single rectangle representing the Purchase transaction type from Figure 3. The rectangle is divided into smaller rectangles representing the fields on the external transaction. The fields with a box containing an ‘n’ are multi-value fields. On the right hand side are two rectangles representing the two accounts to which a Purchase transaction is posted - Accounts Payable and Inventory. Each of these is also divided into fields. The lines show the mapping from the external transaction fields to the internal transaction fields. When the account creates the internal transaction, it copies the values from the external transaction fields to the internal transaction fields based on the mapping.

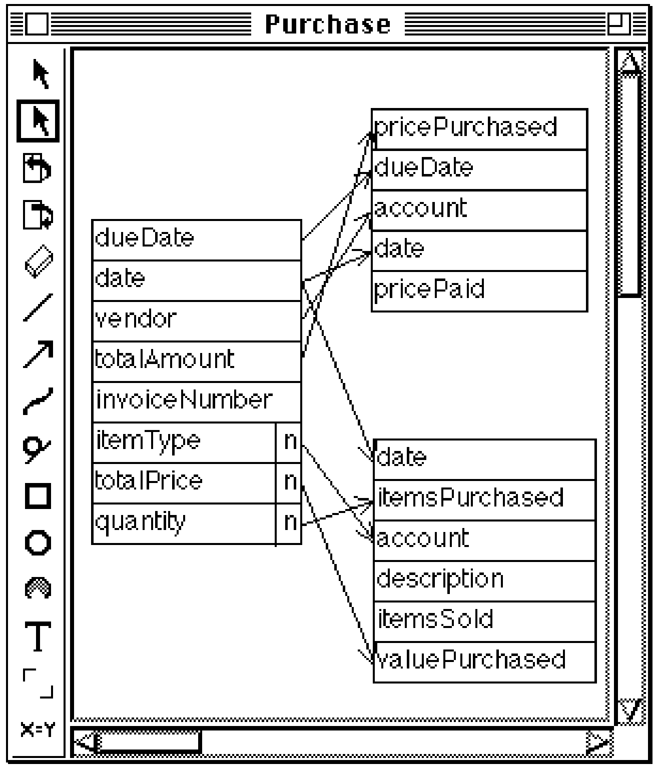

Figure 4

An external transaction field can map to more than one internal transaction. However, an internal transaction field should have either one source or no source. The reason an internal transaction field might not have a source is that the field might only be used with another external transaction type. For example, itemsSold only has meaning for Sale transactions, not for Purchase transactions. For any internal transaction types containing fields whose source is a multi-value field, the account will create multiple internal transactions, one for each list entry on the external transaction. For example, if a Purchase transaction contains purchases of apples, oranges, and kiwis, the Store account will send three internal transactions to the Inventory component account - one for each kind of inventory item on the transaction.

The list of internal transaction fields depends on the class of account to which it will be posted. For a simple account, the transaction field name list is the list of field names from its internal attributes. For a composite account, the transaction field name list is the union of the field names of its components. For an aggregate account, the transaction field name list is the list of fields defined for the external transaction type that gets created. Note that when an aggregate account occurs somewhere other than the top of the account hierarchy, it still requires an external transaction definition to determine the field names, even though it will probably never receive an external transaction.

When a composite account receives a transaction, it expects the name of the component to be present on the transaction in a field called ‘account’. When a simple account receives a transaction, it adds the transaction to its collection of transactions. At least that’s the conceptual view. The actual process is much more complex and is described in the implementation section.

2.6. Attributes

Once an accounting system is created and the users of the system are posting transactions, they need to have some way to find out information about the accounts based on transactions that they’ve posted. This kind of information is always relative to a particular range of time. For example, at the end of the month a produce store manager might want to know how many cantaloupes the store purchased this month. Then, the manager might want to compare this to the amount for the same month last year. Or a travel agency might want to find out how much money they have earned from a particular customer. Account attributes provide the means for performing these queries.

There are several kinds of attributes that the accounts programmer can define. In some cases, the kinds of attributes defined for an account depend on whether the account is an Aggregate, Composite, or Simple account. A Simple account can have an *internal* attribute, which retrieves data from a field on the transactions that have been posted to it. Since transactions are only stored in Simple accounts, internal attributes can only be defined for Simple accounts. Constant attributes, which any account can have, have a value that remains constant through time.

The other kinds of attributes are specified by combining other attributes, either of the account itself, or of one of its components, using arithmetic expressions. Of course, referring to an attribute of a component can’t be done in a Simple account, which doesn’t have any components. Figure 5 shows an example of an attribute.

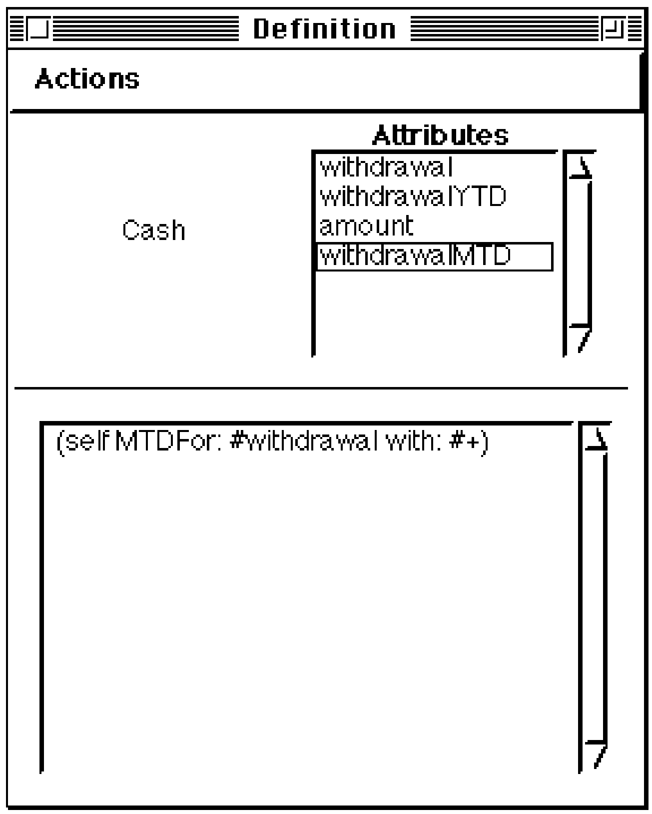
Definition
Actions
Attributes
Cash
withdrawal
withdrawalYTD
amount
withdrawalMTD
(self MTDFor: #withdrawal with: #+)

Figure 5

Chapter 3. The Accounts Framework Design

Figure 6 on the next page shows an entity-relationship diagram for the main classes in the Accounts framework.

3.1. Account Types vs. Accounts

As Chapter 2 points out, defining account types is different from creating accounts. An account type is something like an Inventory Account in an inventory system, and an account is something like “keyboards” for a music store. In fact, there are actually many levels to an account. The music store will not only need to keep track of keyboards overall, it will need to keep track of keyboards by manufacturer, and within manufacturer, by model. Each of these levels can be represented by accounts such as “keyboards”, “Roland”, and “D-50”. Each level serves two functions - as a description of the kind of accounts below it and as a collection of the accounts below it. The “keyboards” account is a particular kind of inventory item that both describes and contains all keyboard accounts, “Roland” describes and contains all keyboards made by Roland, and D-50 is a particular kind of keyboard made by Roland.

This observation about accounts leads to two interesting conclusions about the implementation. Initially, it may sound as though defining the kinds of accounts would occur in a different class than defining the accounts themselves. However, as noted in the previous paragraph, defining new kinds of accounts and collections often occur in the same situations. So it makes sense to use the same class (**Account**) to define the kinds of accounts as well as the collections of accounts.

The second thing to note is that although the kinds of accounts often parallel the account collections themselves, this is not always the case (with aggregate accounts, it is rarely the case). Therefore, the kinds of accounts and the account collections should be separate. The Accounts framework defines accounts using “template” accounts - accounts which ordinarily do not have transactions posted to them, but which act as an account type definition. When a new account is created, it is cloned from a template account. So in the above example, there would be a composite account named “InventoryJournal”, which is a group account containing all inventory-related accounts,

Figure 6

and a simple account named InventoryAccount, which is a template account containing the information about inventory-related accounts. When the “keyboards” account is created, its type is InventoryAccount, and it is a component of InventoryJournal. Initially, the system has three template accounts, “Simple”, “Composite”, and “Aggregate”. There are three type-based account trees (which are distinct from the account composition graph), where these three accounts are the roots of their respective trees.

So all accounts have a parent (the template account), a Smalltalk class (**SimpleAccount, CompositeAccount,** or **AggregateAccount**), and zero or more containers (in most accounting systems, only template accounts and the top-level non-template account will have zero containers). Figure 7 shows the same account structure as in Figure 1 with the template accounts added. The template accounts have an asterisk in the upper right hand corner. The template accounts, the store account, and all of the composite accounts have one of the three initial template accounts as its parent.

There are five advantages to using template accounts. One is that, although the roles of Accounts programmer and the creator of accounts for that particular installation can be filled by separate people, they don’t have to be. Since creating new kinds of accounts just involves creating new accounts and designating them as templates, it is easy to transfer knowledge between the two roles.

The second advantage of using template accounts is that templates allow the composition to be separate from type definition. One might argue that since the components of many group accounts all have the same type, the container could also define the account interface. However, that only applies to composite accounts and not aggregate accounts. And even for composite accounts the components should not be *required* to have the same type.

Consider an investment portfolio, which contains both stocks and bonds. The interface to stocks and bonds would be similar enough that a portfolio could be a composite account. The value of a portfolio would be determined by adding up the values of all its components, regardless of a kind. But there are obvious ways in which the interface would be different. For example, a bond would have an interest rate, but a stock would not.

The third advantage of using template accounts is that they allow the framework to have its own type system for accounts. Typically, the template will contain all or most of the behavior for accounts of that type. The user defines accounting system-specific account behavior with account attributes. All accounts inherit their attribute definitions from their template account, but they can also override them and have additional attributes. This would not be possible using Smalltalk methods, because Smalltalk (like most object oriented languages) does not have object-specific methods.

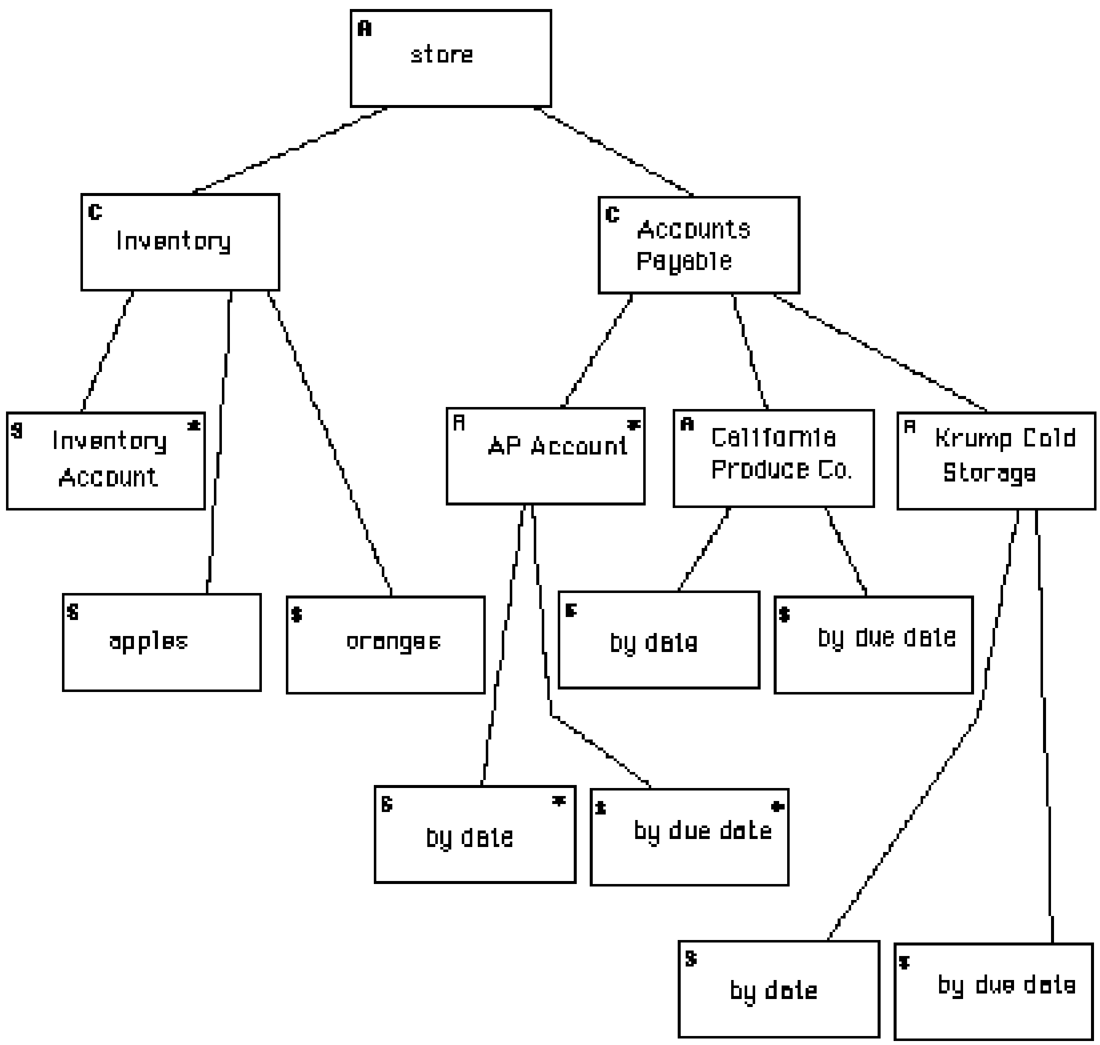

Figure 7

The fourth advantage of using template accounts is that they avoid generating code. When the topic of defining new types of things arises in the development of a Smalltalk system, the first thing that comes to mind is to generate new classes to represent the new types. In fact, the original implementation of the Accounts framework did this. Generating new classes is easy in Smalltalk, and is a somewhat glamorous approach. However, it can lead to several problems.

Generating new classes can cause conflicts with other classes with the same name. One of the interesting areas of framework development is combining several frameworks into a single system. Accounts incorporates two GUI frameworks. In the same way, it is conceivable that another framework could incorporate Accounts. Since new account types are defined by the Accounts programmer, the types can be anything - even the name of an existing Smalltalk class. Of course, the system could check for that, but it would reduce the flexibility in account type naming and would not solve the problem of conflicts with new classes that are created or filed in after the account types are created.

Also, generating new classes makes it impossible to implement the framework in most other object oriented environments without restarting the application after creating the new account types. This is because in most software systems, the development environment is separate from the execution environment. This distinction would encourage separation of the Accounts programmer role from the account creator role.

The third problem with generating new classes is that, although it's not required, it becomes easy to create methods to implement type behavior. Ultimately, the goal of any framework should make it possible to build applications without writing any program code. While it is understandably sometimes an unattainable goal, the development of a framework should proceed with that goal in mind. When new types are represented as objects and not as classes, it encourages the developer to look for solutions that do not require writing methods to implement behavior (unless, of course, the language has instance-based methods). The Accounts framework does not require any program code, nor does it generate any new code. It functions as a stand-alone system. (Additional program code is of course necessary to extend the functionality of the framework.)

The fifth, and possibly most interesting, advantage of using template accounts is that they provide an object oriented user interface. Many object oriented systems only look object oriented to the programmer. The user does not think in terms of objects, interface, inheritance, and reuse. In the Accounts framework, the advantages available to the object oriented programmer are also made available to the user. So the user can create templates that define the behavior for all accounts derived from that template, and can override the behavior in any account that is an exception.

3.2. Account Classes

There are three classes corresponding to the three kinds of accounts: **SimpleAccount, CompositeAccount,** and **AggregateAccount.** Also, there is an abstract class **Account**, a subclass of **Object,** that contains data and behavior common to all accounts. The class hierarchy is:

Account
    AggregateAccount
    CompositeAccount
    SimpleAccount

The **Account** class hierarchy is an example of the Composite pattern [Gamm94]. **AggregateAccount** and **CompositeAccount** are the Composite classes and **Account** is the component.

Although aggregate accounts and composite accounts are both group accounts, there is actually very little in common between the two (beside the features that are common to *all* accounts) other than their component management capability, and even most of the component management methods aren't the same. So it is not worthwhile creating a common abstract composite class. Before looking at these classes in more detail, it would be helpful to see some of the classes that support accounts.

3.3. Managing the "Accounts" Type Hierarchy

Each of the three kinds of accounts: 1) can be contained in another account; 2) can have attributes; 3) can be a template; 4) can be cloned from a template; 5) has a name. Each account is part of two orthogonal hierarchies, the composition hierarchy and the type hierarchy. The composition hierarchy usually has an aggregate account at the top and simple accounts as leaves, with composite accounts (and sometimes aggregate accounts) in the middle. There are three type hierarchies corresponding to the three subclasses of **Account** , and all the elements of a type hierarchy have the same class.

Note that the account type and composition structures are defined in Account and its subclasses, so additional classes for the structures (such as **AccountHierarchy** or **AccountTypeStructure**) aren't necessary. This is typical of the Composite pattern.

The primary methods for managing the type structure and creating an account are:

Method	Description
#type	Returns the name of the account's parent.
#rootType	Returns the name of the account's top-level parent (i.e., 'Simple', 'Composite', or 'Aggregate').
#new	Makes a new account that is a child of the receiver and that has the receiver as the parent. The instance variables are copied or reinitialized appropriately.
#name and #name: *aName*	Accessor methods for the name of the account.
#renameTo: *aName*	Renames the account to *aName* and informs its containers of the change. If the receiver is a template, it renames the account type as well.

The primary functions of the **Account** class methods are to deal with instance creation and to manage the type definitions (as opposed to the type *hierarchy*, which is managed in the instance methods). There are auxiliary class methods for storing the application, providing a new launcher menu, and for error handling, which chapter 4 discusses.

The primary method for instance creation is #makeNewAccount, which prompts the user for the type and then the name of the new account, then clones the template to create the new account.

The primary methods for managing the types are:

Method	**Description**
#addType: *anAccount*	Add *anAccount* to the list of types (i.e., template accounts).
#types	Returns the list of type names.
#typeNamed: *aName*	Returns the template account that is named *aName*.

The primary methods for maintaining the composition structure are at the aggregate and composite account level (see sections 3.5.2 and 3.5.3), although since all accounts can have containers, the addContainer: and removeContainer: methods are in **Account**.

3.4. Attributes

In addition to type and composition structures, all accounts can have attributes. The primary methods for managing the account's attributes are:

Method	**Description**
#attributeNamed: *aName*	Returns an attribute named *aName* (actually, an AttributeReference (see section 3.4.2.4).
#getAttribute: *aName*	Returns the definition of the attribute whose name is *aName*. The difference between this method and #attributeNamed: is that the previous method returns a representative of the attribute, while this method returns the actual attribute definition. See section 3.4.2.4 for further discussion of this distinction.

#addAttribute: *aFunctionOfTime* Named: *aName*	Adds *aFunctionOfTime* to the account, giving it *aName*. This may require updating the TransactionMapping (see section 3.5.3.1.2).
#removeAttribute: *aName*	Deletes the attribute named *aName* from the account. If the deleted attribute refers to an internal attribute, the deletion may cause changes that have to be propagated through the system, particularly the TransactionMapping.
#allAttributeNames	Returns a list of attribute names that are valid for the account, including those inherited from its parent.
#defaultAttrValueFor: *anAttributeName*	When the Accounts programmer creates a new attribute, the definition contains a default string that the programmer can then edit. (Note that the default is different for each subclass.)

3.4.1. Account Attribute

An account attribute is an object used to query some value defined on an account for a specified time range. There is not a separate **AccountAttribute** class. Instead, an account contains a dictionary of attributes where the key is the attribute name and the value is a function of time. Each association in the dictionary is an account attribute. Currently, account attributes cannot be defined for composite accounts. A composite account attribute value is calculated by combining the attribute values of its components.

3.4.1.1. Defining Account Attributes

The account definition view looks the same for each kind of account, and shows the attributes defined for the account. Figure 5 shows an example. The top half of the window contains the name of the account, along with a list of attribute names. The user can add, delete, and rename attributes. The bottom half of the window contains the definition that appears when the user selects an attribute. The user modifies the definition by editing it and selecting the "accept" pop-up menu option. Creating an account attribute just means giving it a name and an expression.

The expression should evaluate to a function of time. The particular subclass of **FunctionOfTime** that it will return depends on the expression.

3.4.2. Function of Time Classes

There are several classes used for defining attributes on an account: **ConstantFOT, BinaryFunctionOfFOT, InternalAttribute, AttributeReference,** and **ComponentAttributeReference.** All of these are subclasses of the abstract class **FunctionOfTime,** which is a subclass of **Object.**

FunctionOfTime is an abstract class that defines how to calculate for an account a value that changes through time. Since each subclass of **FunctionOfTime** contains different information, a function of time contains interface and behavior only. This is helpful for those who wish to extend the framework by adding new kinds of functions of time, and for those who wish to write additional code to access functions of time. To write new functions of time, the following **FunctionOfTime** methods need to be redefined in the subclasses. All of them are defined as self subclassResponsibility in **FunctionOfTime.**

Method	**Description**
#atTime: *aTime* forAccount: *anAccount*	Returns the function of time's value at *aTime* for *anAccount.*
#fields	Returns the set of internal attributes contained in this function of time.
#printOn: *aStream*	Prints the function of time on *aStream*. This method is used for displaying the function of time in the account attribute definition window in Figure 5.

Some **FunctionOfTime** methods are used to create new functions of time. These are described in the subclasses.

3.4.2.1. Account Attribute Definition Language

The Accounts programmer defines an account attribute by giving it a name and an expression that gets converted into a function of time (see Figure 5). The language used to write these expressions uses Smalltalk-80 syntax. This is because we were able to modify the Smalltalk-80 compiler to parse these expressions.

The **WorkspaceCompiler** class is a subclass of **Compiler** with a workspace dictionary that provides a list of valid symbols and what they represent. In the case of an aggregate account, the workspace dictionary is the account's dictionary of components. When the name of one of the aggregate account's components is used in an expression, the workspace compiler looks the name up in its dictionary and it returns the corresponding aggregate component. For a simple account, the workspace dictionary is nil.

When the account sends itself the #convertStringToAttribute: message (where the string is an expression that defines a function of time), it tells the workspace compiler to evaluate the expression with self as the receiver . This means the workspace compiler will create a temporary message whose body is the expression, and it will evaluate the expression by sending that message to the account. This attribute definition language, which creates functions of time to store the interpreted strings, is an example of the Interpreter pattern [Gamm94].

3.4.2.2. Constant Functions of Time

The **ConstantFOT** class defines a value that is the same at all times. The Accounts programmer defines a constant FOT simply by typing an arithmetic value in the account attribute definition window. The **ArithmeticValue** class has a new method called #asFunctionOfTime that creates a new constant FOT. Other kinds of constants in addition to **ArithmeticValue** can be defined as constant FOTs by adding the #asFunctionOfTime method to their classes.

A constant FOT calculates its value by returning the constant. It has no internal attributes to return, and its printing method just prints the constant.

3.4.2.3. Binary Functions of Functions of Time

The **BinaryFunctionOfFOT** class defines a function of time in terms of two operands that are also functions of time. A binary function of FOT's value changes through time as its operands' values change.

The Accounts programmer defines a binary function of FOT by entering an expression of the form *operand1 operator operand2* where the two operands are functions of time and the operators currently supported are +, -, *, and /. These four operators are defined as methods in **FunctionOfTime.** They create a new binary function of FOT using self as the first operand and the input argument as the second operand. Supporting other binary operators would involve adding the operator to **FunctionOfTime** and making sure **ArithmeticValue** understands it.

Since the Smalltalk compiler parses the expression, the precedence rules for combining multiple Binary functions of FOT are the same as for Smalltalk arithmetic expressions - left to right, with parentheses to override that order. So in the expression FOT1 + FOT2 * FOT3, the addition would occur first, but in FOT1 + (FOT2 * FOT3), the multiplication would occur first.

A Binary function of FOT calculates its value by combining the two operators using the operand and prints itself using a left parenthesis, followed by a the first operand, followed by the operator, followed by the second operand, followed by a right parenthesis. The parentheses preserve the order of evaluation. Its internal attributes are the union of the internal attributes of the two operands.

3.4.2.4. Attribute Reference

Almost all programming languages support procedural abstraction, which is the ability to define new operations in terms of existing ones, and then to use the new operation like the older ones. Procedural abstraction makes programs smaller by eliminating duplication. It makes programs more flexible by letting us change the definition of an operation without having to change all the places it is used. And it makes programs easier to understand by breaking them into pieces that can be understood separately.

In the same way, functions of time can be defined in terms of other functions of time. The attribute definition self credits - self debits produces a function of time that refers to the credits and debits functions of time. A reference to a function of time cannot be the function of time itself, because the reference must print the name used to get the attribute (e.g., self credits) and not the definition of the function of time. A user can refer to another function of time in an attribute definition by typing self *attributeName.* This will cause the attribute definition to contain a reference to the function of time corresponding to *attributeName.* However, when the outer function of time prints itself, it cannot reproduce the string self *attributeName,* because it doesn't know the name of the attribute where the inner function of time is stored. And the inner function of time can't print the string either, because it doesn't know the name it's stored under. And it shouldn't know this, because it is the attribute's responsibility (and since there are no special attribute definition objects, it therefore becomes the responsibility of the account that owns it). An attribute reference gives a function of time an additional interface (beyond that provided by all functions of time) so that another function of time that includes it can print itself without printing the definition of the inner function of time. In this sense, it is an example of the Decorator pattern [Gamm94] (also known as the Wrapper pattern), which is used to add additional properties without subclassing.

There is another reason for having an attribute reference. When the user changes a function of time, instead of updating the current function of time object, it deletes it and recreates a new one. This is because the new function of time might be in a different class than the old one. If an outer function of time referred directly to an inner function of time, then every time the attribute definition changed for the inner function of time, the outer function of time would have to be changed as well. Although a dependent mechanism could handle this, it's more time-efficient to have a substitute object that represents the attribute rather than the function of time. This way, even if the inner function of time changes, the outer function of time will not have to. The attribute reference acts as this substitute object, and in doing so, it exhibits properties of the Proxy pattern [Gamm94].

The Accounts programmer defines an attribute reference using an expression of the form self *attributeName.* Since the expression is being sent to the account (see section 3.4.2.1), the #doesNotUnderstand: message is redefined in **Account** to create a new attribute reference. This means that the Accounts programmer should not define an attribute that is the same name as a method defined for the account, or an error message will appear like the one shown in Figure 8.

Figure 8

An attribute reference calculates its value by asking its account for the value of the attribute to which it refers, and prints itself using the word self followed by the name of the attribute to which it refers. Its internal attributes are those of the attribute to which it refers.

3.4.2.5. Component Attribute Reference

A component attribute reference is similar to an attribute reference except that a component attribute reference refers to the attribute of a *component* of the account that contains the component attribute reference. It is used in an aggregate account to define attributes based on the aggregate account's components.

The Accounts programmer defines a component attribute reference using an expression of the form *accountName attributeName* where *accountName* is the name of a component of the aggregate account, and *attributeName* is the name of one of the attributes of the component. This is where the aggregate component objects are used. The *attributeName* is a message sent to the aggregate component, which means that #doesNotUnderstand: is redefined in **AggregateComponent** to create a new component attribute reference. As with an attribute reference, the user should not define an attribute for an account that is the same name as a method defined for the aggregate component.

A component attribute reference calculates its value by asking its account's component for the value of its attribute, and prints itself using the component's name and the attribute's name. Its internal attributes are those of the account's component's attribute.

3.4.2.6. Internal Attributes

A simple account stores transactions and can access them when an internal attribute asks it to do so. The **InternalAttribute** class defines how to calculate values from transactions using the transaction field name, a time range, and an operator for combining transaction values. Note that multiple internal attributes can refer to the same transaction field.

There are two ways to define an internal attribute. The more common method uses an expression of the form self *timeRange*For: #*fieldName* with: #*op* where *timeRange* is either total, MTD, or YTD, *fieldName* is the name of the transaction field that is used to determine this attribute value, and *op* is either +, *, min:, or max:. This means that an internal attribute is calculated by summing, multiplying, finding the minimum value, or finding the maximum value across all of the transactions for the specified time range.

The time range specifies the beginning of the time range for the internal attribute. The end of the time range is specified by the query, as shown in Figure 17. The time ranges currently supported are total (compute the value over all transactions up to the requested date), mtd (compute from the beginning of the month to the requested date), and ytd (compute from the beginning of the year to the requested date). Although any time range could be calculated, the framework only supports O(log n) access for these predefined ranges (see section 4.1). Other ranges would take O(n) time.

useOldUnlessNewNotNil: is a special operator used for attributes that can change over time, but whose values are not cumulative, such as an interest rate or an address. The result of calculating an attribute as of a particular date using useOldUnlessNewNotNil: will be the last non-nil value that was posted to the account as of that date.

The easiest way to use the operation useOldUnlessNewNotNil: is to enter an expression of the form self currentValue: #*fieldName* where *fieldName* is the name of the field in the transaction which is used to determine this attribute value. This syntax is provided so the user does not have to type in self totalFor: #*fieldName* with: #useOldUnlessNewNotNil:.

#+, #*, #min:, and #max:. are existing Smalltalk methods defined in **ArithmeticValue** or its ancestors. But there is no existing Smalltalk method that looks at two values and returns the first value if the second is nil, otherwise it returns the second value. So #useOldUnlessNewNotNil: is a new function defined in **Object.** One of the nice features of Smalltalk is that it allows you to change the pre-defined classes. However, this should not be done recklessly. The name "useOldUnlessNewNotNil:" is descriptive and is complex enough that one is not likely to mistakenly use it for something else.

Once again, since the expression is sent as a message to the account, the account must be capable of handling the expression. #totalFor:with:, #MTDFor:with:, #YTDFor:with:, and #currentValue: are all methods defined in **SimpleAccount** (since only simple accounts can have internal attributes) that create an internal attribute.

An internal attribute calculates its value by asking the account to look at its transactions and to calculate the value as of the input time using the internal attribute. Its method for printing itself depends on the operator. If the operator is useOldUnlessNewNotNil:, it prints the phrase self currentValue: followed by the transaction field name on the input stream. Otherwise, it prints self, followed by the time range code, followed by For:, followed by the transaction field name, followed by with:, followed by the operator. Its internal attribute is itself.

Additional **InternalAttribute** methods are:

Method	**Description**
#identity	Returns an identity value for the internal attribute's operator. This is used to initialize calculated values. The identity for #+ is 0, for #* is 1, for #min: is infinity, for #max: is negative infinity, and for #useOldUnlessNewNotNil: is nil.

#inverseOperator	For time ranges other than total, the internal attribute's value is calculated by determining the value as of the start date, the value as of the end date, and applying the inverse operator. This only works for #+ with an inverse operator of #-, and for #* with an inverse operator of #/. For the other operators, it simply returns the value as of the end date.
#range: *aDate*	For time ranges other than total, returns the start date based on the internal attribute's time range and *aDate*. E.g., if the time range is mtd and the input date is May 19, 1994, the method returns May 1, 1994. If the time range was ytd, the method would return January 1, 1994.

3.4.2.7. Function of Time Examples

Table 1 shows some examples of account attributes and how they are defined. itemsPurchased, purchasedMTD, and itemDescription are all examples of internal attributes. itemsOnHand is a binary function of FOT that subtracts one attribute reference from another. totalAmountOwed, which would be defined in an aggregate account, is a component attribute reference, where APJ is the name of a component of the aggregate account. interestPercent is a binary function of FOT that multiplies an attribute reference by a constant FOT.

Attribute Name	Function of Time Expression
itemsPurchased	self totalFor: #purchased with: #+
purchaseMTD	self mtdFor: #purchased with: #+
itemDescription	self currentValue: #description
itemsOnHand	self itemsPurchased - self itemsSold
totalAmountOwed	APJ amountOwed
interestPercent	self interestRate * 100

Table 1

3.5. Account Subclasses

3.5.1. Simple Accounts

Simple accounts are accounts that have no components and therefore store transactions. They can directly query the values of the transactions. There are some attributes (specifically, internal attributes) that can only be defined for simple accounts. The methods for creating internal attributes are discussed in section 3.4.2.6.

The primary methods for accessing simple account attributes are:

Method	**Description**
#addAttribute: *anAttributeName* compute: *anOperator* calculate: *aCode*	Returns a new InternalAttribute object that has the input name, input operator, and input calculation code (see section 3.4.2.6). If the internal attribute is a new one for its container (i.e., if it would cause a change to the mapping diagram (Figure 4)), a warning message like the one in Figure 9 appears.
#internalAttributeList	Returns a list of all the InternalAttributes contained in the account's attributes (including inherited attributes).

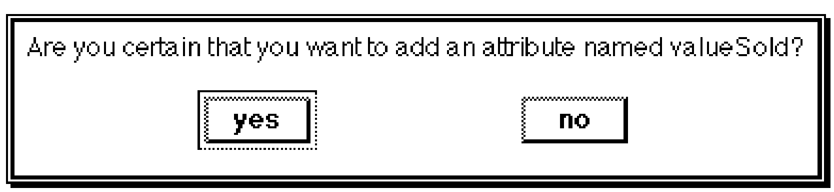

Figure 9

The primary method for accessing transactions is #transactionList, which returns a list of the transactions posted to the account in transaction date order. Methods for posting transactions are discussed in section 4.5.

3.5.2. Composite Accounts

A composite account is an account that contains relatively homogeneous accounts. This implies several things. For the most part, the transactions that get posted to them will have the same interface (i.e., attribute names). This means that first, the list of queryable attributes of a composite account is the union of the attributes defined in its components. And second, querying an attribute for a composite account returns the sum of the values of that attribute over its components.

The primary methods for accessing composite account components are:

Method	**Description**
#addAccount: *anAccount* named: *aName*	Adds *anAccount* as a component of the receiver, using *aName* as a key. If the new component has any attributes that refer to transaction fields that the account doesn't already know about, it gives the user a warning message like the one in Figure 9. It also tells the new component to add the account to its list of containers.
#accountNamed: *aName*	Returns the component named *aName*. If it doesn't find the component, it produces an error.
#accountNames	Returns a list of the names of the receiver's components.
#removeAccount: *aName*	Tells the component named *aName* to remove the receiver from its container list, and then removes the component from the receiver.
#renameAccount: *oldName* to: *newName*	Renames the component from *oldName* to *newName*.

3.5.3. Aggregate Accounts

AggregateAccount is the most complex account class. Aggregate accounts can contain many different types of accounts and transactions, and they can break a transaction into multiple transactions. Therefore, they need a more complex mechanism for determining where to send a transaction.

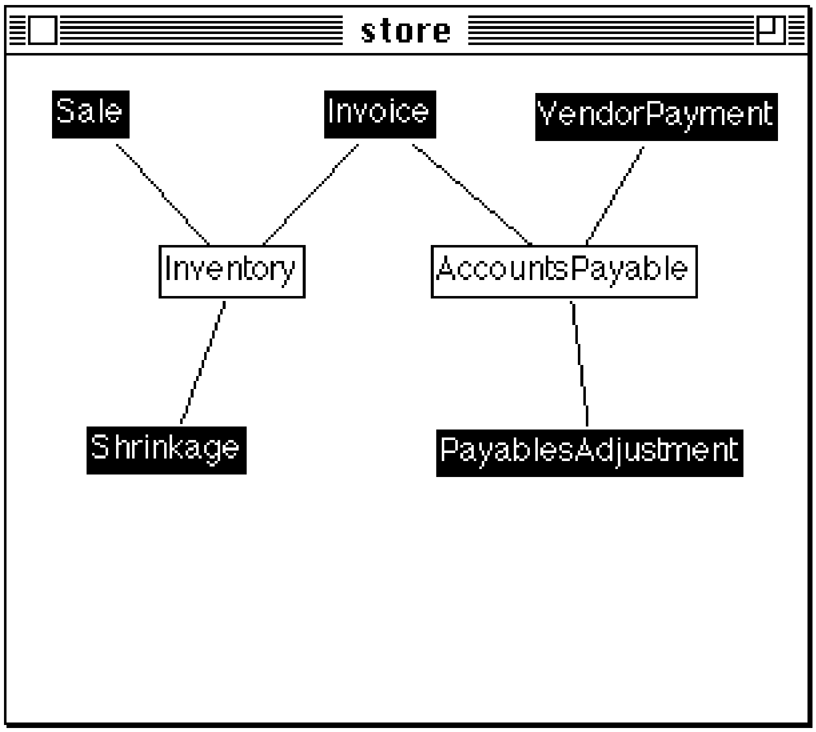

Figure 10

Accounts uses a visual programming language to describe how transactions can be broken into smaller transactions and posted to component accounts. The language itself is rather simple. Figure 10 shows an example of the components of a top-level aggregate account for a store.

The white rectangles represent component accounts and the black rectangles are external transaction types. The lines indicate which accounts each kind of transaction posts to. An Invoice transaction's expected effect is to increase both inventory and the accounts payable amount. So when the user creates a Invoice transaction, the Store account will convert the transaction into two different kinds of internal transactions, and send each kind of internal transaction to its appropriate account, either Inventory or

AccountsPayable. Each of the component accounts in this example is a composite account whose components are accounts of a similar type.

Each component of an aggregate account has an associated internal transaction type whose name is the concatenation of the component name and " Transaction". So the component named "Sale" would have an associated transaction type called "SaleTransaction". This feature does not introduce potential naming conflicts, because each aggregate account has its own name space for transaction types. Components of an aggregate account are already prevented from having the same name. An internal transaction type defines a set of field names that the aggregate account must create when it makes an internal transaction of that type.

The message #fieldNames, defined for all accounts, returns a list of the names of all the fields that should be on a transaction sent to the account. Note that the rules for building this list vary according to the class of the account.

The set of field names depends on the class of the component. The set of transaction field names of an aggregate account is the set of field names from the external transaction type for the transaction that will be created. In the current implementation, all aggregate accounts that are not top-level accounts will only have a single transaction type. This is because the system is operating under the assumption that each component of an aggregate will have a corresponding transaction type for the internal transaction that gets sent to it.

The set of transaction field names of a composite account is the union of all the transaction field names of the composite's components. This is because a composite account simply sends a transaction to its component without changing it, so the transaction the aggregate account creates must be readable by any of the composite account's components.

The set of transaction field names of a simple account is the union of all the transaction field names that the simple account's attributes use. This is because the simple account attributes must be able to read the transactions that the aggregate account creates. When a change occurs in the component's definition that would cause the list of fields to change, the list is updated at that point using the #fieldAdded: message.

So the aggregate account manages the external transaction types, the component accounts, the internal transaction types associated with them, and the relationships between all of these. It also stores the drawing in Figure 10 (an interface class called **AggregateAccountEditor** actually manages the drawing). And like all the other accounts, it manages attributes.

The interface for accessing aggregate account components is the same as for composite accounts, although a couple of the methods work a little differently. #addAccount: *anAccount* named: *aName* creates an aggregate component (see section 3.5.3.1.1) and generates the internal transaction type for *anAccount.*

The primary methods for accessing external transaction types (many of these methods just find the transaction mapping (see section 3.5.3.1.2) for the input external transaction type and send the information to the mapping) are:

Method	**Description**
#addExternalTranType: *aName*	Creates and stores a new instance of ExternalTransactionType (see section 3.6) named *aName*, and creates and stores a new instance of TransactionMapping (see section 3.5.3.1.2), which relates an external transaction type's fields to the fields they map to in the internal transaction type.
#externalTypeFor: *aName*	Returns the external transaction type named *aName.*
#associateInternal: *anInternalTypeName* withExternal: *anIExternalTypeName* and #disassociateInternal: *anInternalTypeName* fromExternal: *anIExternalTypeName*	Tells the transaction mapping for the input external transaction type named *anIExternalTypeName* to add or remove as a destination the input internal transaction type named *anInternalTypeName* .

#addField: *aField* toExternalType: *aType* and #removeField: *aField* fromExternalType: *aType*	Tells the transaction mapping for *aType* to add or remove *aField* .
#transactionFieldAdded: *aName* and #transactionFieldRemoved: *aName*	If the receiver is a component of another account, adding or removing an external transaction type field should cause the transaction mappings in its containers to be updated. These methods tell the containers about this change.

The primary method for accessing the components drawing is:

Method	Description
#drawing	Returns the drawing stored in the aggregate account. If no drawing is stored, it returns a copy of the account's parent's drawing. If there is no parent, it returns a new instance of AccountsDrawing.

3.5.3.1. Auxiliary Account Classes

3.5.3.1.1. Aggregate Components

An aggregate account attribute definition might refer to an attribute of one of its components. **AggregateComponent** provides an interface between the aggregate account and its component that the attribute definition can refer to. In other words, it acts as a wrapper for the component, and has a reference to the aggregate account and to its component. The primary purpose of aggregate components is to be used with component attribute references (see section 3.4.2.5).

3.5.3.1.2. Transaction Mapping

TransactionMapping describes how to convert an external transaction a set of internal transactions that can be posted to the aggregate account's components. Figure 4 shows the interface for the field mapping. It describes how to calculate each field of each internal transaction type. In the current implementation, each field of an internal transaction type is either nil or comes from a field in the external transaction type. Note that a given field on the external transaction type may have zero, one, or many destinations. The transaction mapping stores the drawing and manages the data that the drawing represents. The interface class **TransactionDrawingEditor** manages the drawing.

Figure 11 shows what the mapping in Figure 4 (for the Invoice transaction type) looks like in the Smalltalk inspector. There is a dictionary that has two entries for the two destinations (internal transaction types). The APJTransaction entry has a dictionary with an entry for each destination/source field pair. The "pricePaid" field's source is nil because an Invoice doesn't involve paying money. InvTransaction has a similar dictionary. The "itemsSold" field has a nil source because an Invoice shows items purchased, not items sold. The "itemsPurchased" field has a source field of "quantity" on the items list. That means that "itemsPurchased" comes from a multi-value field, and there will therefore be one InvTransaction created for each entry in the items list on the Invoice.

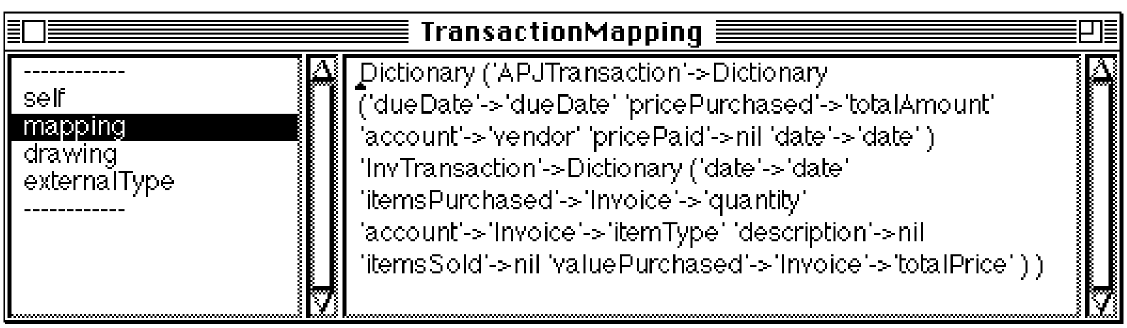

Figure 11

3.5.4. Accounts Interface

Every account has a contents view, which depends on the class of the account. The aggregate account contents view is the visual programming interface shown in Figure 10 and shows the component accounts, the external transaction types that post to the account, and the relationships between the two. The composite account contents view, shown in Figure 12, is simply a list of the components. Both of these views allow the user to add, delete, rename, and view components.

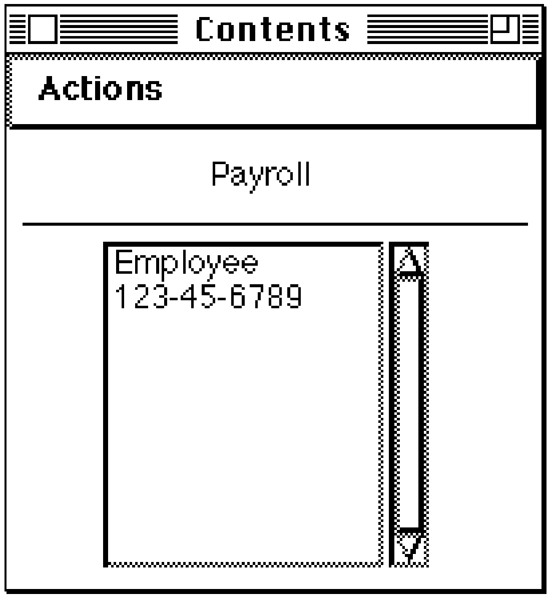

Figure 12

The simple account contents view is a list of the transactions stored in the account's transaction tree, and does not allow the user to enter or modify transactions. Figure 13 shows an example.

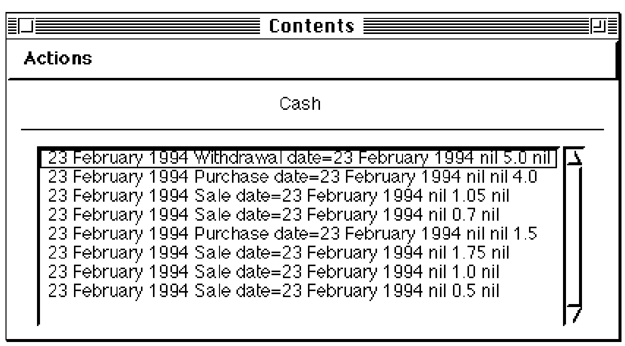

Figure 13

3.6. Transaction Classes

Since an external transaction can have singly-occurring and multiply-occurring fields, the **ExternalTransaction** class's primary responsibility is maintaining these two types of fields. In order to keep method names short, singly-occurring and multiply-occurring fields are called "attributes" and "lists" respectively.

ExternalTransaction has operations #at: and #at:put: for reading and writing attributes. It also has operations #addList: for creating a list, #list: *aName* values: *aDictionary* for adding an entry (row) in the list, #listEntry: attr: *aName* and #list: itemNumber:attr: for finding the value of a particular attribute in a particular row, and #listSize: for determining the number of rows in the list. It also has #typeName, which returns the name of the external transaction's type.

The **ExternalTransactionType** class defines the schema used to create an external transaction and to define the transaction entry window for that transaction type. It defines which attributes, lists, and list fields an external transaction of that type will have, and what the types of the attributes and list fields are. Since an external transaction type is defined for a specific aggregate account, the external transaction type knows its account.

Creating an external transaction type requires defining the fields an external transaction will have. Figure 3 shows an example of the interface for this definition. There are separate windows for single-value fields and multi-value fields. You can add, delete, rename, and set the type of fields using this interface. The valid types are date, number, and string.

After defining an external transaction type, you can design the transaction entry window for that transaction type. The VisualWorks interface code generates a default design based on the transaction definition with the top portion containing the single-value fields and the bottom portion containing the multi-value fields. Figure 14 shows the default transaction entry window generated after pressing the Edit Interface button in Figure 3.

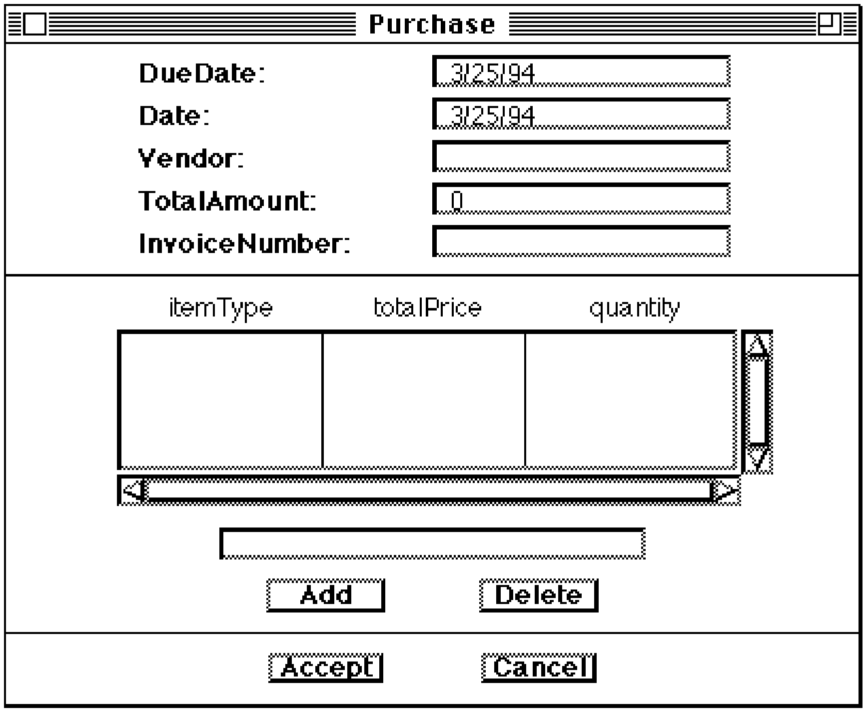

Figure 14

The interface editor allows a great deal of latitude in moving fields around, changing labels, etc. Figure 15 shows an example of a modified transaction layout.

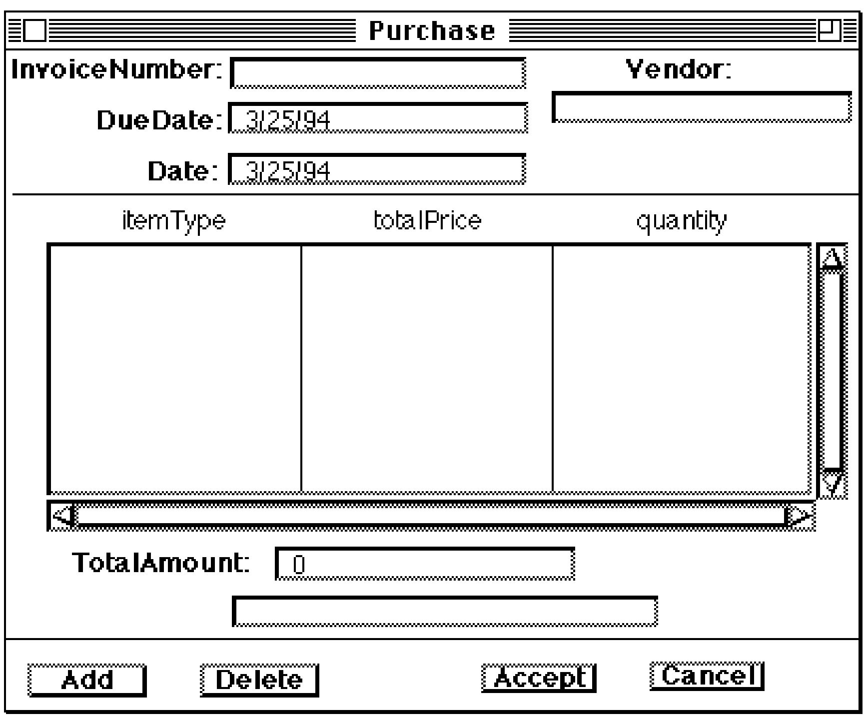

Figure 15

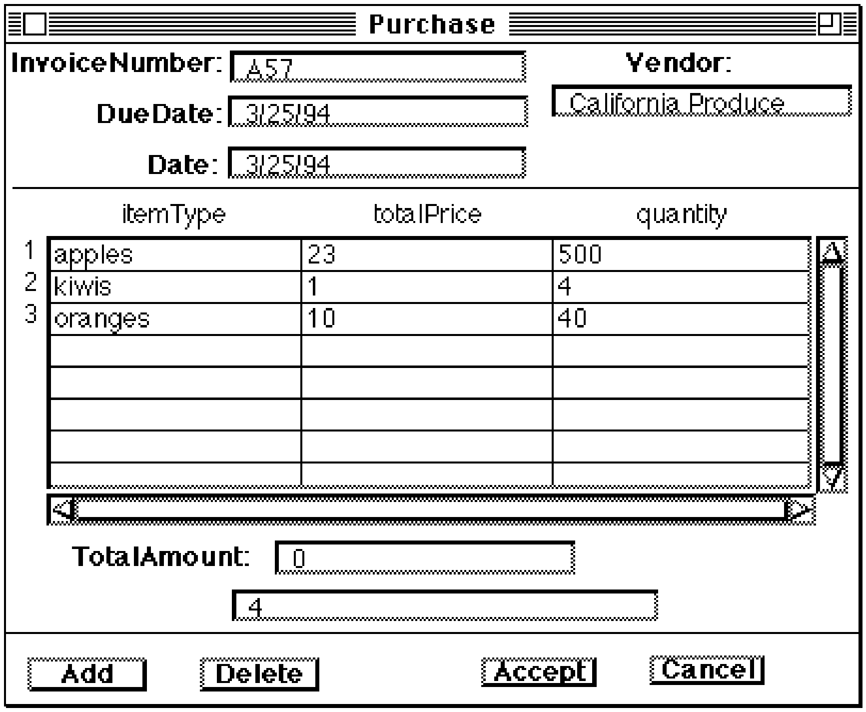

Figure 16

After designing the layout, you can start entering transactions using a window such as the one in Figure 16. Pressing the accept button causes the transaction to be posted to the aggregate account and, if there are no errors, it will be added to the appropriate simple accounts' transaction trees.

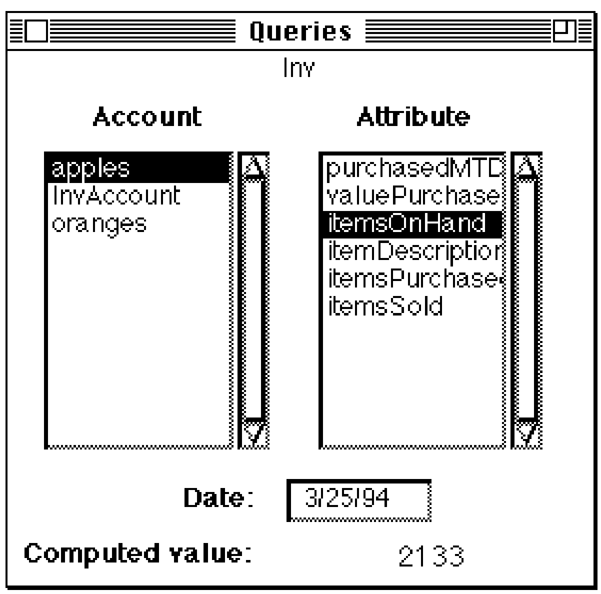

Figure 17

Finally, after entering transactions, you can query the accounts for information about those transactions using the query interface window in Figure 17. The Attribute pane shows a list of attributes defined for an account. Selecting a component account and an attribute will cause the value of that attribute for the specified date to appear at the bottom. The date defaults to the current date, but can be changed to any valid date.

An external transaction type has an #attributeType: *aName* method for returning the type of an attribute, #addAttribute:withType: for adding an attribute, and #removeAttribute: *aName* for removing it. #attributeNames returns a list of a type's attribute names. When an external transaction type adds or removes an attribute, it must notify the aggregate account so it can modify its transaction mapping and tell its containers about the change.

For lists, an external transaction type has #addList: and #removeList: for adding and removing lists (although in the current implementation, an external transaction only has one list), #list:, which returns the schema for a list, #list:attr:type: and #removeList:attr: for adding and removing fields from lists, #list:attrType: to retrieve the type of a field in a list, and #listNames and #listAttributeNames: to retrieve the names of the lists and of the fields in a particular list. When an external transaction type adds or removes an attribute to or from a list, it must notify the aggregate account so it can modify its transaction mapping and tell its containers about the change.

In addition to notifying the aggregate account about changes to the transaction type, the external transaction type must also notify the user interface class (**TransactionInterface**), since the interface must also be changed. (However, this notification does not occur until the interface asks to be notified, because there might be many changes and the interface will just regenerate itself - it doesn't try to do an incremental change like the aggregate account does).

An InternalTransaction is simpler than an ExternalTransaction because it doesn't have any lists. It only has a dictionary of attributes and a type. For an InternalTransaction, the type is just the name of a type. Since the type definition is only used by an AggregateAccount and not by the transaction itself, it only needs to know the name of its type.

Chapter 4. Interesting Implementation Details

This chapter contains additional information about some of the more interesting parts of the Accounts framework. It also contains information about some of the peripheral features of Accounts that aren't integral parts of the framework.

4.1. Storing the Data Using Smalltalk

Although transactions could be stored in a wide variety of data structures, the current implementation uses a red-black tree [Corm90], which is one kind of balanced binary tree. The advantage of using this structure is that it allows an implementation in which each operation takes $O(\log N)$ time where N is the number of transactions posted to the account. The classes used to implement this are **TransactionTree** and **TransactionNode,** both of which are subclasses of **Object.**

4.1.1. Transaction Trees

A transaction tree contains information about the entire tree. It maintains references to its root node, the account (always a simple account) where the transaction tree is stored, and information pertaining to locating a specific internal attribute's partial results.

4.1.2. Transaction Nodes

A transaction node is a single node of the balanced tree. Much of the information it contains is general information used for implementing a red-black tree. A transaction node keeps track of its color (red or black), its parent, its left and right children, the key used for sorting (in this implementation, the date), the value the node represents (for this framework, the transaction), and a reference to the tree it is in. It also knows how to get the values of its transaction's fields.

In order to implement the tree so that calculating internal attribute values takes $O(\log N)$ time, a node cannot contain only the values from its transaction. If it did, then calculating an attribute for a given time would require looking at each node in the tree in the requested time range ($O(N)$ time). Storing the total of the internal attribute values up to a given transaction wouldn't work either, because then inserting a new transaction would require updating all transactions with later dates ($O(N)$ time).

Instead, each transaction node maintains partial results, which maps each internal attribute of the transaction node account to the total (for the internal attribute's field) of all transactions of the transaction node's descendants.

At first glance it might seem unusual to calculate a total for all nodes descended from the transaction node, because some of the nodes are later than the transaction node. And since the values of the transaction node's transaction aren't included in the transaction, the partial result values don't have any significant meaning in the Accounts framework.

However, this allows us to calculate an internal attribute's value at a node in $O(\log N)$ time by "combining" (using the operator of the internal attribute): a) the partial result value of the transaction node's left child; b) the value of the internal attribute's field on the transaction node's left child's transaction; c) the value of the internal attribute's field on the transaction node's transaction; and d) the result of combining a), b), and c) for all of the transaction node's ancestors having an earlier date.

When a new transaction node is inserted in the tree, its partial results can be updated by combining the left child's partial results, the left child's transaction fields, the right child's partial results, and the right child's transaction fields. Then each of the transaction node's ancestors is updated the same way. So inserting a new transaction node also takes $O(\log N)$ time.

4.1.3. Managing Partial Results

Since the collection of partial results depends on which fields are on the internal attributes of the node's tree's account, each node could potentially have a different list of partial results. To avoid making different kinds of nodes, and to make it easy to add new internal attributes to an account, Accounts stores the partial results in an array in each node. An internal attribute just needs to know the offset in the array that it stores its partial result. An account can compute the offset automatically when a new internal attribute is specified, so the user doesn't have to specify it.

The attribute definition process creates a default internal attribute when a user adds an attribute. The user is then free to change the attribute definition to something else. For this reason, it is important to avoid installing new attributes in the tree too early, since this could cause unnecessary changes. Therefore, new attributes are installed using lazy evaluation. The offset of a new internal attribute remains nil until a transaction is posted to the account or one of its children. At that point, any uninstalled attributes are installed in the tree, and their partial results are initialized.

Accounts must determine the offset of an internal attribute. This is complex because attributes may be added or deleted at various levels of the account type hierarchy. For example, suppose account A is the parent of account B and that account A contains 3 internal attribute definitions and account B contains 2 internal attributes. Account B would then have a total of 5 internal attributes, counting the ones it inherits from account A (assuming the names are distinct). If another internal attribute is added to account A, it would be account A's fourth attribute and it would therefore be given an offset of 4. However, account B already has an account with an offset of 4.

There is an interesting parallel between the way attributes act in Accounts and the way instance variables act in Smalltalk. Smalltalk instance variables are indexed by a number. The higher in the inheritance hierarchy an instance variable comes from, the lower its number will be. When an instance variable is added to a class, Smalltalk creates a new class with the same name and mutates all of the old class's instances to be instances of the new class, copying the instance variables to their new locations (i.e., offsets). This process is then repeated for all of the subclasses of the changed class, because the new instance variable will push the lower level instance variables out to a higher offset.

In a similar manner, when an internal attribute is added, the account calculates new offsets for its attributes and the transaction tree tells its nodes to update the positions of its partial results. Then, the account tells its children to do the same thing to themselves so that inherited attributes can have the same offset at each level of the type hierarchy.

4.1.4. Implementation of Partial Results

In order to understand partial results better, it might be helpful to look at how they are implemented. An internal attribute has several instance variables:

offset
- the offset of the partial result in a node's array of partial results

calcCode
- the time scope of the attribute

operator
- the message to send to combine partial results

It has the method:

```
identity
    ^self class legalOperators at: operator
```

that returns a value, which, when combined with another value using the attribute's operator, as in the following method, will return the other value:

```
combine: value1 with: value2
        ^value1 perform: operator with: value2
```

Operator '+' has identity value 0, '*' has identity value 1, 'min:' and 'max:' have positive and negative infinity respectively, and 'useOldUnlessNewNotNil:' has nil. Infinity objects are provided by the Smalltalk APOK extension file MetaNum.st.

A transaction node has several instance variables:

leftChild,. rightChild, parent
- the nodes directly adjacent to the current node in the tree

nodeValueRef
- the transaction stored at that node

partialresults
- the dictionary containing each attribute's offset with the attribute's partial result value for the node

key

- the date of the transaction used to sort the nodes in the tree

It has “helping methods”:

```
transactionValue: anAttribute
        "value from transaction for anAttribute"
        | val |
        val := self nodeValue getAttributeOrNil: anAttribute name.
        val isNil
                ifTrue: [^anAttribute identity]
                ifFalse: [^val]

subtreeTotalValue: anAttribute
        "return total of values from all transactions in the node's subtree"
        ^anAttribute combine: (partialresults at: anAttribute offset)
                with: (self transactionValue: anAttribute)attributeValue: anAttribute

previousChildrenTotalValue: anAttribute
        "return total of values from all nodes in the subtree no later than this one"
        | leftTotal |
        leftTotal := leftChild notNil
                                ifTrue: [leftChild subtreeTotalValue: anAttribute]
                                ifFalse: [anAttribute identity].
        ^anAttribute combine: leftTotal with: (self transactionValue: anAttribute)

nextEarlierParent
        "return the first ancestor that is earlier than the current node"
        parent isNil ifTrue: [^nil].
        ^parent rightChild = self
                ifTrue: [parent]
                ifFalse: [parent nextEarlierParent]
```

```
combinePartialResults: prList1 with: prList2
        "returns a dictionary containing the results of combining the two partial results"
        | temp operators |
        temp := Dictionary new.
        operators := tree operators.
        partialresults keys asOrderedCollection do:
       [:count | temp at: count
                   put: ((prList1 at: count)
                        perform: (operators at:
                                   count)
                        with: (prList2 at: count))].
    ^ temp.
```

The main working methods are **`updatePartialResults`**, which updates the partial results of the node, and **`calculate:,`** which returns the value of an attribute for the node.

```
updatePartialResults
    leftChild notNil
       ifTrue: [partialresults :=
               self combinePartialResults:  leftChild partialResults
                                  with: partialresults].
      rightChild notNil
        ifTrue:  [partialresults :=
                self combinePartialResults: partialresults
                                   with: rightChild partialResults]

calculate: anAttribute
    ^self nextEarlierParent isNil
        ifTrue: [self attributeValue: anAttribute]
        ifFalse: [anAttribute combine:
                 (self nextEarlierParent calculate: anAttribute)
                 with: (self attributeValue: anAttribute)]
```

Month-to-date and year-to-date attributes have to be computed differently. These attributes require two values, the value for the requested date and the value for the day before the first day of the month or year. The "difference" between these two values is calculated using the inverse operator. This is the corresponding operator to use to get the value change between the times of the two values. The inverse operator for '+' is '-', and for '*' it is '/'. Note that for non-cumulative operators, such as 'max:', 'min:', and 'useOldUnlessNewNotNil:', there is no inverse operator, so month-to-date and year-to-date attributes that have these operators cannot be computed using partial results.

4.2. Storing the Data Using a Database

The transaction tree is saved as part of the image. However, if multiple people need to have access to the same data, there must be some repository for it to be stored in. And if they need to have *concurrent* access to the same data or the ability to access the data independently from the application, then a database will be necessary.

The two main data models on which database management systems (DBMSs) are based are the relational and object oriented models. Two advantages of relational DBMSs are that the data model is simple and a lot of people are using them. One major advantage of using object oriented DBMSs with an object oriented software system is that the data model is more closely aligned with the data model of the program so the greater complexity of the data in an object oriented program can be represented in the database. Another advantage is efficiency when combining data from related entities. In relational DBMSs, the joining of data occurs at run time. In object oriented DBMSs, there are navigation paths built into the database, so one object can contain a reference (some sort of a pointer) to related objects.

In Accounts, transactions are very simple. Each kind of transaction can be expressed as one or two (when a list is used) relations and the data stored in a transaction can be a string, a date, or a number. There is no need to model inheritance relationships. Relational DBMSs support these data types and many of them are now supporting references from one tuple to another (which could be useful in transactions having a list), so there is no reason a relational DBMS couldn't be used with Accounts.

There are two ways to access a relational DBMS, statically and dynamically. Dynamic access means that the SQL queries (or whatever method is used to access the database) are determined at run time. Static access means that the queries are predefined.

The tradeoff is flexibility versus performance. Static access allows far superior performance. Some relational DBMS manufacturers have Smalltalk drivers that provide the connection between the Smalltalk environment and the DBMS. They all use dynamic SQL in which the query is generated in Smalltalk and is sent to the DBMS, and the DBMS parses the query, executes it, and returns the result to Smalltalk.

4.2.1. Interface Between Smalltalk and the Database

Accounts uses a static access solution. With static access, the database calls are in C programs which are executed from Smalltalk as user primitives. There are two primitives - one to insert a tuple into a database and another to retrieve a tuple from the database.

A database interface for Accounts was written using a rather unusual DBMS called Unify 5.0. Instead of using SQL queries, it provides C functions for accessing the database. The Smalltalk code would have been the same for static SQL access. The only difference is that the C programs would have had SQL calls instead of C function calls.

In order to retrieve data that was previously inserted into the database, there must be a way to match a particular transaction node to a tuple (or tuples) in the database. Finding a tuple presumes knowing what relation it's in. Transaction type names must be unique within an aggregate account. For database access, we extend that requirement to state that transaction type names must be unique within the application. With this requirement, the relation name can be the same as the transaction type name. For lists, which are stored in separate relations, the relation name is the transaction type name concatenated with the list name.

Next, there must be a way to find the tuple(s) for the appropriate transaction. There is no guarantee that a unique key will exist among the transaction attributes. Therefore, each transaction should have a unique ID. Many DBMSs have a built-in mechanism for generating unique tuple IDs within a relation. Using this mechanism, the tuple ID can be used as a transaction ID. For list tuples, which are stored in a separate relation, the transaction ID must be copied to the list tuple. Note, however, that some relational DBMSs and all extended relational DBMSs (hybrids of relational and object oriented DBMSs) provide nested relations, where a tuple in one relation can cluster or reference tuples in another relation. In this situation, it's possible to avoid storing the transaction ID in a list tuple.

4.2.2. The TransactionReference Class

The permanent home of transaction data is the database. The transaction tree should only retrieve actual transactions as they are needed. This is the same principle used in most object oriented DBMSs. They use an object descriptor [Kim88] (also called a *handle* [Catt91]) which points to the main memory copy of an object if the object is there, and points to the database otherwise.

Accounts includes a **TransactionReference** class which acts has a handle for the transaction. It is an example of the Proxy pattern [Gamm94]. A transaction reference contains a transaction ID, a reference to the transaction, and the transaction type of the transaction. Accounts implements the handle mechanism by setting the transaction to nil if the transaction is not in the transaction tree. With database access, the pointer to the transaction in a transaction node now points to a transaction reference instead of an internal transaction.

When an Accounts application wants to look at a transaction (e.g., when evaluating a query) the transaction node sends the #getValue message to the transaction reference which checks whether transaction is nil. If it isn't, it returns the transaction. Otherwise, it creates a new internal transaction instance (garbage collection should have deleted the old one) and retrieves the transaction from the database by calling a user primitive. Each call will retrieve one tuple, so if the transaction includes a list, then multiple database calls will be needed. The user primitive will return the tuple as an array of values, so it's important to return them in the same order as the names are stored in the transaction type. The transaction reference then copies the values into the transaction.

When the user creates a transaction, the transaction node will create a transaction reference, which will store the transaction into the database by calling the user primitive to insert a tuple. The primitive will return the transaction ID for the new transaction.

4.2.3. The User Primitives that Access the Database

When retrieving a tuple from the database, the transaction reference passes the primitive the name of the relation (transaction type) and the tuple ID. The actual database access depends on the needs of the application and what the DBMS provides. However, there are some things that will be the same.

Smalltalk provides a C interface for creating certain kinds of Smalltalk objects from certain types of C data items, including integers, floating point numbers, strings, and arrays. The primitive will return the tuple to Smalltalk as an array of Smalltalk objects. The size of the array will depend on how many attributes the tuple has. The primitive can determine this by querying the database schema or, if the primitive contains hard-coded queries, the size of the array can be hard-coded. Smalltalk does not provide an interface for dates. Therefore, the date must be converted to an integer and then converted back to a date by the transaction reference (alternatively, the database could just store the date as an integer, but this would depend on the users' needs). Accounts uses the formula (year * 10000) + (month * 100) + day, but any valid formula can be used as long as the Smalltalk and C versions match.

The primitive to insert a new transaction is in many respects the reverse of the primitive to read from the database. The array of values is passed in, along with the relation name and the transaction ID (which should always be 0 unless the framework is extended to allow modification of transactions). The schema data shouldn't be necessary, because it's possible to ask a value passed from Smalltalk for its type using Smalltalk-provided C functions such as UPisInteger(), UPisString(), etc. The fields passed in from Smalltalk should be converted to their C equivalents (any date fields which have been passed in as integers should be converted back to dates) The primitive then inserts the tuple, and returns the transaction ID of the new row using the UPreturnHandle() C function.

4.3. Saving and Loading an Application

Once an accounting system is created, it can be saved in a Smalltalk image file. However, Smalltalk images are very large, so they can be difficult to transfer between machines or users. They also might contain information specific to a user, such as additional classes and objects from other frameworks. Therefore, the Accounts framework has a mechanism for saving the objects that were created for a particular application. The file that Accounts creates is much smaller than an image, and only contains information specific to Accounts. Therefore, people with two very different images can transfer a file containing an Accounts application, provided they both have the same version of the Accounts framework.

Smalltalk provides a service called BOSS (Binary Object Streaming Service), which creates a stream of objects that is typically stored in a file. One difference between BOSS and a DBMS is that BOSS does not provide query facilities, so any data that a user wants to query without entering the application should be put in a database instead.

The BOSS interface to Accounts was initially written for a prototype version of Accounts that created new subclasses and generated new code. Now that Accounts no longer does these things, the BOSS interface is much simpler.

The Accounts framework accesses BOSS through class methods in **Account.** When an object is written to a BOSS stream, all of the objects connected to it are written to the stream with a few exceptions. Classes are not written, so class variables must be written separately. In the Accounts framework, only **Account** has class variables. So the class method writes the type dictionary to the BOSS stream. Since all accounts either have a template or are a template (or both), and since each template stores the accounts cloned from it, all of the accounts in the application are connected to the type dictionary. And all of the objects in the framework (except some of the interface objects) are connected to accounts. So saving the type dictionary is the only thing necessary to save an application.

Also, reading block closures that BOSS previously wrote can cause problems. At one time Accounts used sorted collections, which contain block closures. Since then, the code has been modified to remove them, so this problem does not exist in the current BOSS/Accounts interface.

The file containing an application can be transferred, copied, and distributed so that several can run the application by loading it from the file. **Account** contains another class method to load the application from a BOSS file. It simply reads in the type dictionary, which will automatically read in all of the other application objects.

4.4. Launcher Changes

The entry point into Accounts is through the VisualWorks [Parc92] Launcher, which now has an Accounts option. The option was added by modifying the #smalltalkLauncherView and #initializeLauncherMenu class methods of the **UIVisualLauncher** class to call an **Account** class method called #accountsMenu. Figure 18 shows the new Launcher and the Accounts menu options which are used to save and load the accounting system using BOSS and to create and find accounts. Typically, a user would start up VisualWorks, load the accounting system (if it's not already stored in the image), and find an account to start with (often the top-level aggregate account).

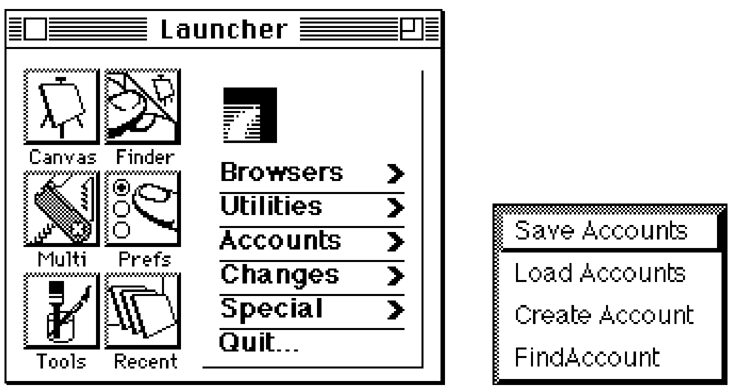

Figure 18

4.5. Posting Transactions

When a user posts an external transaction to an aggregate account, the aggregate account often creates multiple internal transactions. One or more of these internal transactions may contain an error (such as an undefined account). In the event that an error occurs, none of the internal transactions should be stored and the user should have the opportunity to correct the error and repost the transaction.

This functionality is provided by a two-phase commit process based on Smalltalk's error handling capabilities. When the user presses the Accept button on the transaction entry window (see Figure 16), the VisualWorks interface method responsible for calling the aggregate account's #postFromTransaction: method creates an accountNotFoundSignal which handles the situation when an account is not found. If an account is asked for a component that it does not have, it raises an exception using this signal. The exception handler will present an error to the user such as that shown in Figure 19 and will then reshow the transaction entry window to the user for correction.

Figure 19

When an internal transaction reaches a simple account with no errors, the simple account cannot just store it directly, because another internal transaction from the same external transaction might have an error, and it would be difficult to undo the posting of the valid internal transactions.

Instead, the aggregate account to which the user posts the external transaction maintains a transaction log that contains the internal transactions that reached the simple account with no errors. Before starting the posting process, the aggregate account empties out the transaction log and creates a recordPostSignal that defines how to handle a valid internal transaction. When the internal transaction reaches the simple account, the simple account just raises the recordPostSignal, which adds the internal transaction to the aggregate account's transaction log.

After the external transaction makes it through the post phase with no errors, the store phase stores the internal transactions from the transaction log into their appropriate simple accounts. Note that this simple implementation assumes that an aggregate account will only process one external transaction at a time. Processing multiple external transactions at once would involve creating a unique transaction ID and storing that ID in the transaction log with the internal transactions. Rolling back or committing an external transaction would then require knowing the transaction ID and only processing the internal transactions in the transaction log with a matching transaction ID. This would be a relatively simple change.

The two-phase nature of this process is similar to the commit process used in distributed databases[Ullm88]. A distributed database transaction is atomic. It can run on several machines, but the transaction cannot commit until all of the sub-transactions vote to commit. The posting process in Accounts is a distributed procedure in the sense that a transaction is split into multiple pieces, each of which are processed separately, and the validity of a transaction depends on all of its pieces being valid.

Chapter 5. Case Study

Up to this point, I've described various features of the Accounts framework, but I haven't described how they all fit together. This chapter gives a complete view of a single application, and shows how to create a complete application using Accounts.

Diane's Bookstore (a fictional company) wants a system that will help keep track of the store's inventory, personnel, customers, creditors, and money. Although some businesses have separate systems for these things, Accounts encourages people to integrate their accounting systems so that a user only needs to enter a transaction once. For example, selling a book will affect inventory, customer information, and personnel information (how many books each person sold). If there were separate systems, the user would have to remember to enter information three times.

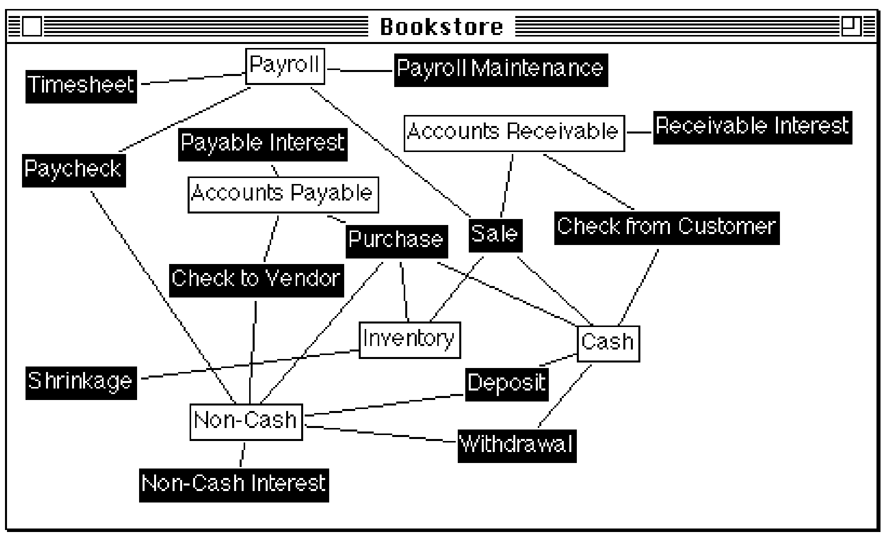

Figure 20

5.1. Bookstore Accounts

The top-level account for Diane's Bookstore is called Bookstore. Figure 20 shows its contents. If there were more than one store, Diane could make Bookstore a template account and each of the individual bookstores could be cloned from the template. A new top-level composite account could combine information from the stores. For this case study, however, assume there is only one store.

The components of Bookstore are Payroll, which stores personnel information, Accounts Receivable, which stores customer information, Accounts Payable, which stores creditor information, Inventory, which stores information about items the book-store stocks (mostly books), Cash, which keeps track of the store's cash assets, and Non-Cash, which keeps track of the store's non-cash assets. All of these components are composite accounts except Cash, which is a simple account.

Each composite account contains at least one template account. This is done for convenience rather than necessity. Although there needs to be a template for each composite account, the templates don't have to be stored in the composite. Storing them there makes it easier to remember which templates are used in which composites. Payroll contains a simple account called Employee which is a template for all the employees of the store. The template contains several attributes shown in Table 2. Notice that although there's an internal transaction field called #paid (as indicated in the definition of paidYTD), there is no attribute called paid. This is because Diane isn't interested in knowing the total amount she has paid her employees - just the amount for a particular year. So even though the default expression makes it look like there might be a connection between the two names, there is really no connection.

Attribute Name	**Function of Time Expression**
sold	self totalFor: #sold with: #+
soldYTD	self ytdFor: #sold with: #+
hoursWorkedMTD	self mtdFor: #hoursWorked with: #+
paidYTD	self ytdFor: #paid with: #+
hourlyWage	self currentValue: #hourlyWage
monthlyPay	self hoursWorkedMTD * self hourlyWage
name	self currentValue: #name
address	self currentValue: #address

Table 2

Although hourlyWage is a current value attribute, it could have been defined as a constant value of 0. The constant attribute definition couldn't be inherited like the other attribute definitions, because each employee potentially has a different hourly wage. It would have to be redefined for each employee. However, the template still needs the interface definition of hourlyWage in order for the definition of monthlyPay to be valid. Although the current value attribute definition doesn't have to be redefined for each employee, a transaction will be required for each employee to set the hourly wage. However, this isn't a problem, since the employee name and address are already set using a transaction.

The Inventory component account contains a simple account called Inventory (an account can have the same name as its container) which is a template for all the inventory items the store stocks. It contains attribute definitions that are common to all inventory items in the store. There are three internal attributes, sold, purchased, and lost, which tell how many of a particular inventory item have been sold, purchased, and lost. One other attribute, onHand, tells how many of a given item there are in stock. The onHand attribute is defined as (self purchased - self sold - self lost).

The store currently carries two kinds of inventory items - books and greeting cards. There is a template account for each of these kinds of items. These two template accounts are cloned from the Inventory template account, so they inherit all of Inventory's attributes. They also have additional attributes. Book, which is the template account of all books the store stocks, has two "current value" attributes, author and title. Greeting Card, which is the template account of all greeting cards the store stocks, has one "current value" attribute, occasion.

The Non-Cash component account contains a simple account called Bank Accounts that is a template for all the store's bank accounts. If the store had other kinds of non-cash monetary assets, it could use a hierarchical template structure like Inventory has. Bank Account has two internal attributes, withdrawn and deposited, and it has one other attribute, currentAmount, which is defined as (self deposited - self withdrawn).

Cash, the simple component account, has attributes similar to Non-Cash. It has two internal attributes, incoming and outgoing, which keep track of additions to and subtractions from the cash supply at the store. The other attribute, onHand, is defined as (self incoming - self outgoing).

Accounts Payable and Accounts Receivable have the most complex structure, but they are similar to each other. Accounts Payable contains an aggregate account called Creditor which is a template for all the companies to which the store might own money (typically, publishing and greeting card companies). Since Creditor is an aggregate account, it contains transaction types and accounts. Figure 21 shows its contents view.

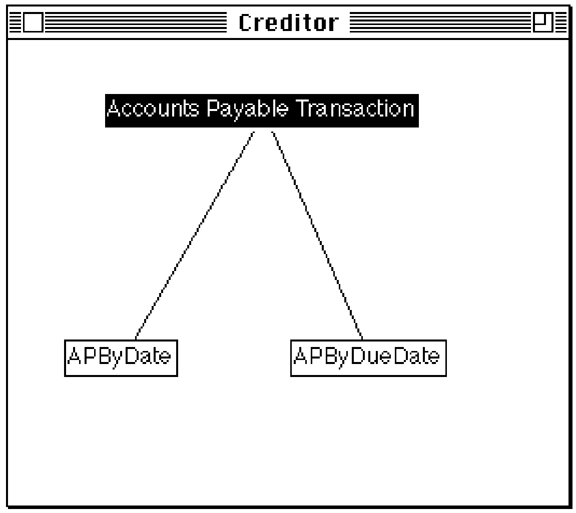

Figure 21

The name of the transaction type (Accounts Payable Transaction) is the same as the name of the internal transaction type that was generated for the higher-level aggregate account (Bookstore). The fields in the transaction type determine which fields will be in the transactions that Bookstore sends to Accounts Payable. The two simple component accounts, APByDate and APByDueDate, are distinguished by the date by which transactions are sorted (transaction date vs. due date), as described in Chapter 2. These two components are also templates, so when Creditor is cloned, its children's components will be children of APByDate and APByDueDate and will have the same names as APByDate and APByDueDate. In fact, the contents view of Creditor's children will look just like Figure 21.

APByDate has two internal attributes, paid and purchased. purchased gives the total amount purchased from that vendor, while paid gives the total amount paid to the creditor. APByDueDate has an attribute called charged, which gives the amount charged to the creditor. Creditor has an attribute called due, which calculates the amount due to the creditor as (APByDueDate charged - APByDate paid). Since interest is calculated differently by different creditors, it will simply be counted as an increase in the charged attribute, instead of trying to calculate it as a separate attribute.

Accounts Receivable is similar to Accounts Payable. It contains an aggregate account, Customer, which is a template for all customers to whom the store has granted credit. Customer's contents view, in Figure 22, looks similar to Creditor's contents view. ARByDate has one internal attribute, paid, and ARByDueDate has two internal attributes, charged and chargedMTD. Note that unlike with accounts payable side, the bookstore doesn't try to keep track of the total amount a customer has purchased because much of the time a customer pays in cash, so there's no reason to ask the person's name.

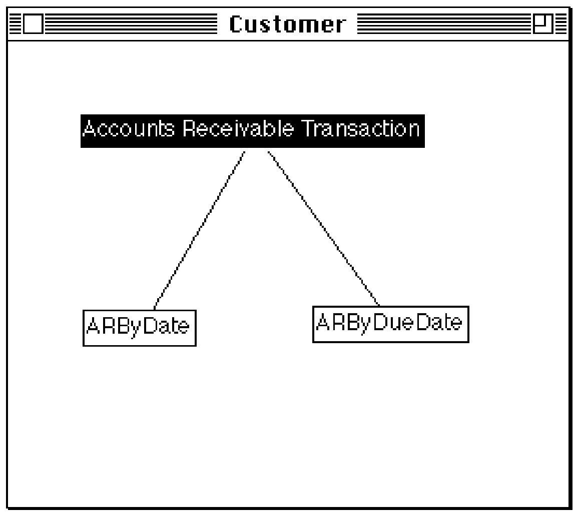

Figure 22

Table 3 shows most of the attributes defined for Customer. Although interestRate is defined as a constant attribute, it could also be defined as a current value attribute. In that case, it would require a transaction to set it and update it.

Attribute Name	**Function of Time Expression**
paid	ARByDate paid
charged	ARByDueDate charged
chargedMTD	ARByDueDate chargedMTD
due	self charged - self paid
interestRate	0.06

Table 3

While creating the Accounts Receivable portion of the system, Diane realized that she wanted to be able to calculate interest due for her customers. The current system doesn't have the ability to generate external transactions automatically, but it would at least be nice to have the system calculate the amount. Amounts whose due dates are in the current month should not have interest due on them, so the interest amount would be calculated by subtracting the total charged this month from the total amount due, and multiplying that by the interest rate. But what if the result of that subtraction is negative? This would happen if the customer overpaid or prepaid. The interest rate should only be nonzero if the result of the subtraction is positive.

The solution to this problem is that Diane will create a new kind of function of time (see section 3.4.2) called **PositiveTestFOT.** A positive test FOT will test another function of time. If the other function of time's value is positive, the positive test FOT will evaluate to the value of its operand. Otherwise, its value will be zero.

Diane goes through several steps to define this new kind of function of time:

1. Create a new Smalltalk class.

   She creates a Smalltalk class called **PositiveTestFOT**, making it a subclass of **FunctionOfTime**. Since this new class represents a simple unary operation, it only needs one instance variable, operand.

2. Define the methods needed for a new function of time.

   Section 3.4.2 points out that there are 4 methods which every function of time needs. The value: method, which initializes the instance variables, is simply defined as operand := anFOT where anFOT is the input to the method. The printing method, printOn:, simply tells the operand to print itself on the input stream, and then it prints the word positive. The method for evaluating a function of time, atTime:forAccount:, calls the atTime:forAccount: method for its operand, tests if it's greater than 0, and returns either the value of its operand's atTime:forAccount: method or 0. Finally, the fieldList method, which returns the names of the transaction fields to which a function of time refers, simply returns the result of its operand's fieldList method.

3. Define the method(s) used to create a new function of time.

A positive test FOT is indicated in the attribute definition by typing the word positive after another function of time expression. Therefore, the **FunctionOfTime** class needs a method called positive, which will create a new Positive Test FOT. This method's definition is ^PositiveTestFOT new value: self.

Once the class is created, Diane can define an interest attribute for Customer using the definition (self due - self charged) positive * self interestRate.

5.2. Bookstore Transactions

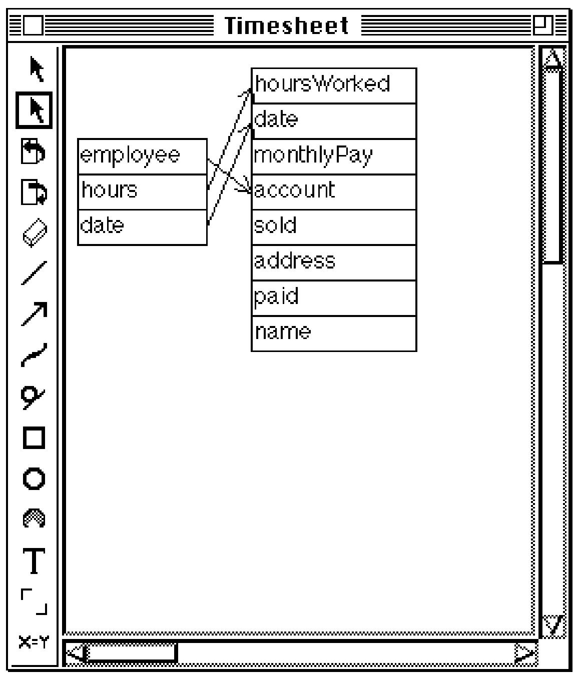

Figure 23

Each of the transactions in Bookstore affects from 1 to 4 component accounts. A Timesheet transaction records the hours an employee has worked. It only affects the Payroll component. Its transaction mapping drawing in Figure 23 shows that a

Timesheet has three fields - employee, which becomes the account on Payroll's internal transaction, hours, which translates to hoursWorked, and the ubiquitous date.

Notice how the general field name "hours" on the Timesheet becomes the more specific name "hoursWorked" on the Payroll Transaction. This is because a Timesheet has a very simple purpose, so the name "hours" doesn't need to be qualified further (although one can imagine a more complicated timesheet that a software consulting firm might use, with hours broken down by project using multi-value fields). On the other hand, a Payroll Transaction is more complex and can get information from one of several sources. Therefore, a more specific name is needed to understand what the field is for.

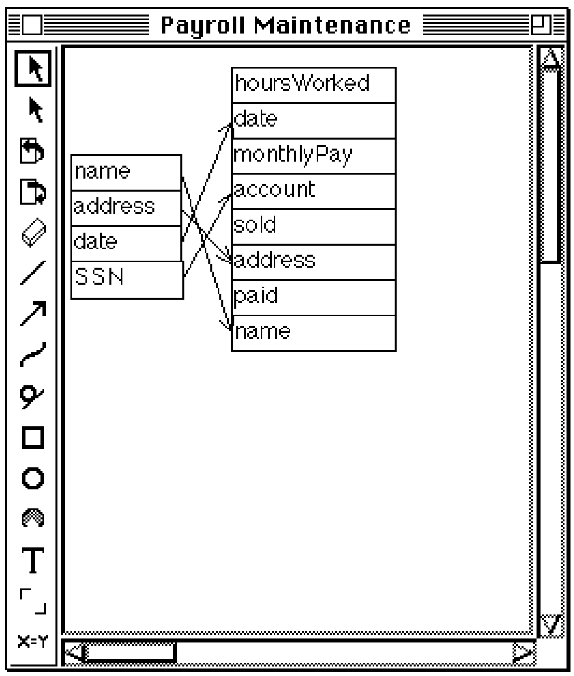

Figure 24

A Payroll Maintenance transaction updates the current value fields and gives an example of how "current value" fields should normally be updated. (For simplification purposes, this case study does not use maintenance transactions for its other "current value" fields.) A Payroll Maintenance transaction only affects the Payroll component. Its transaction mapping drawing in Figure 24 shows that the name and/or address are updated with a Payroll Maintenance transaction, and that the account is identified using the social security number on the transaction. Because of the nature of "current value" fields, the previous values of name and address are still accessible by performing a query using a date earlier than the date of the transaction.

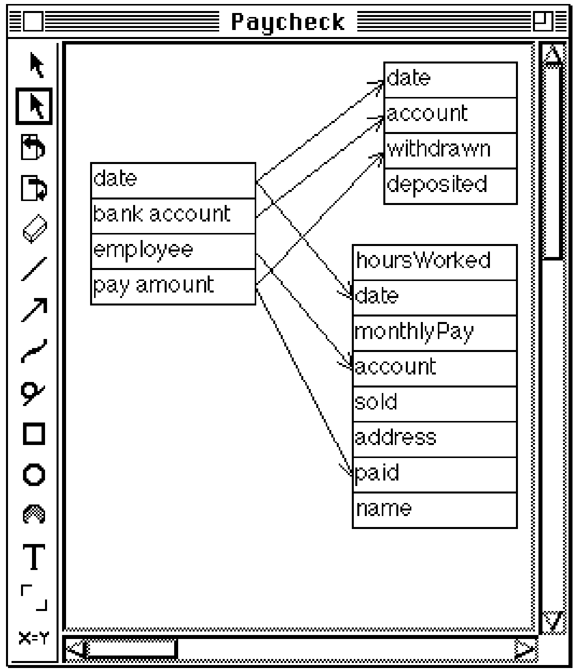

Figure 25

A Paycheck records a check written on one of the store's bank accounts and given to an employee. Its mapping drawing in Figure 25 shows that a paycheck's effect is to withdraw money from a specific bank account and to add to the amount paid to an employee.

Bank accounts (i.e., Non-Cash) are also affected by Non-Cash Interest transactions. Since each bank has its own complicated and often-changing method of calculating interest, Diane has decided not to try to incorporate these calculations into her system, but rather to enter them manually when she gets the account statements from the bank. The transaction mapping drawing in Figure 26 shows that interest payments are treated as money deposited into the account.

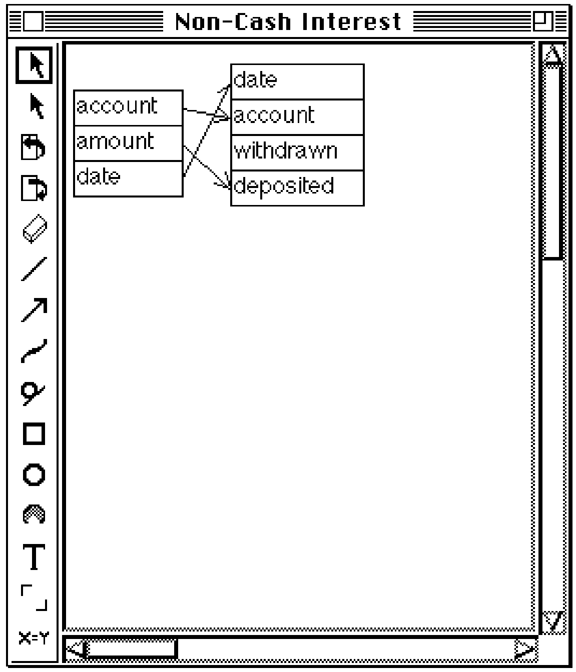

Figure 26

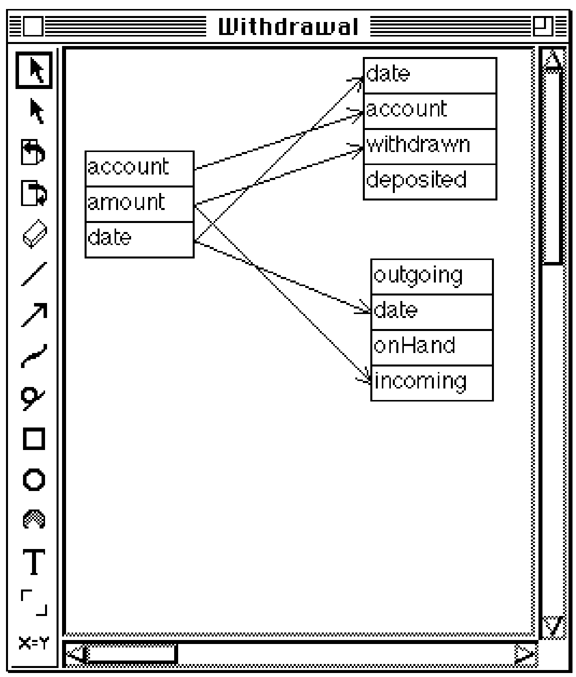

Figure 27

As one might suppose, Deposit transactions take money from cash and put them in bank accounts, and Withdrawal transactions take money from bank accounts and add them to cash. Although Accounts does not handle account transfers directly (an external transaction cannot send two transactions to the same account (i.e., Non-Cash) except when using multivalue fields), an account transfer would consist of a withdrawal followed immediately by a deposit. The Withdrawal transaction mapping drawing in Figure 27 shows that the amount gets withdrawn from Non-Cash and gets added to incoming Cash. The Deposit drawing looks the same except the arrows go to deposited instead of withdrawn and outgoing instead of incoming.

The two most complex kinds of transactions in both structure and mapping are Purchases and Sales. Purchases generally reflect information from an invoice. They contain information about the invoice, how it was paid for, and what items the store received. An invoice can be paid for in three ways - by cash, check, or credit (assumed here as credit directly with a vendor rather than with a third party).

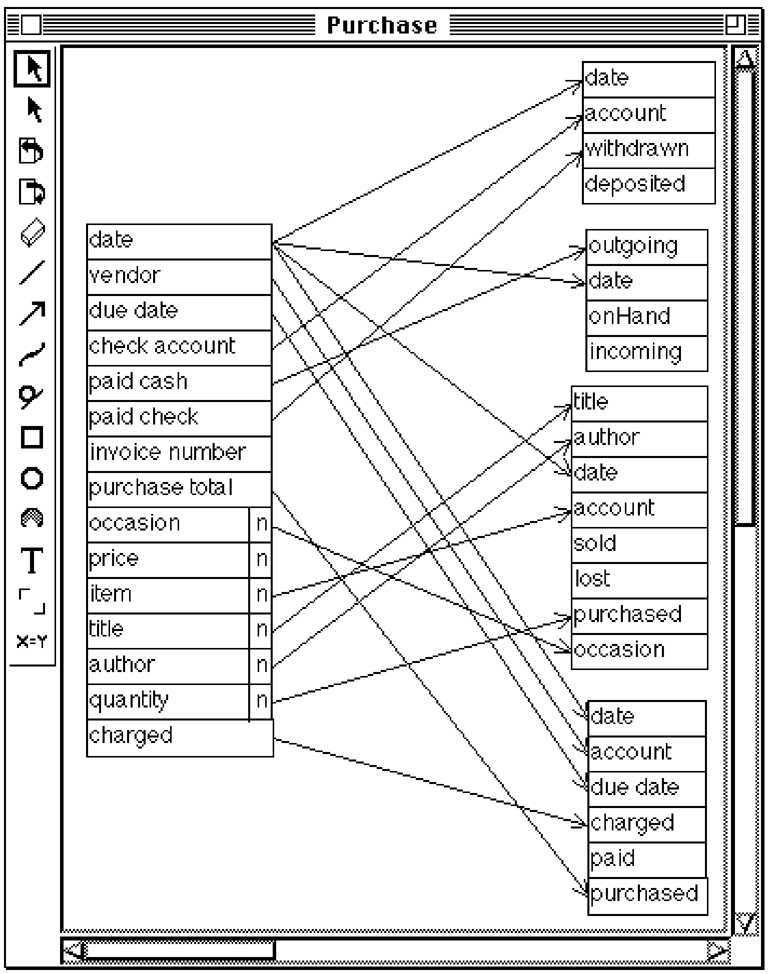

Figure 28

Figure 28 shows how a Purchase transaction affects four different accounts. Paying by check causes the check amount to be withdrawn from a bank account. Paying by cash causes the outgoing amount in Cash to increase. Paying by credit adds to the amount charged to the vendor of the invoice. Since Diane wants to keep track of the total amount purchased from each vendor, the total purchased amount (whether or not any portion of the purchase was charged) is also updated in Accounts Payable. Finally, Inventory is updated using the multi-value fields corresponding to the items received with the invoice. A separate internal transaction will be created for each item on the invoice. The Inventory "current value" attributes title and author can also be set using a Purchase transaction (though they should never need to be changed). A better but more complex design would have an Inventory Maintenance transaction similar to the Payroll Maintenance transaction discussed earlier.

Overall, a Sale transaction's effect is almost the opposite of a Purchase transaction's with two exceptions. A Sale transaction cannot directly affect Non-Cash. This is because Diane hasn't installed debit card access to allow money to go directly from the customer's bank account to her bank account. (Ten years ago she wrote an article on debit cards (specifically, the "smart cards" used in France) and recognized the potential security risks, so she's decided not to participate in that part of the Technical Revolution.) However, she does allow good customers to carry a balance (that's what Accounts Receivable is for). She also allows customers to pay by check, but those checks are counted as cash until they can be deposited in the bank. Diane wants to keep track of how much each employee sells, so a Sale transaction, unlike a Purchase transaction, affects Payroll.

Figure 29 shows how a Sale transaction affects Cash, Payroll, Accounts Receivable, and Inventory respectively. As with Purchase, a Sale transaction creates one transaction for each different item sold.

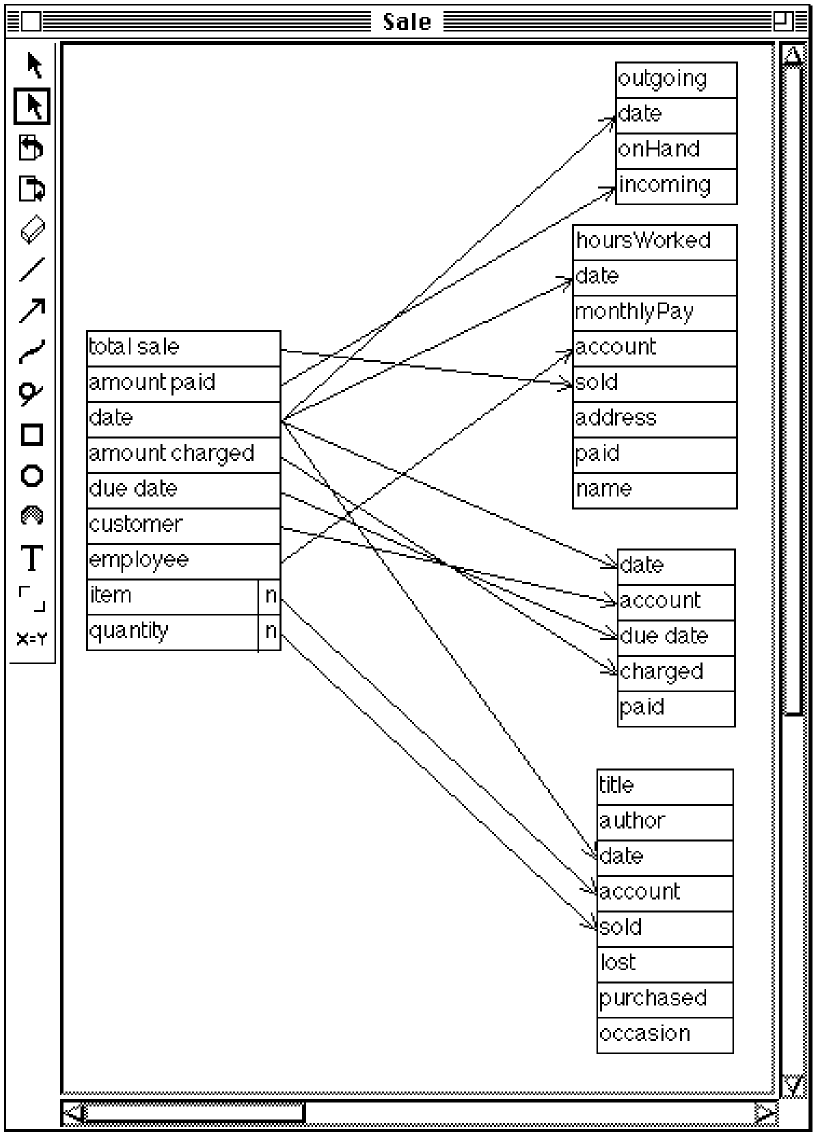

Figure 29

Check to Vendor transactions and Check from Customer transactions are similar to each other. A Check to Vendor takes cash from a bank account and increases the amount paid to a vendor in Accounts Payable. A Check from Customer increases the amount paid by a customer (i.e., decreases the amount owed) and adds to the money in Cash. Figures 30 and 31 show the transaction mapping drawing for Check to Vendor and Check from Customer respectively. Since a Check from Customer goes to cash, it doesn't need the bank account that Check to Vendor needs.

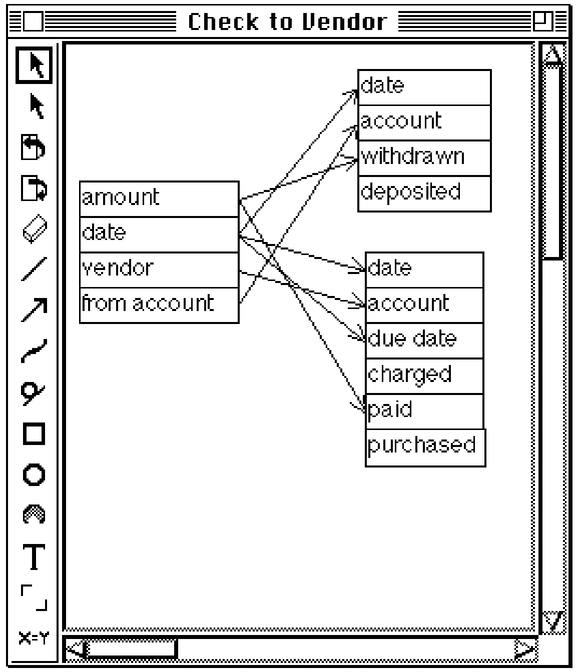

Figure 30

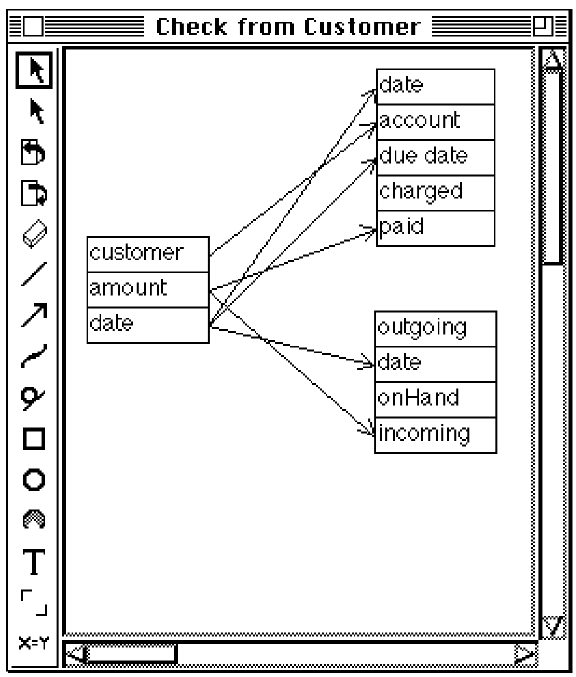

Figure 31

Payable Interest and Receivable Interest transactions indicate interest that is now due to a creditor or from a customer. As stated earlier, the system calculates the receivable interest amount, but the transaction is entered manually. Figure 32 shows the mapping for Payable Interest. The mapping for Receivable Interest looks almost exactly the same. The only important difference is that instead of a field called vendor, there is a field called customer.

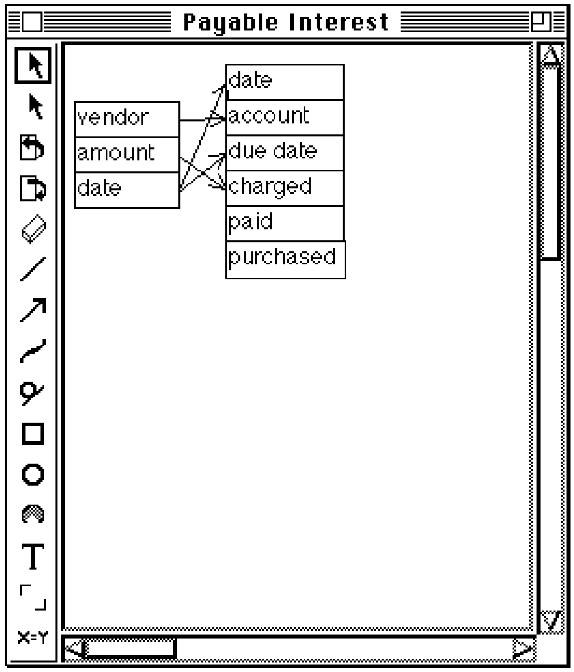

Figure 32

Finally, a Shrinkage transaction is needed when someone discovers that an inventory item has mysteriously disappeared. Shrinkage can occur for a number of reasons. Sometimes items get thrown out by mistake or an employee (or the owner) takes a book home intending to bring it back, but no one else knows it. When the item is later discovered or is brought back, a negative number of items in the Shrinkage transaction could reflect this (mysterious reappearances don't happen often enough to justify a new kind of transaction). Spoilage could also be considered shrinkage (e.g., when someone spills coffee on a paperback book). However, one of the inescapable annoyances of owning a retail business is the practice of shoplifting, and this is the most common source of shrinkage. Figure 33 shows the very simple effect Shrinkage has on Inventory.

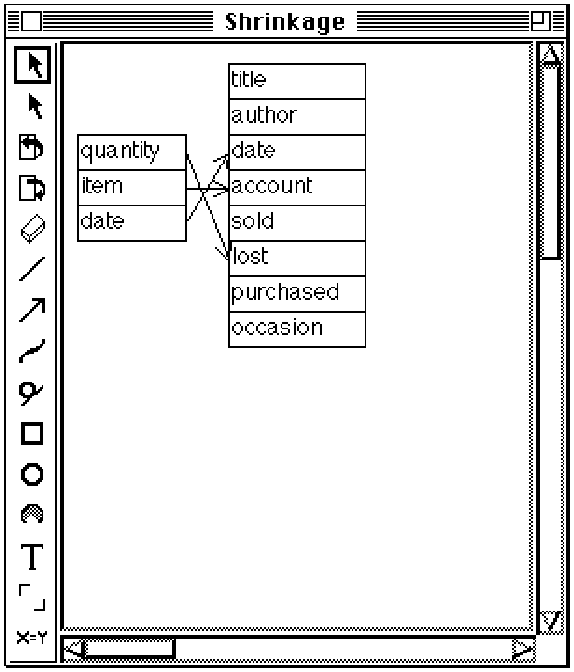

Figure 33

5.3. A Transaction Lifetime

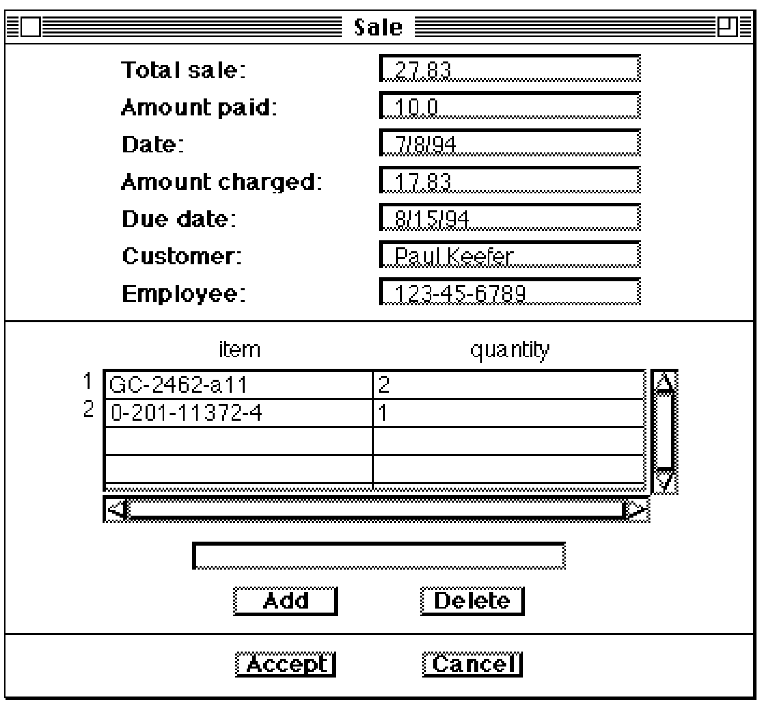

Figure 34

This section traces the life of a Sale transaction from its creation through its storage and use in queries. First, the user enters a Sale transaction as shown in Figure 34, and posts it to Bookstore.

As shown in Figure 29, Bookstore will create 4 kinds of internal transactions from this external transaction. The first will be sent directly to the Cash account where it will be stored, and will just have the values from the Date: and Amount paid: fields from the original transaction.

The second internal transaction will be sent to Payroll, which is a composite account. The value from Employee: on the original transaction will become the account field in the internal transaction. Payroll will then look for its component named the same as the employee, and send the transaction to the component, where it will be stored. This transaction will also have the Date: and Total sale: fields from the original transaction.

Bookstore will create 2 internal transactions corresponding to the two different inventory items sold. They will both be sent to Inventory, which will send them to two different components based on the item numbers from the original transaction. The quantity and Date: fields are on these transactions as well.

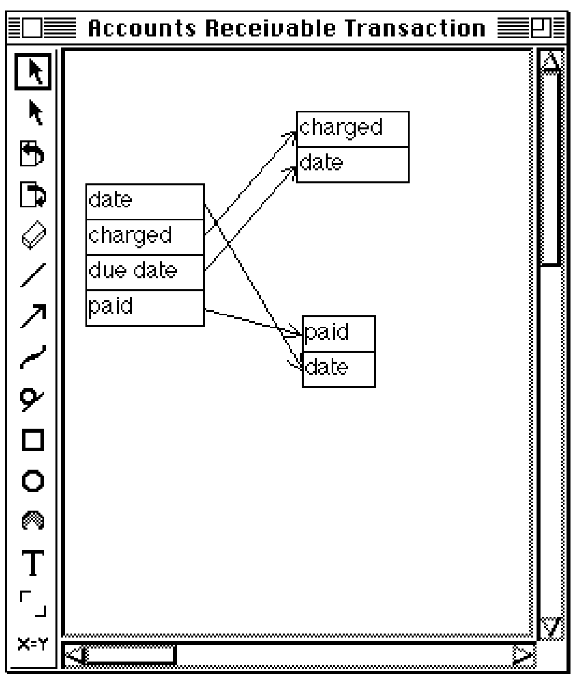

Figure 35

The other internal transaction is for Accounts Receivable, since there is an amount charged. It will put both dates on the transaction, and the account will come from the Customer: field from the original transaction. It will also have the Amount charged: field. Accounts Receivable will send the internal transaction to its component whose name matches the customer name. However, the customers are aggregate accounts (see Figure 22), so the customer will split this transaction into two internal transactions based on the transaction mapping drawing in Figure 35. The transaction that goes to the account that is sorted by date will have only a date on it, since the customer did not pay anything toward the amount he owed.

After all the transactions have been stored, you can then see their effects. One way is to look at the contents views of the simple accounts where the transactions are stored. For example, the Cash contents view in Figure 36 shows the transaction.

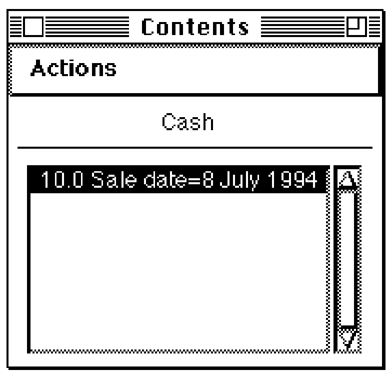

Figure 36

The other way to see the results of the transaction is to query the account for an attribute value. Figure 37 shows the onHand attribute value for the current date. There was no cash on hand before this transaction was posted, which is why the attribute value matches the transaction value. Figure 38 shows the effect the Sale transaction had on the employee's sold attribute.

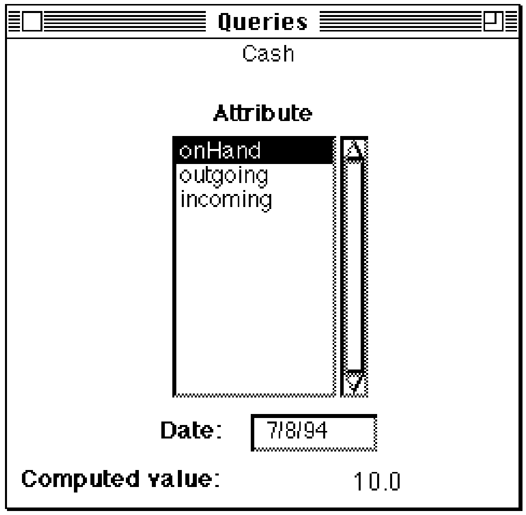

Figure 37

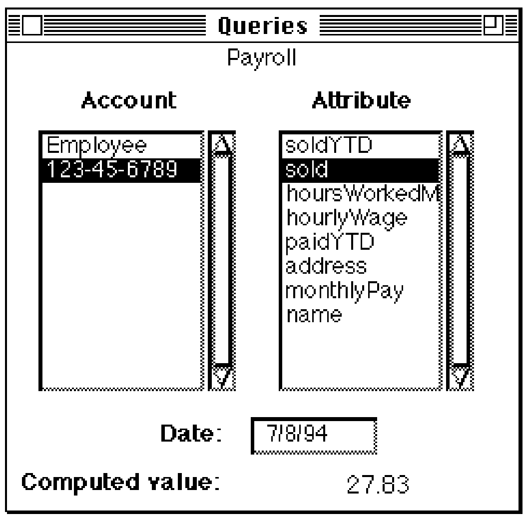

Figure 38

Chapter 6. Future Directions

There is every reason to believe that people will continue to work on the Accounts framework to add more functionality. This chapter contains some suggestions for those who wish to continue working on it.

6.1. Mapping Language

Currently, the mapping language for indicating the sources of internal transaction fields is very simple. The field either comes from an external transaction field or it is nil. Clearly, there should be a richer variety of sources. A field might have a constant value, or might be calculated by a formula involving multiple constants and multiple transaction fields. In fact, the mapping language should provide functionality that is similar to the attribute definition language.

6.2. Attribute Definition Language

It was very convenient to be able to use the Smalltalk compiler to parse attribute definitions. However, the Smalltalk syntax, while relatively easy for a programmer, is not friendly enough for a non-programmer. Also, while using `#doesNotUnderstand:` is a reasonable solution for a prototype system, it's entirely possible that a user might define an attribute name that is the same as a method name. It would be a good idea to develop a special purpose attribute definition language with its own context.

6.3. Transfer Transactions

In the case study, one kind of transaction that the Accounts framework couldn't support are transfer transactions from one bank account to another. There are two ways to solve this problem.

One way is to allow the user to define the aggregate account so that it will send two transactions to the same composite. The system currently cannot do this. One problem is the way the aggregate account contents interface works. It draws its lines from the center of a transaction rectangle to the center of an account rectangle, so it's impossible to have two lines going from the transaction to the account (unless one line on top of the other can be considered two lines, but then it will be invisible to the user). This limitation also prohibits general ledger transactions which have a debit and credit account and would require splitting the transaction in two and sending them both to a general ledger composite account.

The second reason an aggregate account can't send two transactions to the same component is that the system is set up to have one internal transaction type for each component account. This problem could be solved by having each component account keep track of its own internal transaction types and give them to the aggregate account when it asks for them. Currently, the aggregate account keeps track of them.

The other way to solve the transfer transaction problem is for an external transaction to generate another external transaction, something it can't currently do. Since a transfer transaction is equivalent to a deposit and withdrawal, a transfer transaction could generate one of each kind of external transaction. This would also be useful for things like automatically generating transactions for interest due.

6.4. Additional Field Types

In the current system, transaction fields have one of three types - date, number, or string. There are some situations where other types would be helpful. For example, books, which are a kind of inventory item in the case study, have an author attribute which comes in as a string in a transaction. However, many books have multiple authors, so rather than having a single string containing all the authors, a set of strings would give a better representation of the data.

Representing an author as a string or set of strings is acceptable if the only information we need is the author's name. But if we want more information about an author, it will be better to have the ability to store a more complex object in an attribute - for example, a set of Person objects. Of course, each accounting system might need different kinds of objects. A system that doesn't have authors might not need person objects. A system to help a teacher keep track of grades might have AssignmentType objects. There should be some way to dynamically specify the types used in transaction fields in a specific system. Each type should correspond to a Smalltalk class (i.e., there should be a class called Person or AssignmentType), and each class should respond appropriately to the #readFrom: message.

6.5. Databases

In order for any significant number of people to use Accounts concurrently, further database interfaces should be explored. It would seem natural to connect Accounts to GemStone and Versant, two object oriented DBMSs that have Smalltalk interfaces.

Adding traditional SQL relational database access (as opposed to Unify 5.0) would make Accounts more accessible to people with SQL-based relational DBMSs. Connecting Accounts to an SQL-based relational DBMS should be relatively straightforward. It should only mean changing the C primitives that access the Unify database to use SQL calls instead of C function calls. The one possibly difficult part is making sure the C primitives can be run when they are bound to a DBMS plan.

Also, a couple of changes will be required in the **TransactionReference** class. That code was written when transaction types were implemented as generated classes. The code that uses that information must be changed. Also, there is an instance variable called listdbIDs that isn't mentioned in section 4.2.2. It was added to deal with a quirk in Unify 5.0 which made it difficult to store the transaction ID in the lists. Instead, all the tuple IDs of all the list items were stored in listdbIDs, a dictionary of list names with ordered collections of tuple IDs. For any other relational DBMS, this instance variable would not be necessary.

6.6. Additional Examples

So far, the Accounts developers have been the only users of Accounts. It would be very helpful to have some real users using Accounts to build real accounting systems. The advantage to having real world users is that they can find the shortcomings in the framework. In particular, [John94] speculates that an investment broker should be able to use Accounts to keep track of clients' investments, but also that Accounts will need additional functionality to support this. A real investment broker trying to build an accounting system will find the areas of improvement more readily.

6.7. Additional Tools

There are several tools that most accounting systems have. Adding them to the Accounts framework would increase its functionality. These tools are a spreadsheet, a check writing tool, and a report writing tool. The check writing tool could be hooked to the transaction type definition, which would have a flag indicating whether the transactions for that type should be capable of printing checks for themselves.

6.8. Accounts Graph

Currently, the Accounts interface only allows an account to be a component of one account. And once it is added to an account, it can't be moved or copied to another account. This is probably too limiting for many accounting systems. The framework code allows for the possibility that an account might be in more than one container. Theoretically, only the interface should have to change in order to allow a directed, acyclic graph of accounts rather than the account forest that the framework currently uses.

6.9. Class Reorganization

There may be some opportunity for factoring out common behavior in three places. External transactions and internal transactions both have fields, and they are both posted to accounts. It may be advantageous to create a **Transaction** class from which both external transactions and internal transactions inherit.

Also, the code for handling template accounts is part of the **Account** class hierarchy, so all accounts have this code whether they are templates or not. More properly, "template" is a role that an account plays, so there should be a separate class for defining the functionality of template accounts. All accounts would have a role and for template accounts, the role would be an instance of **Template**.

The **TransactionTree** class contains information about transactions as well as information about how a red-black tree works. It might be beneficial to separate these into two separate classes - a generic class for red-black trees, and a specialized class for the things a transaction tree needs to know. A further separation would occur when applying the Strategy pattern [Gamm94] or subclassing to separate the abstract operations of a tree from the specific strategy used for a particular tree.

6.10. Account Type Checking

Currently, there is no mechanism for specifying that all components of a composite account will have the same type. And yet, this is almost always the case. The advantage of including the ability to make this specification is to introduce account type checking into the framework.

6.11. Efficient Calculation for All Supported Operators and Time Ranges

Section 4.1.4 describes how transaction attribute values are calculated efficiently for the supported time ranges of origin-to-date, month-to-date, and year-to-date. The exception to this is that month-to-date and year-to-date attributes cannot be calculated using partial results for attributes that use the 'min:', 'max:', and 'useOldUnlessNewNotNil:' operators. This is because there is no way to get the correct result given the attribute's value at the beginning and end of the time range, at least if they are the same. For example, if one wants to find the largest sale an employee had in a given month, and the maximum value at the beginning and end of the month were both $1000.00, there would be no way of telling what the maximum value was during only that month.

One way to fix this would be to devise a way of using partial results to store values for a specific time range. This would allow a month-to-date or year-to-date value to be calculated with one pass of the height of the table rather than two. This appears to be a hard problem.

6.12. Budgeting and Reconciling

Companies will often make plans for how they plan to receive and spend their money in the future. Their plans are typically recorded as transactions posted to a separate ("shadow") set of books. Then, when the money is actually spent, it is reconciled to the plan. Accounts should be able to support this generally, given a reporting mechanism and automatic transaction generation.

However, there are two additional things that some companies will want to have as part of their budgeting process. One is the ability to associate regular transactions with specific budget transactions so that the amount spent per budget transaction can be recorded. This capability is also needed for determining cost of goods sold (i.e., determining profit by calculating how much each item sold originally cost) where a specific sales transaction item is associated with a specific purchase transaction item.

The other thing companies like to do is to specify a higher level of granularity when budgeting then when recording. For example, a company might decide it is going to buy 1000 computers., but not specify what kinds it will buy. Although 'computer' would probably be a simple account and could therefore have transactions posted to it, it would also be a template account for accounts representing specific models of computers. The recording process would store its transactions at a lower level in the account type hierarchy. So the budget reconciliation process would have to expect to get its data from several different places.

6.13. Other Time-Dependent Data

Currently, attributes' values change over time because the attribute depends on different transactions for different time ranges. However, the definition of an attribute could also change through time. An example of this would be calculation of vacation time for employees. Sometimes, vacation time is calculated according to the rules that were in effect when the employee was hired. So it is necessary to be able to find how the attribute for vacation time was defined on a particular date, and to be able to specify the source of the date in the attribute definition.

6.14. Framework Intersections

The Accounts user interface was written using two frameworks for user interfaces. HotDraw [John92] was used for the aggregate account components view (Figure 10) and for the transaction mapping interface (Figure 4). The transaction mapping interface was written using John Brant's extensions to HotDraw [Bran94].

The rest of the Accounts user interface uses ParcPlace VisualWorks [Parc92]. In contrast to HotDraw, which uses geometric graphics objects, VisualWorks uses more traditional graphical user interface objects, such as buttons, tables, and pull-down menus. (For more detail about this interface, see [Evan94].)

Both HotDraw and VisualWorks are examples of horizontal frameworks, which can be used for applications in many different domains. The domain of this framework is accounting systems. Therefore, although the interface classes primarily contain knowledge provided by their frameworks, they also contain some knowledge about accounting systems.

For example, the **TransNode** class (which is a subclass of **CachedFigure**) in the HotDraw-based aggregate account contents interface, knows that when the user draws a line from a transaction rectangle to an account rectangle, the trans node object should tell the account associated with the drawing to relate the component account for the account rectangle with the transaction definition for the transaction rectangle (see the #addTarget: method). On the other hand, if the user draws a line in the wrong direction, from an account rectangle to a transaction rectangle, the **AccountNode** class knows that it should not do anything (see the #handles method). So the interface code knows that transactions post to accounts and not vice versa.

Another example from the VisualWorks interface is that the **CacctContent** class, which handles the contents view for a composite account, knows in its #initialize: method that the contents of a composite account should be a list of its components. On the other hand, the **SacctContent** class, which handles the contents view for a simple account, knows in its #initialize: method that the contents of a simple account should be a list of its transactions. So the interface code knows that some kinds of accounts have components and that other kinds of accounts that do not have components have transactions.

One interesting direction for future research would be to examine ways to specify intersections between horizontal and vertical frameworks in ways that don't require the developer to write code. HotDraw and VisualWorks already provide this to some degree. John Brant's extensions to HotDraw [Bran94] include a tool to specify the drawing tools used for specific applications of HotDraw. VisualWorks generates **ApplicationModel** subclasses and initialization information for the interface based on information provided by the application developer (the name of the new subclass and the fields in the model).

6.15. Domain Languages

At a more general level, Accounts provides numerous opportunities for framework research. The Accounts framework is an example of a domain language - a language used to describe a particular application domain. Combining Accounts with a study of other frameworks could lead to a study of domain languages in general. Are there any general features that apply to all domain languages? Is it possible to create a meta-framework - a framework for building frameworks? As software continues to evolve, it will be written at a more general level, and the problems which now have software solutions will in the future be solved using specifications selected by a user of the system. Framework research can help speed this process.

Literature Cited

[Bran94] John Brant and Ralph E. Johnson. "Creating Tools in HotDraw by Composition." In *Proceedings of Conference on Technology of Object-Oriented Languages and Systems* : 445-454 (1994)

[Catt91] R.G.G. Cattell. *Object Data Management.* Addison-Wesley, Reading, Massachusetts, 1991.

[Corm90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[Cox91] Brad J. Cox. *Object-Oriented Programming, 2nd Edition.* Addison-Wesley, Reading, Massachusetts, 1991.

[Deut89] L. Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 Programming System. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Vol 2:* 55-71, ACM Press, 1989.

[Evan94] Rebecca Evans. *An Object Oriented Interface to an Object Oriented Framework.* M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1994.

[Gamm94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[Gold83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[John88] Ralph E. Johnson and Brian Foote. "Designing Reusable Classes." In *Journal Of Object-Oriented Programming, Vol. 1, No. 2* : 22-35 (June/July 1988)

[John91] Ralph E. Johnson and Vincent F. Russo. "Reusing Object-Oriented Designs." University of Illinois tech report *UIUCDCS 91-1696.*

[John92] Ralph E. Johnson. "Documenting Frameworks Using Patterns." In *Proceedings Of Conference on Object-Oriented Programming*, *Systems, and Applications*: 63-76 (1992)

[John94] Ralph E. Johnson, Ward Cunningham, Rebecca Evans, and Paul Keefer. "The Accounts Framework." Unpublished.

[Kim88] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. "Integrating an Object-Oriented Programming System with a Database System." In *Proceedings Of Conference on Object-Oriented Programming*, *Systems, and Applications*: 142-152 (1988)

[Meye88] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall International (UK) Ltd., Hemel Hempstead, Hertfordshire, UK, 1988.

[Parc92] ParcPlace Systems, Inc. *ParcPlace VisualWorks Release 1.0 User's Guide.* Sunnyvale, California, 1992.

[Ullm88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, Rockville, Maryland, 1988.