Domain Analysis

Chapter 12, Domain Analysis

- Chapter 12 is on the topic of domain analysis for object-oriented design
- In brief, this means identifying classes, relationships between classes, and attributes of classes
- The object-oriented approach given is very close to the modeling process for relational databases

12.1 Overview of Analysis

- Analysis starts with some sort of vague statement of the problem to be solved
- It involves taking a look at existing systems and talking to users of an old system or people who are requesting a new system (possibly the same people)
- The goal is to arrive at an unambiguous specification for the new system
- The diagram on the following overhead is supposed to summarize these ideas

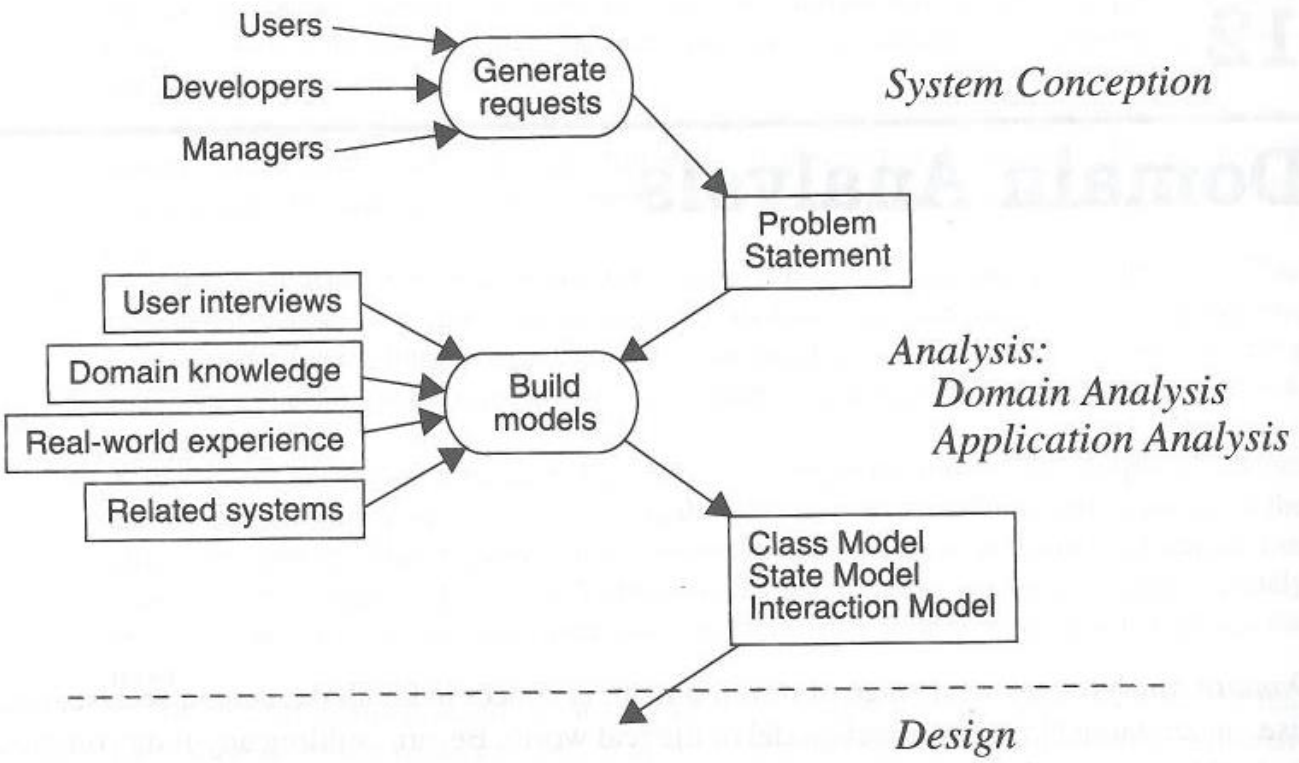

Figure 12.1 Overview of analysis. The problem statement should not be taken as immutable, but rather as a basis for refining the requirements.

12.2 Domain Class Model

- The following list summarizes the contents of the chapter, outlining the sequence of steps in analysis
- Find classes. [12.2.1-12.2.2]
- Prepare a data dictionary. [12.2.3]
- Find associations. [12.2.4-12.2.5]
- Find attributes of objects and links. [12.2.6-12.2.7]

- Organize and simplify classes using inheritance. [12.2.8]
- Verify that access paths exist for likely queries. [12.2.9]
- Iterate and refine the model. [12.2.10]
- Reconsider the level of abstraction. [12.2.11]
- Group classes into packages. [12.2.12]

Finding Classes

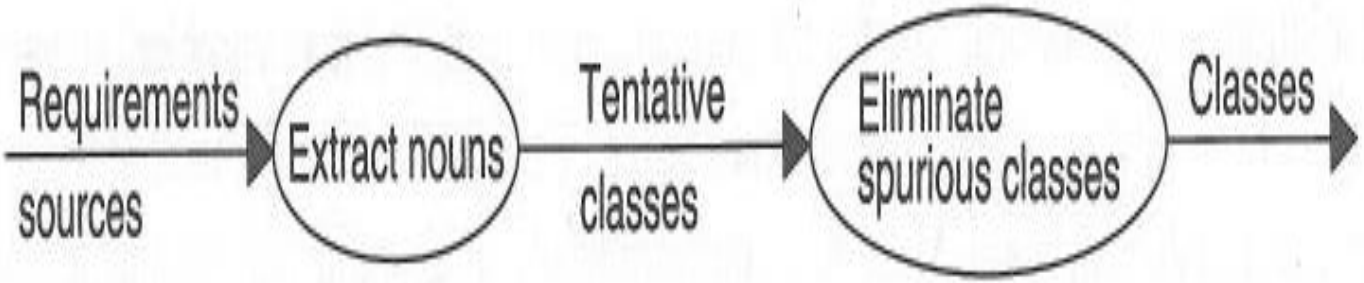

Figure 12.2 Finding classes. You can find many classes by considering nouns.

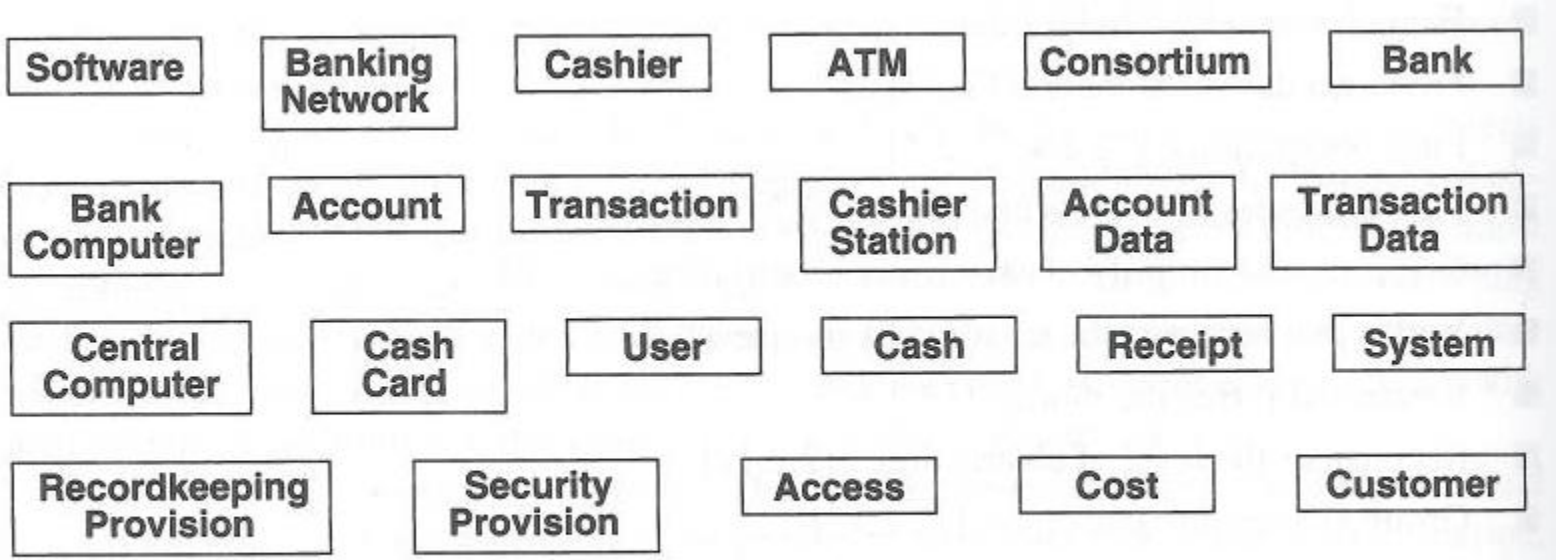

Figure 12.3 ATM classes extracted from problem statement nouns

Figure 12.4 ATM classes identified from knowledge of problem domain

12.2.2 Keeping the Right Classes

- The book has a list of the kinds of classes that can be eliminated from a design
- The descriptive names alone may give an idea of why the classes are not suitable
- In other cases further commentary is helpful

1. Redundant classes

- ATM example: Customer and User are redundant
- Customer is kept because it's more descriptive
- Customer and passenger in airline ticket booking system

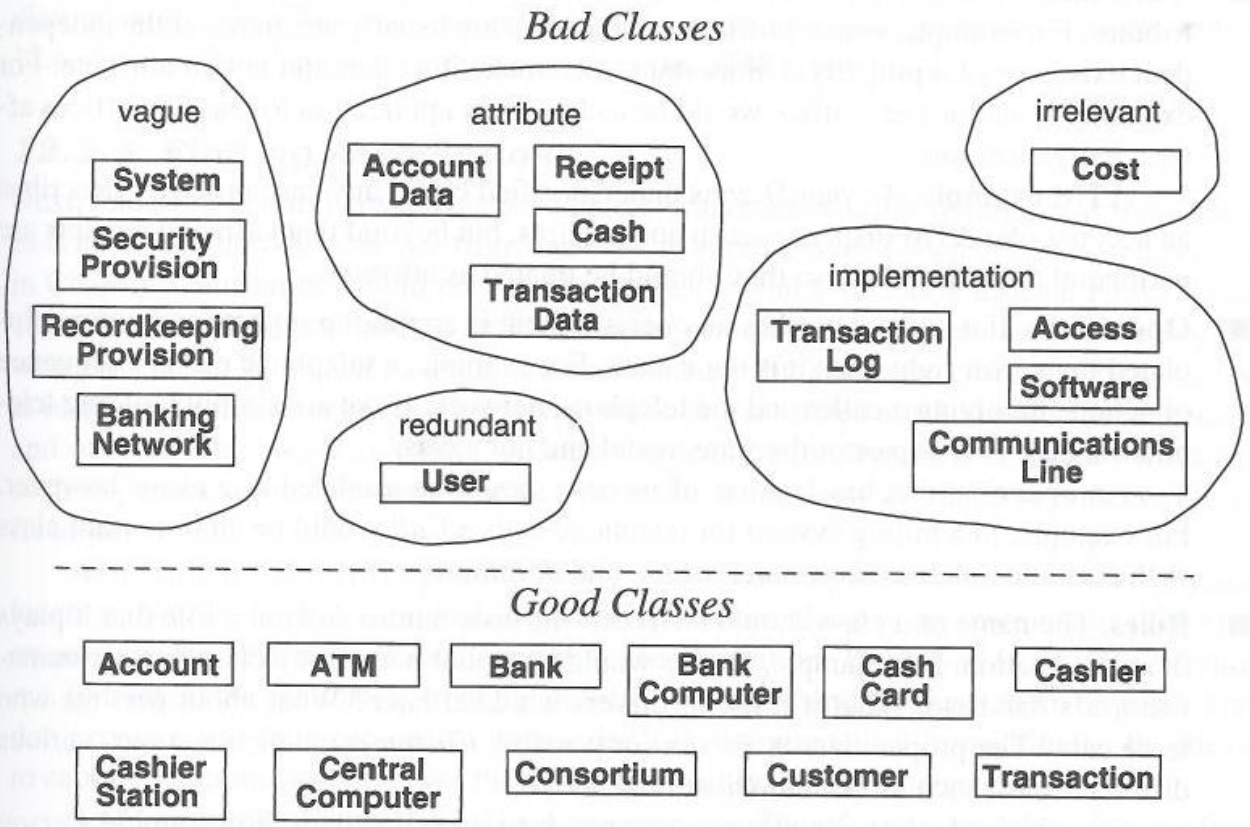

Figure 12.5 Eliminating unnecessary classes from ATM problem

2. Irrelevant classes:

- ATM example: Keeping track of Cost is beyond the scope of this application
- In Theater ticket reservation system, the occupation of ticket holder is not important, occupation of theater personnel is important.

3. Vague classes

- ATM example: Record Keeping Provision is vague
- It is most likely part of a Transaction

4. Attributes

- These are things you identify as nouns, but which are attributes of classes, not classes themselves
- ATM example: Account Data is likely an attribute of an Account

5. Operations

- Again, these are things that you identify as nouns, but which turn out to be things which would be implemented as methods within classes, not as classes

6. Roles

- Think sub-type here.
- A person may be an employee, a boss, a spouse, but a person is a person, not a role.

7. Implementation constructs

- This is a danger whenever letting computer people design applications for users…
- ATM example: TransactionLog and CommunicationsLine are instances of this

8. Derived classes

- The meaning here is not too clear.
- This does not mean eliminate subclasses specifically.
- It does mean eliminate identified classes which could just as well be a kind of some other class.

- The following overhead shows the ATM example again
- Good classes are identified
- Classes to be eliminated are classified according to which of the bad categories they fell into
- Not all of the bad categories are illustrated, just five of them

12.2.3 Preparing a Data Dictionary

- This is a deceptively simple requirement
- From the O-O perspective it means that it should be possible to write a concise description of a valid class
- The book illustrates this with the figure shown on the next overhead
- The relational requirement is more concrete
- It should be possible to make a list of all tables and all fields of those tables, identifying the meanings of the fields and what domains they are on.

Account—a single account at a bank against which transactions can be applied. Accounts may be of various types, such as checking or savings. A customer can hold more than one account.

ATM—a station that allows customers to enter their own transactions using cash cards as identification. The ATM interacts with the customer to gather transaction information, sends the transaction information to the central computer for validation and processing, and dispenses cash to the user. We assume that an ATM need not operate independently of the network.

Bank—a financial institution that holds accounts for customers and issues cash cards authorizing access to accounts over the ATM network.

BankComputer—the computer owned by a bank that interfaces with the ATM network and the bank's own cashier stations. A bank may have its own internal computers to process accounts, but we are concerned only with the one that talks to the ATM network.

CashCard—a card assigned to a bank customer that authorizes access of accounts using an ATM machine. Each card contains a bank code and a card number. The bank code uniquely identifies the bank within the consortium. The card number determines the accounts that the card can access. A card does not necessarily access all of a customer's accounts. Each cash card is owned by a single customer, but multiple copies of it may exist, so the possibility of simultaneous use of the same card from different machines must be considered.

Cashier—an employee of a bank who is authorized to enter transactions into cashier stations and accept and dispense cash and checks to customers. Transactions, cash, and checks handled by each cashier must be logged and properly accounted for.

CashierStation—a station on which cashiers enter transactions for customers. Cashiers dispense and accept cash and checks; the station prints receipts. The cashier station communicates with the bank computer to validate and process the transactions.

CentralComputer—a computer operated by the consortium that dispatches transactions between the ATMs and the bank computers. The central computer validates bank codes but does not process transactions directly.

Consortium—an organization of banks that commissions and operates the ATM network. The network handles transactions only for banks in the consortium.

Customer—the holder of one or more accounts in a bank. A customer can consist of one or more persons or corporations; the correspondence is not relevant to this problem. The same person holding an account at a different bank is considered a different customer.

Transaction—a single integral request for operations on the accounts of a single customer. We specified only that ATMs must dispense cash, but we should not preclude the possibility of printing checks or accepting cash or checks. We may also want to provide the flexibility to operate on accounts of different customers, although it is not required yet.

Figure 12.6 Data dictionary for ATM classes. Prepare a data dictionary for all modeling elements.

12.2.4 Finding Associations

- Again, at this stage, the goal is not to find “is-a” inheritance relationships
- It is to find “has-a” relationships
- In O-O terms, most often this arises when one class has an instance variable that is a reference to an instance of another class
- In a relational database, the relationship will eventually be captured by means of shared attributes (key pairs) but that lies in the future

Verb phrases

Banking network includes cashier stations and ATMs
Consortium shares ATMs
Bank provides bank computer
Bank computer maintains accounts
Bank computer processes transaction against account
Bank owns cashier station
Cashier station communicates with bank computer
Cashier enters transaction for account
ATMs communicate with central computer about transaction
Central computer clears transaction with bank
ATM accepts cash card
ATM interacts with user
ATM dispenses cash
ATM prints receipts
System handles concurrent access
Banks provide software
Cost apportioned to banks

Implicit verb phrases

Consortium consists of banks
Bank holds account
Consortium owns central computer
System provides recordkeeping
System provides security
Customers have cash cards

Knowledge of problem domain

Cash card accesses accounts
Bank employs cashiers

Figure 12.7 Associations from ATM problem statement

12.2.5 Keeping the Right Associations

- Just like with classes, the first step at a design might tentatively identify associations that do not belong in the final design
- Again, the book gives a list of the kinds of associations which should be thrown out:

1. Associations between eliminated classes

- I guess it never hurts to state the obvious.
- ATM example: There are many instances of this due to the number of eliminated classes

2. Irrelevant or implementation associations

- Clearly, irrelevant associations don't belong in a model
- Likewise, when modeling, implementation concerns don't belong
- ATM example: System handles concurrent access
- With multiple ATM's, concurrency will occur
- Reality is concurrent—but this is an implementation problem, not a design problem

3. Actions

- Associations should embody structural relationships, not transient processing events
- ATM example: ATM accepts cash card, ATM interacts with user, Central computer clears transaction with bank, and Central computer communicates with bank all describe transient actions, not structural relationships

4. Ternary associations

- This terminology is linked to database concerns
- It is possible to have a three-way relationship among three different classes.
- What they want to eliminate are cases where there are two base classes and the third item in the association description is just an attribute that is connected with the relationship
- This idea will come up again when considering attributes

- ATM example:
- Bank computer processes transaction against account breaks down into Bank computer processes transaction and Transaction concerns account
- Cashier enters transaction for account and ATMs communicate with central computer about transaction can also be broken into two binary associations

5. Derived associations

- This is largely a warning against redundancy in the design.
- If there is a relationship between classes and this ‘implies’ another relationship—possibly with a different name, but not differing in how things are related, then there is no need to capture the second relationship in the design.
- The first relationship captures it all.

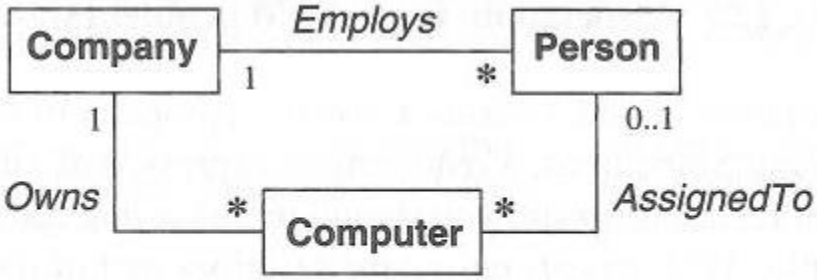

Figure 12.8 Nonredundant associations. Not all associations that form multiple paths between classes indicate redundancy.

- A company owns a set of computers
- It also employs a set of employees
- Employees are assigned to computers (or vice-versa)
- The point is that AssignedTo is not a derived relationship
- Which computer an employee is assigned to cannot be derived from the fact that an employee works for a given company

1. Misnamed associations

- You eliminate them if they are redundant.
- You rename them if they're valid and not redundant.
- Names should state what a relationship is, not be based on how it came about or some other extraneous description.
- ATM example: Bank computer maintains accounts describes an action
- Rename this Bank holds accounts

2. Association end names

- Like in an E-R model, remember to label any ambiguous links

3. Qualified associations

- This is a new aspect of UML notation
- A qualifier distinguishes objects on the many side of an association
- ATM example: The qualifier bankCode distinguishes the different banks in a consortium
- This will be further explained just before the next UML diagram is shown

4. Multiplicity

- When doing a first design, you can try to figure out the cardinalities of the ends of links, but you can always straighten out the details later.

5. Missing associations

- Again, stating the obvious never hurts.
- If you forgot something, add it to the design.
- ATM example: The book suggests that all of these associations were forgotten in the foregoing list: Transaction entered on cashier station, Customers have accounts, Transaction authorized by cash card, and possibly Cashier authorized on cashier station

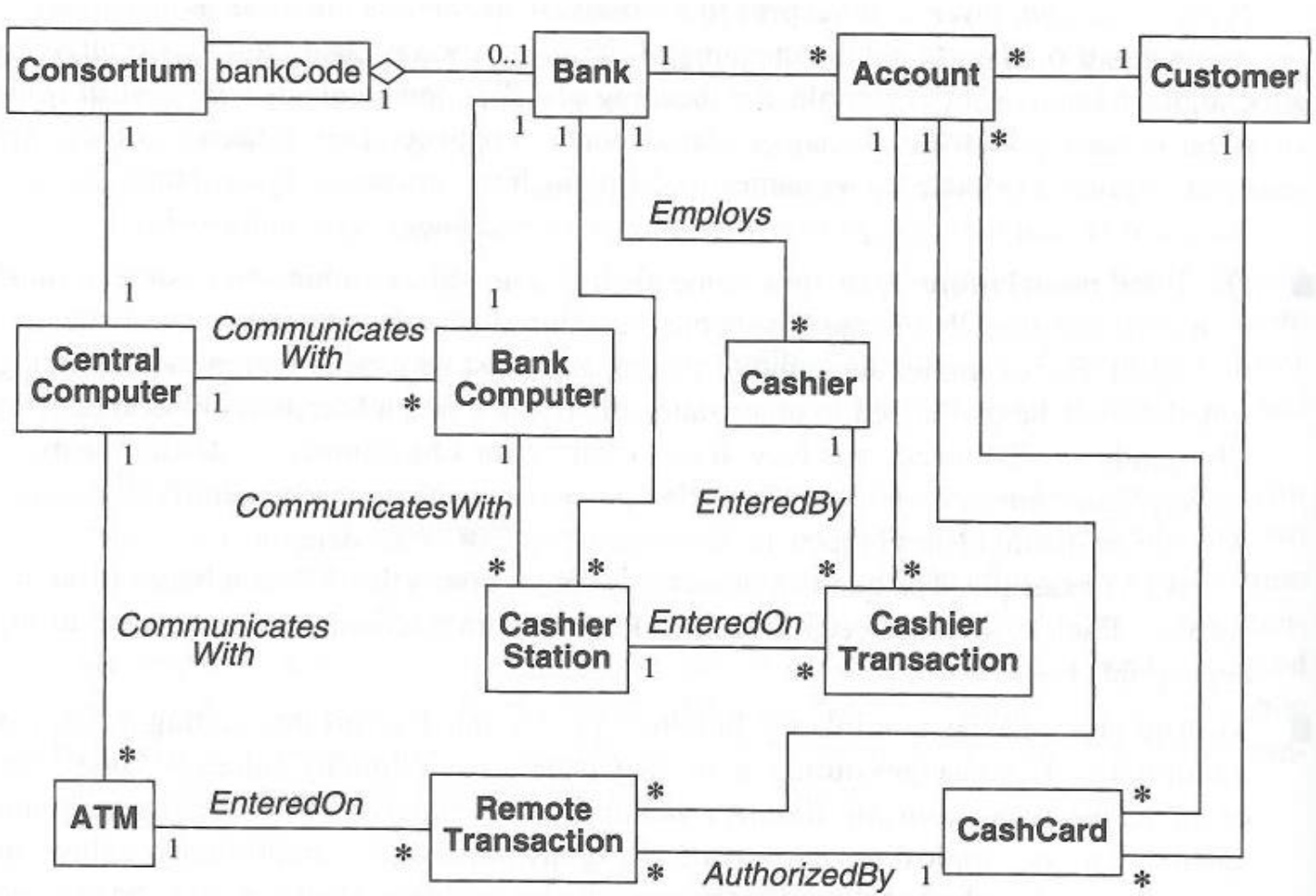

Figure 12.9 Initial class diagram for ATM system

12.2.6 Finding Attributes

- The book suggests that attributes can be identified as nouns X, which appear in descriptions like these:
- The X of Y
- Y's X
- Just like with relational modeling, meaningful names are useful
- Also, any attribute that can be calculated from other attributes is not a base attribute

12.2.7 Keeping the Right Attributes

- Just like with classes and associations, the first step at a design might tentatively identify attributes that do not belong in the final design
- Again, the book gives a list of the kinds of attributes which should be thrown out:
- 1. Objects
- If something is an object it's not a simple attribute
- However, note that instance variables can be object references

- 2. Qualifiers
- This item illustrates an important difference between O-O modeling and relational modeling
- What does the book mean by qualifier?
- It gives an example:
- Suppose you have a design with both Company and Person classes
- Suppose you also identify an attribute with a qualified name like employeeNumber

- 3. Name
- This is also kind of obscure, but tied to relational modeling concerns
- If a name attribute has to be unique, then the book suggests that in an O-O model it is likely not an attribute
- It is serving the moral purpose of a pk and its implementation is likely to end up being as part of a reference

- 4. Identifier
- This is also at least in part related to relational design ideas
- An identifier in an O-O setting might be a hash code for example
- This is along the lines of a hidden record identifier in a table
- This implementation level unique identifier does not belong to the set of domain level attributes of an object in an O-O design

- 5. Attributes on associations
- This is another interesting point that will resonate with relational questions
- The book gives an example:
- membershipDate as an attribute on a many-to-many relationship

- 6. Internal values
- If the attribute is not visible outside of the class, then it's not necessary to include it in the modeling.
- It's an implementation issue
- 7. Fine detail
- Fine detail can be ignored when modeling.
- If necessary more detail can be included in a future iteration of the model

- 8. Boolean attributes
- The book suggests that quite often more than two values become evident and a boolean can be modeled as an enumeration (a set)
- Note also that in the end this is really an implementation issue
- You don’t initially concern yourself with the types of attributes

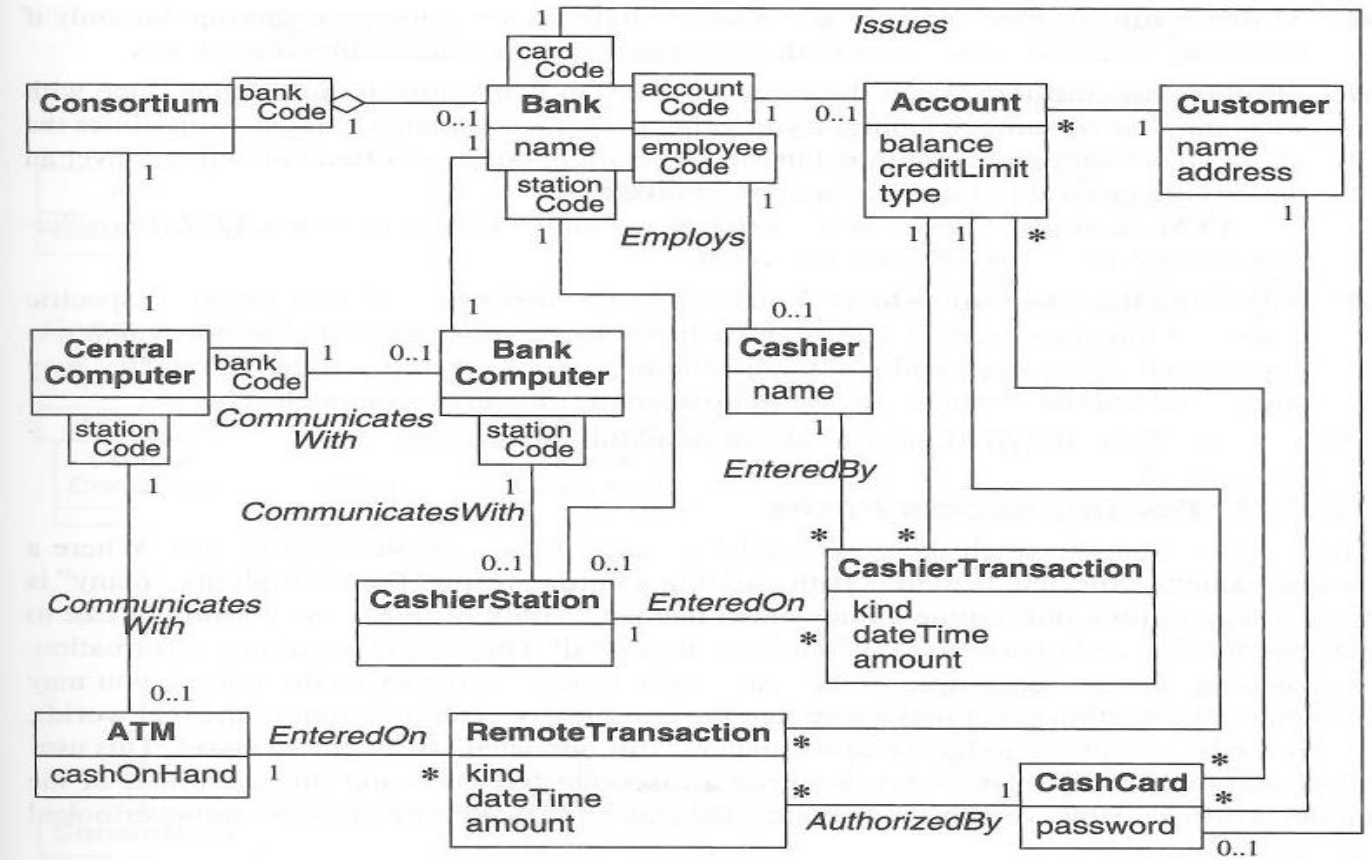

Figure 12.10 ATM class model with attributes

12.2.8 Refining with Inheritance

- 1. Bottom-up generalization
- Find classes with similar attributes, operations, and associations
- Implement common features in a super class.
- 2.Top-down generalization
- We are analyzing from application domain.
- Look for noun pharses in application

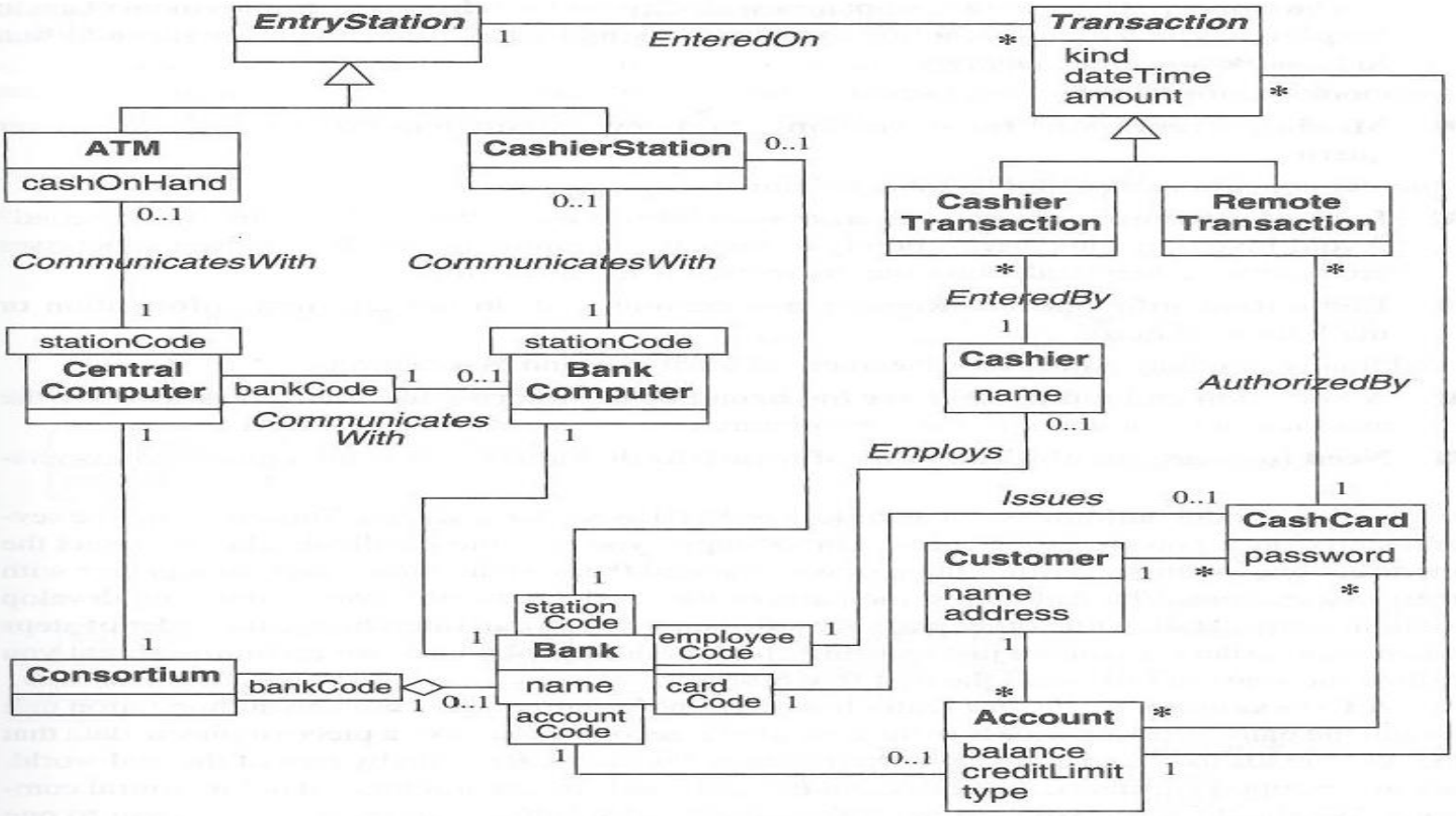

Figure 12.11 ATM class model with attributes and inheritance

12.2.9 Testing Access Paths

- What this basically means is follow the associations shown in the model and test them
- If you believe that ultimately there is a relationship between class X and class Y, it should be possible to get from class X to class Y by following links in the diagram
- This is related to relational design in the following sense:
- The book expresses the idea as following associations in order to answer queries
- In other words, the associations should exist so that you can find out what's related to what

- As you follow the links, it may be important to pay attention to their cardinalities and ask in particular whether it will be possible to uniquely identify which one of many is under consideration when following a particular path
- Once again, this question should ring a bell for someone interested in relational design

12.2.11 Shifting the Level of Abstraction

- The background to this section is the idea that every noun in the problem domain may lead to a class in the design
- In an improved design, this may no longer be true
- The book gives an example which is essentially the same as an example used to illustrate the concept in the database modeling notes given earlier

- You may have employees and bosses
- Ultimately, they are both types of person, or you might say, roles that people play
- The book illustrates this with a three layer management hierarchy
- This can be replaced with a class in a relationship with itself, as shown on the following overhead

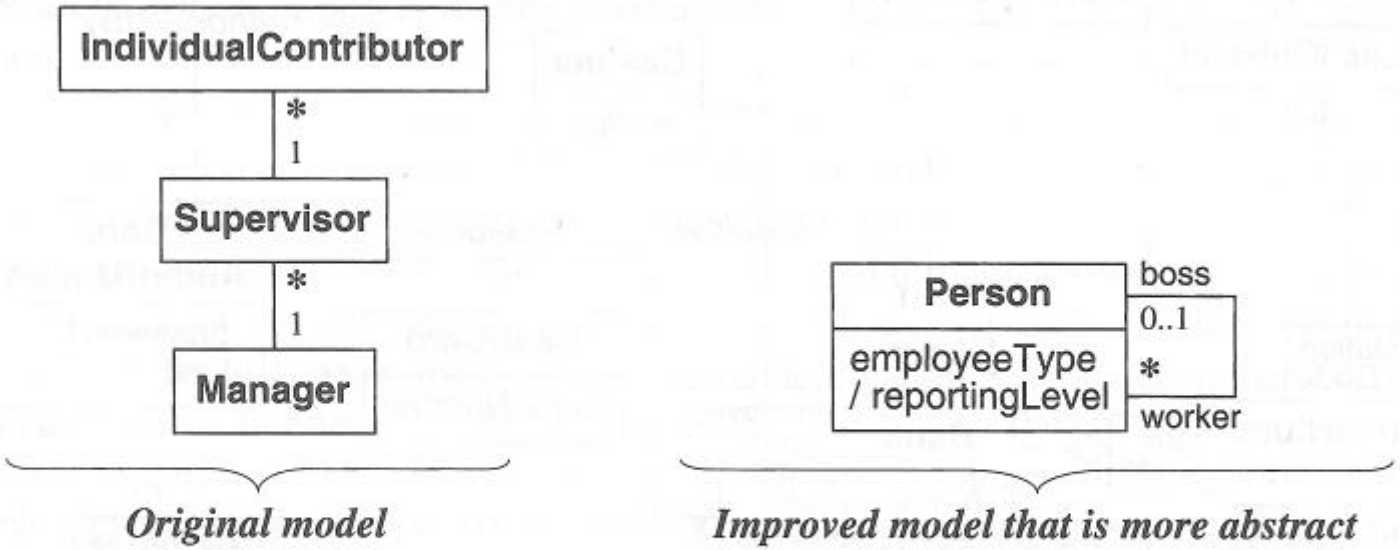

Figure 12.13 Shifting the level of abstraction. Abstraction makes a model more complex but can increase flexibility and reduce the number of classes.

12.2.12 Grouping Classes into Packages

- This is somewhat of a dark art, and even the ATM example is still too small to illustrate this realistically
- The book gives two interesting ideas that are helpful in finally understanding how to go about this
- 1. In a sense, the motivation stems from UML diagrams
- The goal is to divide classes into packages in a way that allows for clean UML diagrams

- 2. The separation of classes into packages also has semantic meaning
- In other words, classes that are put together in one package should be more closely related than classes in different packages
- The UML motivation and the semantic motivation are connected
- If classes are semantically grouped, then the UML diagrams for the packages should be self-contained and consistent

- Such a cut point class can be in both of the packages that it connects
- Then when diagramming the model, subdivided into packages, there will be no lines between the packages
- Their connection is embodied by the presence of the common class in each of them

- As noted, the ATM model is still too small to be broken down into packages
- However, the book suggests in outline form groupings that could become packages:
- 1. A teller package, including: cashier, entry station, cashier station, ATM

- 2. An account package, including: account, cash card, card authorization, customer, transaction, update, cashier transaction, remote transaction
- 3. A bank package, including: consortium, bank

12.3 Domain State Model

- Most objects in a model probably do not have states, or states worthy of modeling
- Their use is concisely given by listing their attributes and operations
- However, some objects may pass through different states which are significant to the modeling of the system
- In these cases it's useful to develop a domain state model and diagram it

- These are the steps in constructing a domain state model:
- 1. Identify domain classes with states [12.3.1]
- 2. Find states [12.3.2]
- 3. Find events [12.3.3]
- 4. Build state diagrams [12.3.4]
- 5. Evaluate state diagrams [12.3.5]

12.3.1 Identifying Classes with States

- Search for classes where objects of that class have an identifiable life cycle or history
- This can be truly cyclic, where an object returns to an initial state
- It can also be progressive, running from one state to the next until the end
- The ATM Account class is the only one with significant state
- Its life cycle is a mixture of progressive and cyclic changes of state

12.3.2 Finding States

- States, fundamentally, are captured by the values attributes have and the associations with particular other objects that a given object is in
- The idea of a state might arise from the problem domain, but it will be reflected in values associated with an instance
- You might find a particular configuration of values and also decide that this represents a significant state in the model

- States should be given a descriptive name that describes what they are, not how they came about
- Although states are based on values, differences between states should be describable as differences in quality, not just quantity of some attribute, for example
- The behavior of an object should be related to the state that it’s in

- Not all states may be apparent before a model is complete
- However, the idea that something has state should be identifiable earlier on
- In the ATM model, the Account class has state.
- Here are examples of particular states: Normal, Closed, Overdrawn, Suspended

12.3.3 Finding Events

- The events of interest are events that cause the states of objects to change
- An easy analytical approach is to ask:
- What causes a state to be initiated or entered?
- What causes a state to end or be left?

- Keep in mind that you're modeling now, not coding
- You are not interested in GUI events
- You're interested in domain events
- Here are examples of events in the ATM model: Close account, withdraw excess funds, repeated incorrect PIN, suspected fraud, and administrative action

12.3.4 Building State Diagrams

- In the a diagram a state is an oval and an event/transition is an arrow
- Analytically, for each state, you want to consider what events are possible
- For each state/event pair, you need to determine what other state, if any, this leads to

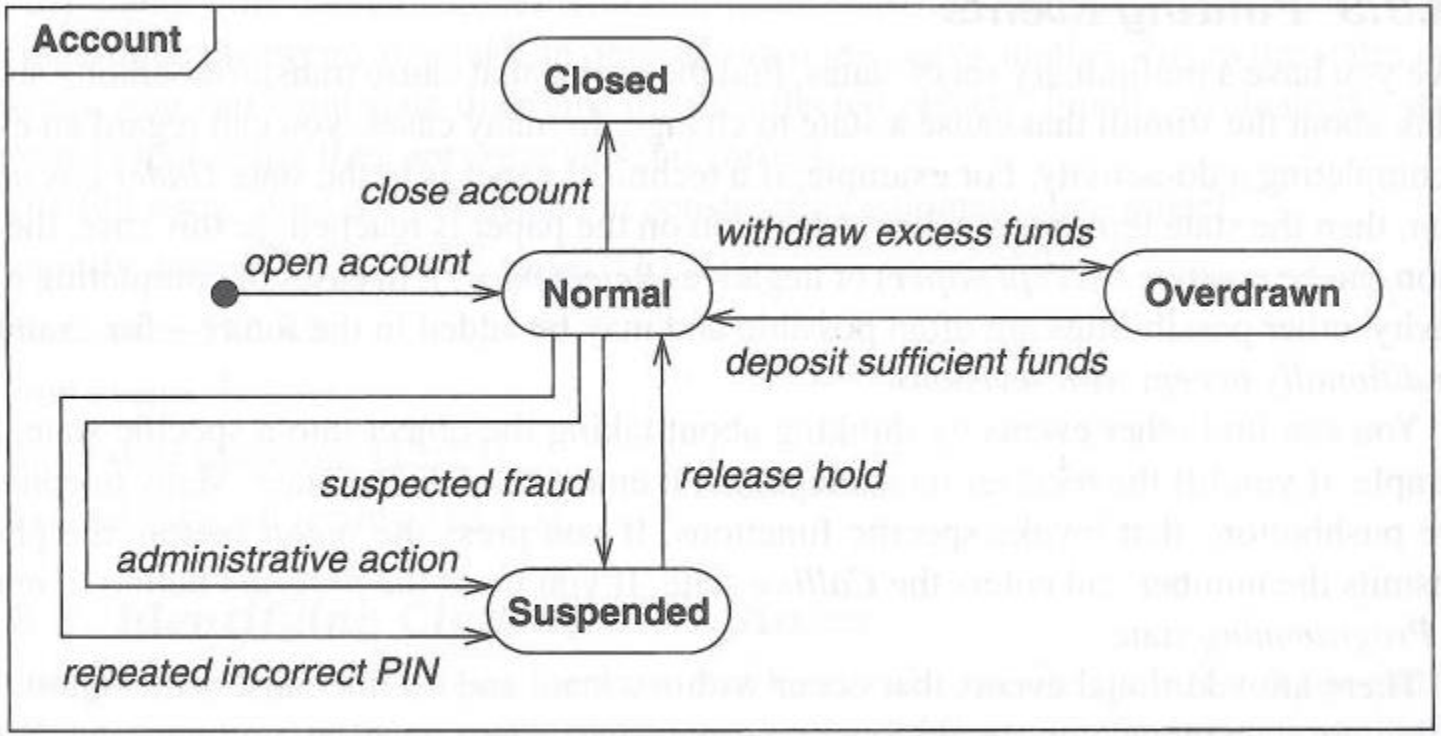

Figure 12.14 Domain state model. The domain state model documents important classes that change state in the real world.

12.4 Domain Interaction Model

- This term refers to an analysis of user interactions with the system
- It is not pursued here.

12.5 Iterating the Analysis

- This is just a statement of the obvious
- You typically have to go through the analysis process more than once
- The first pass will give a first approximation
- Following passes will refine the model
- This is not just the result of human failings
- It stems from the fact that parts of the design depend on each other
- You can’t understand one until you understand the other—and vice-versa

12.5.1 Refining the Analysis Model

- A large model will consist of multiple parts
- When refining, you can work from the top down, using the divide and conquer approach
- This works all the way down to individual classes and attributes

- Signs of problems include:
- 1. Lots of classes that are similar, but not quite the same
- This is a sign of not having generalized correctly
- In particular, you may have tried to generalize, but you should have generalized on an alternative set of shared attributes

- 2. You have combined more than one concept into a class
- This can happened because a physical entity has more than one logical aspect
- You have tried to model the physical entity as one class
- Instead, the different logical aspects should be modeled as separate classes

- 3. If your model seems to embody many exceptions or special cases, that may be a sign that something was missed in analysis
- 4. If you expected certain things to appear symmetrically in the model, and they don't, this may mean that you overlooked something (or had false expectations…)

- 5. A model may be inappropriate if it too rigidly captures details of current business practice
- A good model will be abstract enough that will allow for changes in specific business practices in the future
- It is difficult to quantify flexibility and how much flexibility is needed, but it is still an important analysis concept

12.5.2 Restating the Requirements

- The model resulting from the analysis process may not agree exactly with the requirements that were given at the beginning of the process
- Revised requirements should be explicitly written down, and they should be verified with the users who originally requested the system

12.5.3 Analysis and Design

- Dividing analysis and design into two steps is a convenient application of divide and conquer
- The idea is that analysis proceeds without any preconceived ideas about how the system might be implemented
- The reality is that analysis is never done in a complete vacuum