12.2.1 *Finding Classes*

The first step in constructing a class model is to find relevant classes for objects from the application domain. Objects include physical entities, such as houses, persons, and machines, as well as concepts, such as trajectories, seating assignments, and payment schedules. All classes must make sense in the application domain; avoid computer implementation constructs, such as linked lists and subroutines. Not all classes are explicit in the problem statement; some are implicit in the application domain or general knowledge.

11.3.1 *The ATM Case Study*

Figure 11.3 shows a problem statement for an automated teller machine (ATM) network.

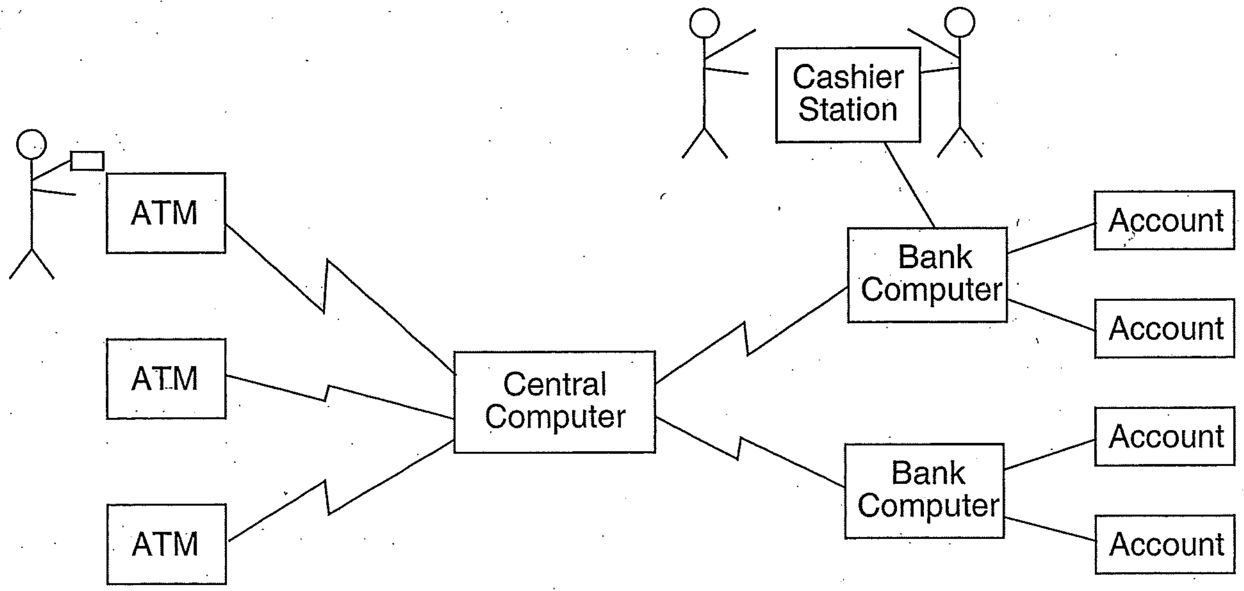

Figure 11.3 ATM network. The ATM case study threads throughout the remainder of this book.

Design the software to support a computerized banking network including both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data.

Automatic teller machines communicate with a central computer that clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate recordkeeping and security provisions. The system must handle concurrent accesses to the same account correctly.

The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

Utdrag ur:
Object-Oriented Modeling and Design with UML
av M Blaha & J Rumbaugh

As Figure 12.2 shows, begin by listing candidate classes found in the written description of the problem. Don't be too selective; write down every class that comes to mind. Classes often correspond to nouns. For example, in the statement "a reservation system to sell tickets to performances at various theaters" tentative classes would be *Reservation, System, Ticket, Performance,* and *Theater.* Don't operate blindly, however. The idea to is capture concepts; not all nouns are concepts, and concepts are also expressed in other parts of speech.

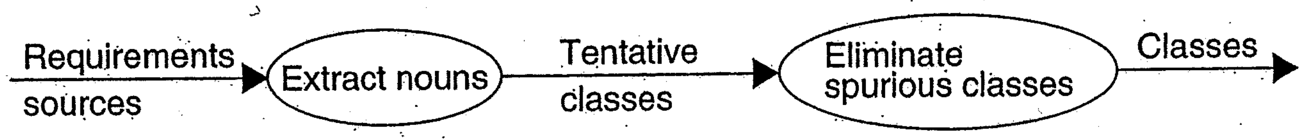

Figure 12.2 Finding classes. You can find many classes by considering nouns.

Don't worry much about inheritance or high-level classes; first get specific classes right so that you don't subconsciously suppress detail in an attempt to fit a preconceived structure. For example, if you are building a cataloging and checkout system for a library, identify different kinds of materials, such as books, magazines, newspapers, records, videos, and so on. You can organize them into broad categories later, by looking for similarities and differences.

ATM example. Examination of the concepts in the ATM problem statement from Chapter 11 yields the tentative classes shown in Figure 12.3. Figure 12.4 shows additional classes that do not appear directly in the statement but can be identified from our knowledge of the problem domain.

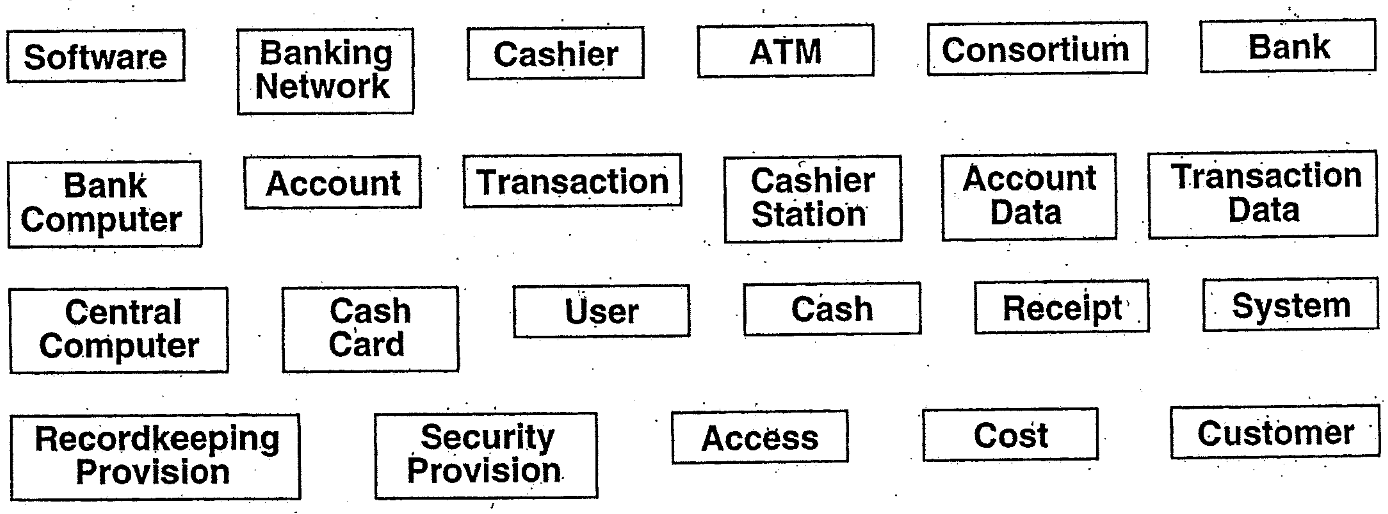

Figure 12.3 ATM classes extracted from problem statement nouns

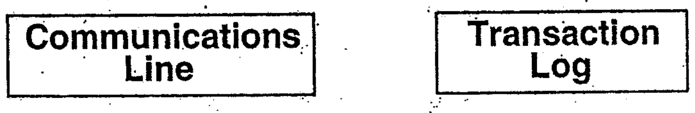

Figure 12.4 ATM classes identified from knowledge of problem domain

12.2.2 Keeping the Right Classes

Now discard unnecessary and incorrect classes according to the following criteria. Figure 12.5 shows the classes eliminated from the ATM example.

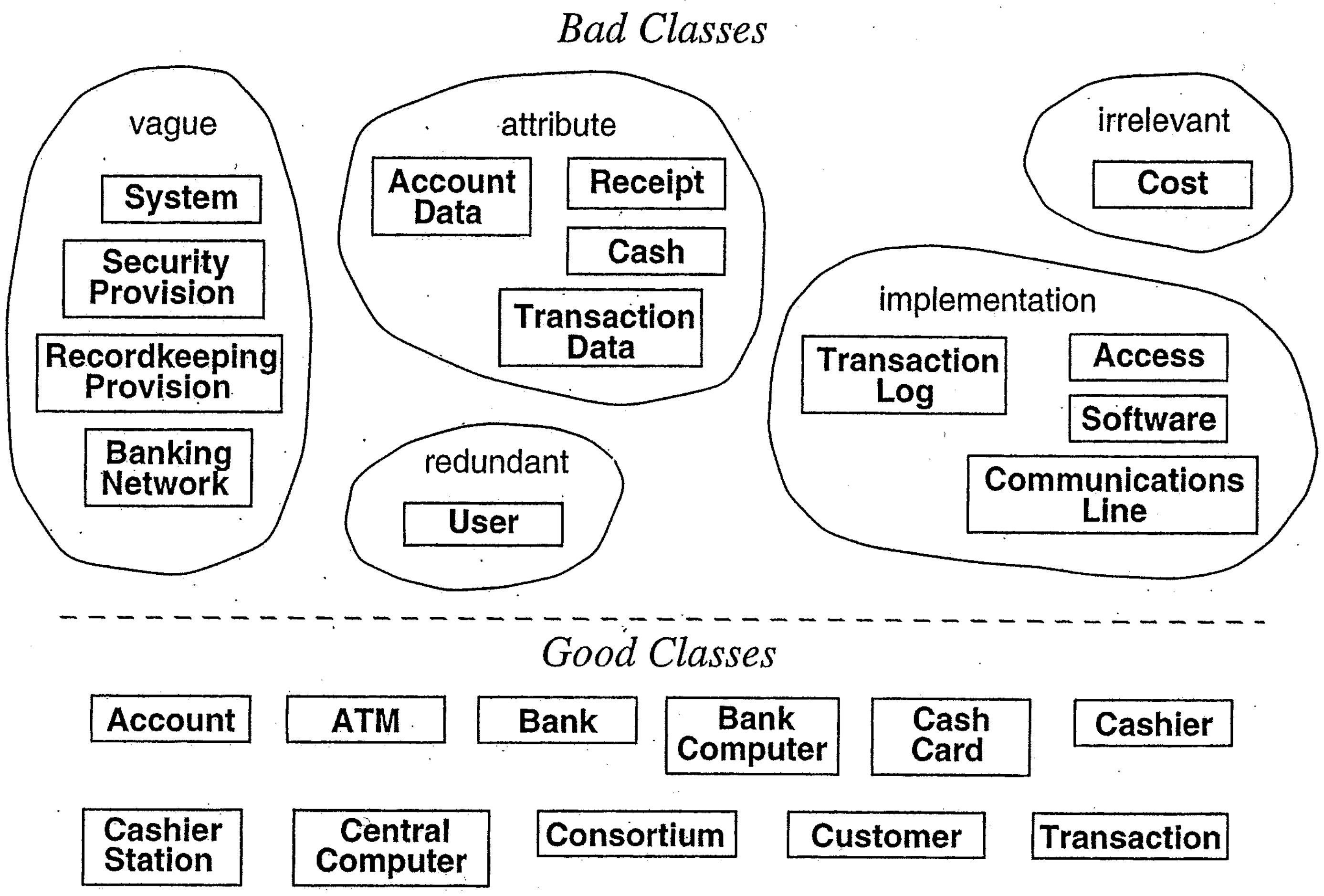

Figure 12.5 Eliminating unnecessary classes from ATM problem

- **Redundant classes.** If two classes express the same concept, you should keep the most descriptive name. For example, although *Customer* might describe a person taking an airline flight, *Passenger* is more descriptive. On the other hand, if the problem concerns contracts for a charter airline, *Customer* is also an appropriate word, since a contract might involve several passengers.

  ATM example. *Customer* and *User* are redundant; we retain *Customer* because it is more descriptive.
- **Irrelevant classes.** If a class has little or nothing to do with the problem, eliminate it. This involves judgment, because in another context the class could be important. For example, in a theater ticket reservation system, the occupations of the ticket holders are irrelevant, but the occupations of the theater personnel may be important.

  ATM example. Apportioning *Cost* is outside the scope of the ATM software.
- **Vague classes.** A class should be specific. Some tentative classes may have ill-defined boundaries or be too broad in scope.

ATM example. *RecordkeepingProvision* is vague and is handled by *Transaction.* In other applications, this might be included in other classes, such as *StockSales, TelephoneCalls,* or *MachineFailures*.

- **Attributes**. Names that primarily describe individual objects should be restated as attributes. For example, *name*, *birthdate*, and *weight* are usually attributes. If the independent existence of a property is important, then make it a class and not an attribute. For example, an employee's office would be a class in an application to reassign offices after a reorganization.

  ATM example. *AccountData* is underspecified but in any case probably describes an account. An ATM dispenses cash and receipts, but beyond that cash and receipts are peripheral to the problem, so they should be treated as attributes.

- **Operations**. If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class. For example, a telephone call is a sequence of actions involving a caller and the telephone network. If we are simply building telephones, then *Call* is part of the state model and not a class.

  An operation that has features of its own should be modeled as a class, however. For example, in a billing system for telephone calls a *Call* would be an important class with attributes such as *date*, *time*, *origin*, and *destination*.

- **Roles**. The name of a class should reflect its intrinsic nature and not a role that it plays in an association. For example, *Owner* would be a poor name for a class in a car manufacturer's database. What if a list of drivers is added later? What about persons who lease cars? The proper class is *Person* (or possibly *Customer*), which assumes various different roles, such as *owner, driver,* and *lessee*.

  One physical entity sometimes corresponds to several classes. For example, *Person* and *Employee* may be distinct classes in some circumstances and redundant in others. From the viewpoint of a company database of employees, the two may be identical. In a government tax database, a person may hold more than one job, so it is important to distinguish *Person* from *Employee;* each person can correspond to zero or more instances of employee information.

- **Implementation constructs**. Eliminate constructs from the analysis model that are extraneous to the real world. You may need them later during design, but not now. For example, CPU, subroutine, process, algorithm, and interrupt are implementation constructs for most applications, although they are legitimate classes for an operating system. Data structures, such as linked lists, trees, arrays, and tables, are almost always implementation constructs.

  ATM example. Some tentative classes are really implementation constructs. *TransactionLog* is simply the set of transactions; its exact representation is a design issue. Communication links can be shown as associations; *CommunicationsLine* is simply the physical implementation of such a link.

- **Derived classes**. As a general rule, omit classes that can be derived from other classes. If a derived class is especially important, you can include it, but do so only sparingly. Mark all derived classes with a preceding slash ('/') in the class name.