**Data Modelling:**

1. Over time, the DFD (data flow diagramming or process modeling) team continued to struggle with basic problem domain understanding. In contrast, the Data Base Team gained a strong, in-depth understanding.
2. Frequently, problems with data quality can be traced to a lack of consistency in (a) defining and interpreting data, and (b) implementing mechanisms to enforce the definitions. In our insurance example, is Birth Date in U.S. or European date format (mm/dd/yyyy or dd/mm/yyyy)?  
   Inconsistent assumptions here by people involved in data capture and retrieval could render a large proportion of the data unreliable.
3. More broadly, we could define integrity constraints on Birth Date. For example, it must be a date in a certain format and within a particular range.

**What makes a Good Data model?**

1. Completeness

Does the model support all the necessary data?

1. Non redundancy

Recording the same data more than once increases the amount of space needed to store the database, requires extra processes (and processing) to keep the various copies in step, and leads to  
consistency problems if the copies get out of step.

1. Enforcing business rules

If this rule correctly reflects the business requirement, the resulting database  
will be a powerful tool in enforcing correct practice, and in maintaining data quality as discussed in Section 1.5.3. On the other hand, any misrepresentation of business rules in the model may be very difficult to correct later (or to code around)

1. Data Reusability

If data has been organized with one particular application in mind, it is often difficult to use for other purposes. There are few greater frustrations for system users than to have paid for the capture and storage of data, only to be told that it cannot be made available to suit a new information  
requirement without extensive and costly reorganization

1. Stability and Flexibility

A data model is stable in the face of a change to requirements if we do not need to modify it at all. We can sensibly talk of models being more or less stable, depending on the level of change required. A data model is flexible if it can be readily extended to accommodate likely new requirements with only minimal impact on the existing structure.

1. Elegance

Elegance can be a difficult concept to pin down. But elegant models are typically simple, consistent, and easily described and summarized.

Our overall goal is to develop a model that provides the best balance among these possibly conflicting objectives. As in other design disciplines, achieving this is a process of proposal and evaluation, rather than a stepby-step progression to the ideal solution. We may not realize that a better solution or trade-off is possible until we see it.

1. The usual (and recommended) procedure is to develop the data model without considering performance, then to attempt to implement it with the available hardware and software. Only if it is not possible to achieve adequate performance in this way do we consider modifying the model itself.  
   In effect, performance requirements are usually “added to the mix” at a later stage than the other criteria, and then only when necessary.

**Conceptual, Logical, and Physical Data Models:**

Diagram

Description automatically generated

The **conceptual data model** is a (relatively)5 technology independent specification of the data to be held in the database. It is the focus of communication between the data modeler and business stakeholders, and it is usually presented as a diagram with supporting documentation.

The **logical data model** is a translation of the conceptual model into structures that can be implemented using a **database management system (DBMS).** Today, that usually means that this model specifies tables and columns, as we saw in our first example. These are the basic building blocks of relational databases, which are implemented using a **relational database management system (RDBMS).**

The **physical data model** incorporates any changes necessary to achieve adequate performance and is also presented in terms of  
tables and columns, together with a specification of physical storage (which may include data distribution) and access mechanisms.

**Normalization:**

Our principal tool is **normalization**, a set of rules for allocating data to tables in such a way as to eliminate certain types of redundancy and incompleteness.

Process:

1. One Fact per Column
2. Any Hidden Data
3. Derivable Data

Remember our basic objective of nonredundancy. We should remove any data that can be derived from other data in the table and amend the columns accordingly.

Table

Description automatically generated

Most often, however, it is added with the intention of improving performance. Even from that perspective, we should realize  
that there will be a trade-off between data retrieval (faster if we do not have to assemble the base data and calculate the total each time) and data update (the total will need to be recalculated if we change the base data).

1. Determining the primary key

the primary key is a minimal set of columns that contains a different combination of values for each row of the table.

**Repeating Groups and First Normal Form:**

To summarize: We have a set of columns repeated a number of times—a “repeating group”—resulting in inflexibility, complexity, and poor data reusability. The table design hides the problem by using numerical suffixes  
to give each column a different name.

Operation Model using Relational notation:

Text, letter

Description automatically generated

It is better to face the problem squarely and document our initial structure as in Figure 2.10. The braces (curly brackets) indicate a repeating group with an indefinite number of occurrences. This notation is a useful convention, but it describes something we cannot implement directly with a simple table. In technical terms, our data is *unnormalized.*

Text

Description automatically generated

At this point we should also check whether there are any repeating groups that have not been marked as such. To do this, we need to ask whether there are any data items that could have multiple values for a given value of the key. For example, we need to ask question whether an operation can take more than one surgeon. If yes, than {surgeon Number and Surgeon speciality} becomes another repeating group.

The procedure is to split the original table into multiple tables (one for the basic data and one for each repeating group) as follows:

1. Remove each separate set of repeating group columns to a new table  
(one new table for each set) so that each occurrence of the group  
becomes a row in its new table.  
2. Include the key of the original table in each new table, to serve as a  
cross-reference (we call this a **foreign key**).  
3. If the sequence of occurrences within a repeating group has business significance, introduce a “Sequence” column to the corresponding new table.  
4. Name each new table.  
5. Identify and underline the primary key of each new table, as discussed in the next subsection.

**First Normal Form Definition:**

■ All data of the same kind is now held in the same place. For example, all drug names are now in a common column. This translates into elegance and simplicity in both data structure and programming (we could now sort the data by drug name, for example). One fact per column.

■ The number of different drug dosages that can be recorded for an operation is limited only by the maximum possible number of rows in the **Drug Administration** table (effectively unlimited). Conversely, an operation that does not use any drugs will not require any rows in the **Drug Administration** table.

**First normal form** (**1NF**) is a property of a [relation](https://en.wikipedia.org/wiki/Relation_(database)) in a [relational database](https://en.wikipedia.org/wiki/Relational_database). A relation is in first normal form if and only if the [domain](https://en.wikipedia.org/wiki/Data_domain) of each [attribute](https://en.wikipedia.org/wiki/Column_(database)) contains only [atomic](https://en.wikipedia.org/wiki/First_normal_form#Atomicity) (indivisible) values, and the value of each attribute contains only a single value from that domain.[[1]](https://en.wikipedia.org/wiki/First_normal_form#cite_note-1)

**Problems with Tables in First Normal Form:**

Eliminating Redundancy:

Determinants:

whole procedure of separating hospital data relied on the fact that for a given hospital number there could be  
only one hospital name, contact person, hospital type, and teaching status. In fact we could look at the dependency of hospital data on hospital number as the cause of the problem. Every time a particular hospital number appeared in the **Operation** table, the hospital name, contact person, hospital type, and teaching status were the same. Why hold them  
more than once?

Formally, we say that Hospital Number is a determinant of the other four columns. We can show this as:  
Hospital Number  Hospital Name, Contact Person, Hospital Type, Teaching Status where we read “” as “determines” or “is a determinant of.”

Determinants need not consist of only one column; they can be a combination of two or more columns, in which case we can use a + sign to indicate such a combination.

For example: Hospital Number + Operation Number Surgeon Number.

**This leads us to a more formal description of the procedure:**

1. Identify any determinants, other than the primary key, and the columns they determine (we qualify this rule slightly in Section 2.7.3).  
2. Establish a separate table for each determinant and the columns it determines. The determinant becomes the key of the new table.  
3. Name the new tables.  
4. Remove the determined columns from the original table. Leave the determinants to provide links between tables.  
Of course, it is easy to *say* “Identify any determinants.” A useful starting point is to:  
  
1. Look for columns that appear by their names to be identifiers (“code,” “number”, “ID”, and sometimes “Name” being obvious candidates). These may be determinants or components of determinants.  
2. Look for columns that appear to describe something other than what the table is about (in our example, hospitals rather than operations). Then look for other columns that identify this “something” (Hospital Number in  
this case).

Our “other than the key” exception in step 1 of the procedure is interesting. The problems with determinants arise when the same value appears in more than one row of the table. Because hospital number 17 could appear in more than one row of the **Operation** table, the corresponding values of Contact Person and other columns that it determined were also held  
in more than one row—hence, the redundancy. But each value of the key itself can appear only once, by definition.

Note that if the business rules were different, some determinants might well be different. For example, consider the rule “We use a standardized cost.” If this did not apply, the determinant of Dose Cost would include Hospital Number as well as the other data items identified.

Finding determinants may look like a technical task, but in practice most of the work is in understanding the meaning of the data and the business rules.

**What happened to 2nd Normal Form:**

Our approach took us directly from first normal form (data in tabular form) to third normal form. Most texts treat this as a two-stage process, and deal first with determinants that are part of the table’s key and later with non-key determinants. For example, Hospital Code is part of the key of **Operation**, so we would establish the **Hospital** table in the first stage. Similarly, we would establish the **Drug** and **Standard Drug Dosage** tables as their keys form part of the key of the **Drug Administration** table. At this point we would be in Second Normal Form (2NF), with the **Operation Type** and **Surgeon** information still to be separated out. The next stage would handle these, taking us to 3NF. Most importantly, we only see 2NF as a stage in the process of getting our data fully normalized, never as an end in itself.

**Is 3NF fully normalized:**

Unfortunately, no. There are three further well-established normal forms: Boyce-Codd Normal Form (BCNF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF). We discuss these in Chapter 13. The good news is that in most cases, including this one, data in 3NF is already in 5NF. In particular, 4NF and 5NF problems usually arise only when dealing with tables in which every column is part of the key. By the way, “all key” tables are legitimate and occur quite frequently in fully normalized structures. A Sixth Normal Form (6NF) has been proposed, primarily to deal with issues arising in representing time-dependent data.

**Determinants and Functional Dependency:**

For each value of the determinant, there can only be one value of some other nominated column(s) in the table at any point in time. Equivalently we can say that the other nominated columns are **functionally dependent** on the determinant. The determinant concept is what 3NF is all about; we are simply grouping data items around their determinants.

**Formal 3NF definition:**

The concepts of determinants and candidate keys give us the basis for a more formal definition of Third Normal Form (3NF). If we define the term “nonkey column” to mean “a column that is not part of the primary key,”  
then we can say:  
A table is in 3NF if the only determinants of nonkey columns are candidate keys.7

A foreign key convention in relational notation is to use asterisk

A picture containing text

Description automatically generated

Normalization Summary:

Normalization is a set of techniques for organizing data into tables in such a way as to eliminate certain types of redundancy and incompleteness, and associated complexity and/or anomalies when updating it. The modeler  
starts with a single file and divides it into tables based on dependencies among the data items. While the process itself is mechanistic, the initial data will always contain assumptions about the business that will affect the outcome. The data modeler will need to verify and perhaps challenge these assumptions and the business rules that the data dependencies represent.  
Normalization relies on correct identification of determinants and keys. In this chapter, we covered normalization to third normal form (3NF). A table is in 3NF if every determinant of a nonkey item is a candidate key. A table can be in 3NF but still not fully normalized. Higher normal forms are covered in Chapter 13. In practice, normalization is used primarily as a check on the correctness of a model developed using a top-down approach.

E R Diagram:

The basic Symbols: Boxes and Arrows

We start by presenting our model as a **data structure diagram** using just two symbols:  
1. A “box” (strictly speaking, a rectangle)2 represents a table.  
2. An arrow3 drawn between two boxes represents a foreign key pointing back to the table where it appears as a primary key.

Diagrammatic representation of Foreign keys:

To understand how to draw the arrows, look at the **Operation** and **Surgeon** tables. The primary key of **Surgeon** (Hospital Number + Surgeon Number) appears in the **Operation** table as a foreign key. Draw a line between the two boxes, and indicate the direction of the link by putting a “crow’s foot”4 at the foreign key end (Figure 3.3). You can think of the  
crow’s foot as an arrow pointing back to the relevant surgeon for each operation.

Graphical user interface

Description automatically generated with low confidence

Optionality:

The diagram may also raise the possibility of operations that do not involve any surgeons at all: “We don’t usually involve a surgeon when we are treating a patient with a small cut, but we still need to record whether any drugs were used.” In this case, some rows in the **Operation** table may not contain a value for Surgeon Number. We can show whether the involvement of a surgeon in an operation is **optional** or **mandatory** by using the conventions of Figure 3.5.

You can think of the circle as a zero and the perpendicular bar as a one, indicating the minimum number of surgeons per  
operation or (at the other end of the arrow) operations per surgeon.

Diagram, schematic

Description automatically generated

Diagram

Description automatically generated

Removing redundant arrows:

Look at the arrows linking the **Hospital**, **Operation**, and **Surgeon** tables. There are arrows from **Hospital** to **Surgeon** and from **Surgeon** to **Operation**. Also there is an arrow from **Operation** direct to **Hospital**. Does  
this third arrow add anything to our knowledge of the business rules supported by the model? It tells us that each operation must be performed at one hospital. But we can deduce this from the other two arrows, which specify that each operation must be managed by a surgeon and that each surgeon operates at a hospital. The arrow also shows that a program could “navigate” directly from a row in the **Operation** table to the corresponding row in the **Hospital** table. But our concern is with business rules rather than navigation. Accordingly, we can remove the “short-cut” arrow from the diagram without losing any information about the business rules that the model enforces.

Chart, box and whisker chart

Description automatically generated

A good entity class definition:

A good entity class definition will clearly answer two questions:  
1. What distinguishes instances of this entity class from instances of other entity classes?  
2. What distinguishes one instance from another?

Relationship Diagramming conventions:

We have already used a convention for annotating the lines to describe their meaning (relationship names), **cardinality** (the crow’s foot can be interpreted as meaning “many,” its absence as meaning “one”), and **optionality** (the circles and bars representing “optional” and “mandatory” respectively).

The value of this **assertion** form is in improving communication. While diagrams are great for conveying the big picture, they do not encourage systematic and detailed examination, particularly by business specialists. If we record plural forms of entity class names in our documentation tool, generating these sentences can be an entirely automatic process. Of course, when reading from a diagram we just pluralize the entity class  
names ourselves. Some CASE tools do support such generation of assertions, using more or less similar formulae. We like to use the expression “one or more” rather than “many,” which may have a connotation of “a large number” (“Oh no, nobody would have *many* occupations, two or three would be the most”). We also like the  
“may” and “must” approach to describing optionality, rather than the “zero or more” and “one or more” wording used by some. “Zero or more” is an expression only a programmer could love, and our aim is to communicate  
with business specialists in a natural way without sacrificing precision. An alternative to using “must” and “may” is to use “always” and “sometimes”: “Each company sometimes issues one or more shares,” and “Each share is always issued by one company.” “Might” is also a workable alternative to “may.” In order to be able to automatically translate relationships into assertions about the business data, a few rules need to be established:  
■ We have to select relationship names that fit the sentence structure. It is worth trying to use the same verb in both directions (“hold” and “be held by,” or “be responsible for” and “be the responsibility of”) to ensure that the relationship is not interpreted as carrying two separate meanings.  
■ We have to name the relationships in both directions, even though this adds little to the meaning. We make a practice not only of placing each relationship name close to the entity class that is the object of the sentence, but also of arranging the names above and below the line so they are read in a clockwise direction when generating the sentence (as, for example, in Figure 3.9).  
■ We need to be strict about using singular names for entity classes. As mentioned earlier, this discipline is worth following regardless of relationship naming conventions. Finally, we need to show the optional/mandatory symbol at the crow’s foot end of the relationship, even though this will not usually be enforceable by the DBMS (at the end without the crow’s foot, “optional” is normally implemented by specifying the foreign key column as optional or **nullable**, that is, it does not have to have a value in every row). Despite this there are a number of situations, which we discuss in Section 14.5.3, in which the mandatory nature of a relationship at the crow’s foot end is very important.

Diagram

Description automatically generated

Diagram

Description automatically generated

**Typical Relationship Types:**

A crow’s foot may appear at neither, one, or both ends of a relationship. The three alternatives are referred to as one-to-one, one-to-many, and many-to-many relationships, respectively.  
■ There may be more than one relationship between the same two entity classes.  
■ It is possible for the same entity class to appear at both ends of a relationship. This is called a “self-referencing” or “recursive” relationship.

Diagram

Description automatically generated

Diagram

Description automatically generated

**Self-referencing model having both ends optional:**

Diagram

Description automatically generated

This reflects the fact that the organizational hierarchy has a top and bottom (some employees have no subordinates, one employee has no manager). A mandatory symbol on a self-referencing relationship should always raise  
your suspicions, but it is not necessarily wrong if the relationship represents something other than a hierarchy.

Self-referencing relationships can also be many-to-many. Figure 3.21 shows such a relationship on a **Manufactured Part** entity class. In business terms, we are saying that a part can be made up of parts, which themselves can be made up of parts and so on. Furthermore, we allow a given part to be used in the construction of more than one part—hence, the many-to-many relationship.

Diagram

Description automatically generated

This relationship, being many-to-many, cannot be implemented19 by a single table with suitable foreign key(s). We can, however, resolve it in much the same way as a many-to-many relationship between two different entity classes.

Figure 3.22 shows an intuitive way of tackling the problem directly from the diagram. We temporarily split the **Manufactured Part** entity class in two, giving us a familiar two-entity class many-to-many relationship, which  
we resolve as described earlier. We then recombine the two parts of the split table, taking care not to lose any relationships. Figure 3.23 shows the same result achieved by representing the structure with a repeating group and normalizing. The structure shown in Figure 3.22(d) can be used to represent any self-referencing many-to-many relationship. It is often referred to as the **Bill of Materials** structure, because in manufacturing, a bill of materials lists all the lowest level components required to build a particular product by progressively breaking down assemblies, subassemblies, and so forth. Note that the **Manufactured Part Usage** table holds two foreign keys pointing to **Manufactured Part** (Assembly Manufactured Part Number and Component Manufactured Part Number) to support the two relationships.

**Transferability:**

Diagram

Description automatically generated

**Relationship Names:**

A good example of the need for meaningful names is the relationship between **Country** and **Currency**, as might be required in a database to support foreign currency dealing. Figure 3.29 shows the two entity classes.  
What is the relationship between these two entity classes? One-to-many? Many-to-many? We cannot answer these questions until the meaning of the relationship has been clarified. Are we talking about the fact that currency  
is *issued* by a country, is *legal tender* in the country, or is *able to be traded* in that country? The result of our investigation may well be that we identify more than one relationship between the same pair of entity classes.  
There is an even more fundamental problem here that may affect cardinalities. What do we mean by “country”? Again, a word can have many meanings. Does the Holy See (Vatican City) qualify as a country? If the relationship is “issued by” do we define the Euro as being issued by multiple countries, or do we revise the definition (and name) of the Country entity class to accommodate “European Union,” thus keeping the relationship as  
one-to-many? The point is that definition of the relationship is closely linked to definitions of the participating entity classes. We focus on the entity class definitions first, but our analysis of the relationships may lead us to revise these definitions.

Diagram

Description automatically generated

Let’s look at some further examples of the way in which entity class and relationship definitions interact. Consider Figure 3.30: if the **Customer** entity class represents *all* customers, the relationships are correct since  
every purchase must be made by a customer but not every customer belongs to a loyalty program. However, if the business is an airline or a retail store, it may not keep records of customers other than those in loyalty programs. In this case, not all purchases are made by customers (*as defined in the model*), but all customers (*as defined in the model*) belong to loyalty programs. The relationships should now look like those in Figure 3.31.

Diagram

Description automatically generated

Diagram

Description automatically generated

**SubTypes and SuperTypes:**

The ability to represent different levels of generalization requires a new diagramming convention, the box-in-box. You should be very wary about overcomplicating diagrams with too many different symbols, but this one literally adds another dimension (generalization/specialization) to our models. We call the use of generalization and specialization in a model **subtyping**. **Man** and **Woman** are **subtypes** of **Person**. **Person** is a **supertype** of **Man** and of **Woman.**

Diagram

Description automatically generated

**Surrogate Keys:**

The two arguments most commonly advanced against surrogate keys are programming complexity and performance. Frequently, we need to access a reference table to find the corresponding natural identifier. This situation occurs often enough that programmers are frequently opponents of surrogate keys. However, performance is not usually a problem if the reference tables are small and can reside in primary storage.

The more common performance-related issue with surrogate keys is the need for additional access mechanisms such as indexes to support access on both the surrogate and natural keys.

In databases handling high volumes of new data, problems may also arise with contention for “next available numbers.” However, many DBMSs provide mechanisms specifically to generate unique key values efficiently.

One of the most difficult problems with surrogate keys is the possibility of allocating more than one value to the same real-world object, a violation of **singularity**, which requires that each real-world object be represented by  
only one key value and, hence, only one row in the relevant database table.

**Subtypes and Surrogate Keys:**

If we decide to define a surrogate key at the supertype level, that key will be applicable to all of the subtypes. An interesting question then arises if we choose to implement a different table for each subtype: should we  
allow instances belonging to different subtypes to take the same key value? For example, if we implement **Criminal Case** and **Civil Case** tables, having previously defined a supertype **Legal Case**, should we allocate case numbers as in Figure 6.1(a) or as in 6.1(b)? If contention for “next available number,” as described earlier in this section, is not a serious problem, we recommend you choose option (b). This provides some recognition of the  
supertype in our relational design. A supertype table can then be constructed using the “union” operator and easily joined to tables that hold case numbers as foreign keys (Figure 6.2).

Table

Description automatically generated

Diagram

Description automatically generated

**Structured Keys:**

A **structured key** (sometimes called a “concatenated key” or “composite key”) is technically just a key made up of more than one column.

What we are doing, technically, in these cases is including one or more mandatory foreign keys in the primary key for a table. Most experienced data modelers will automatically do this in at least some cases.  
Structured keys often cause problems, but not because there is anything inherently wrong with multi-attribute keys. Rather, the problem keys usually fail to meet one or more of the basic requirements discussed earlier—in particular, stability.

The rule for using structured keys is straightforward: you can include a foreign key in a primary key only if it represents a *mandatory nontransferable4* relationship. The relationship needs to be mandatory because an optional relationship would mean that some rows would have a null value for the foreign key; hence, the primary key for those rows would be partially null. The problems of nulls in primary key columns are discussed in Section 6.7.  
The reason for the nontransferability may not be so obvious. The problem with transferable relationships is that the value of the foreign key will need to change when the relationship is transferred to a new owner.  
For example, if an employee is transferred from one department to another, the value of Department ID for that employee will change. If the foreign key is part of the primary key, then we have a change in value of the primary  
key, and a violation of our stability criterion. In this example, Department ID should not form part of the primary key of **Employee**

**Programming and Structured Keys:**

Structured keys may simplify programming and improve performance by providing more data items in a table row without violating normalization rules. In Figure 6.4, we are able to determine the department from which a leave application comes without needing to access the **Employee** table. But can an employee transfer from one department to another? If so, the primary key of **Employee** will be unstable—almost certainly an unacceptable price to pay for a little programming convenience and performance. If performance was critically affected by the decision, it would probably be better to carry Department ID redundantly as a nonprimary-key item in the  
**Leave Application** table. In any event, these are decisions for the physical design stage!

Diagram

Description automatically generated

**Running out of number problem with Structured keys:**

Structured keys are prone to a particular kind of stability problem—running out of numbers—which can ultimately require that we reallocate *all* key values. The more parts to a key, the more likely we are to exhaust all possible values for one of them. Of course, this may also imply running out of numbers for the relevant owner entity instances, but the impact on what is often only a reference table may be more local and manageable.  
Incidentally, the owner entity class may not actually be represented by a table in the database; its key may provide sufficient information in itself for our purposes. If we do run out of numbers, it may be prohibitively expensive to redefine the key and amend the programs that use it. Experience suggests that we (or the system users) will be tempted to add new data and meaning to other parts of the key in order to keep the overall value unique. In turn,  
program logic now has to be amended to extract the meaning of the values held in these parts.

Another had to completely redevelop a system because they ran out of insurance agent identifiers (the agent identifier consisted of a State Code, Branch Code within state, and Agent Number within  
state and branch; when all agent numbers for a particular branch had been allocated, new numbers were assigned by creating phantom branches and states). **As a result of problems of this kind, it is often suggested that  
structured keys be avoided altogether.** However, a structured key should involve no more risk than a single-column key, as long as we make adequate provision for growth of each component, and do not break the basic rules of column definition and key design.

**Choosing a Primary Key:**

**We strongly recommend that you always nominate a single primary key for each table.**

The presence of more than one candidate key may be a clue that an entity class should be split into two  
entity classes linked by a one-to-one transferable relationship.

If after this we still genuinely have two (or more) candidate keys for the same entity that are equally applicable and stable, the shortest of these may result in a significant saving in storage requirements, as primary keys are  
replicated in foreign keys and indexes.

**Guidelines for choosing Primary Key:**

We divide the problem into two cases, based on the concepts of dependent and independent entity classes introduced in Section 3.5.7. Recall that a dependent entity class is one that has at least one many-toone mandatory, nontransferable relationship with another entity class. An independent entity class has no such relationships.

**IF Tables implementing Independent Entity Classes:**

The primary key of a table representing an independent entity class must be one of the following:

**1**. A natural identifier: one or more columns in the table corresponding to attributes that are used to identify things in the real world: if you have used the naming conventions outlined in Chapter 5, they will usually be columns with names ending in “Number,” “Code,” or “ID.”  
**2**. A surrogate key: a single column. A sensible general approach to selecting the primary key of an independent entity class is to use natural identifiers when they are available and surrogate keys otherwise.

**IF Tables implementing dependent Entity Classes and Many to Many relationships:**

We have an additional option for the primary key of a table representing a dependent entity class or a many-to-many relationship in that we can include the foreign key(s) representing the relationships to the entity  
classes on which the entity class in question depends. Obviously, a single foreign key alone is not sufficient as a primary key, since that would only allow for one instance of the dependent entity for each instance of the associated entity

The additional options for the primary key of the table representing a dependent entity class are as follows:

1. The foreign key(s) plus one or more existing columns. For example, a scheduled flight will be flown as multiple actual flights; there is therefore a one-to-many relationship between **Scheduled Flight** and **Actual Flight**. Actual flights can be identified by a combination of the Flight No (the primary key of **Scheduled Flight**) and the date on which the actual flight is flown
2. Multiple foreign keys that together satisfy the criteria for a primary key. The classic example of this is the implementation of an intersection entity class (Section 3.5.2) (though this approach will not work for all  
   intersection entity classes (Many-to-Many Relationships, some of which will require options 1 or 3, [i.e., the addition of an existing column (e.g., a date) or a surrogate key)].
3. The foreign key(s) plus a surrogate key. For example, a student could be identified by a combination of the Student ID issued by his or her college and the ID of the college that issued it (the foreign key representing the relationship between **Student** and **College**).

Our general rule is to include all foreign keys that represent dependency relationships, adding a surrogate or (if available) an existing column to ensure uniqueness if necessary. By doing this, we are enforcing non transferability, as long as we stick to the general rule that primary key values cannot be changed. We nearly always use primary keys containing foreign keys for tables representing dependent entity classes, but will sometimes find that such a table has an excellent stand-alone key available. We may then choose to  
trade enforcement of nontransferability for the convenience of using an available “natural” key. For example, it may not be possible for a passport to be transferred from one person to another; hence, we could include  
the key of **Person** in the key of **Passport**, but we may prefer to use a well-established stand-alone Passport Number.

**Boyce-Codd Normal Form:**

**Example of Structure in 3NF but Not in BCNF**Look at the model in Figure 13.1, which represents data about an organization’s branches and how each branch services its customers. Figure 13.2 shows the **Branch-Customer Relationship** table.  
Note three things about this table:  
1. The table enforces the rule that each branch will serve a customerthrough only one salesperson, as there is only one Salesperson No for each combination of Customer No and Branch No. This rule cannot be deduced from the diagram alone. We need the additional information

Diagram

Description automatically generated

**BRANCH-CUSTOMER RELATIONSHIP** (Customer No, Branch No, Visiting Frequency,  
Relationship Establishment Date, Salesperson No)

that Customer No and Branch No form the primary key of the table, so each combination can occur only once. (If the primary key also included Salesperson No, then the table would support multiple salespersons for each combination of branch and customer.)  
2. The table is in 3NF; there are no repeating groups, and every determinant of a nonkey item is a candidate key.  
3. If we are given the additional information that each salesperson works for one branch only, then the table will still have some normalization problems. The fact that a particular salesperson belongs to a particular branch will be recorded in every row in which that salesperson’s identifier appears. The underlying reason for the normalization problems is that we havea dependency between Salesperson No and Branch No; Salesperson No is a**determinant** of Branch No. (A reminder on the terminology: this means that  
 for every Salesperson No, there is only one corresponding Branch No.) The unusual feature here is that Branch No is part of the key. In all our examples so far, we have dealt with determinants of *nonkey* items. We now have a real problem. What we would like to do is set up a reference table with Salesperson No as the key (Figure 13.3).  
 But this does not really help. Although we can now record which branch a salesperson belongs to, regardless of whether he or she is serving any customers, we cannot take anything out of the original table. We would like to remove Branch No, but that would mean destroying the key.  
The trick is to recognize that the original table has another candidate key. We could just as well have used a combination of Salesperson No and Customer No as the primary key (Figure 13.4, next page). The new key suggests a new name for the table: **Customer-Salesperson Relationship**. But now we are no longer in 3NF (in fact not even in 2NF). Salesperson No is a determinant of Branch No, so we need to split these columns off to another table (Figure 13.5, next page). We now have our **Salesperson** reference table, including the foreign  
key to **Branch**, and we have eliminated the problem of repeated data.

**SALESPERSON** (Salesperson No, Branch No)

**CUSTOMER-SALESPERSON RELATIONSHIP** (Customer No, Salesperson No,  
Visiting Frequency, Relationship Established Date, Branch No)

Technically, we have resolved a situation in which the tables were in 3NF but not BCNF.

**Definition of BCNF**For a table to be in BCNF, we require that the following rule be satisfied: *Every determinant must be a candidate key.*In our example, Salesperson No was a determinant of Branch No, but was not a candidate key of **Branch-Customer Relationship**. Compare this with the definition of 3NF: “Every determinant of a *nonkey* column must be a candidate key.” If you compare the two definitions it should be clear that BCNF is stronger than 3NF in the sense that any table in BCNF will also be in 3NF. Situations in which tables may be in 3NF but not BCNF can only occur  
when we have more than one candidate keyto be more precise, *overlapping* candidate keys. We can often spot them more quickly in diagrammatic form. In Figure 13.1, the **Branch-Customer-Relationship** box indicates a three-way relationship between **Branch**, **Customer**, and **Salesperson**. Approaching the problem from an Entity-Relationship perspective, we would normally draw the model as in Figure 13.6, recognizing the direct relationship between **Salesperson** and **Branch**. Any proposed relationship between **Customer-Salesperson Relationship** and **Branch** would then be seen as derivable from the separate relationships between **CustomerSalesperson Relationship** and **Salesperson**, and between **Salesperson** and **Branch**. Taking this top-down approach, we would not have considered holding Branch No as an attribute of **Customer-Salesperson Relationship**, and the BCNF problem would not have arisen.