Department of Electrical and Computer Engineering
Dalhousie University

Decision Tables

Dr. Larry Hughes

Introduction

When analysing a problem that requires the use of conditional selection, system analysts are often left with two nagging doubts:

- have all possible choices been enumerated?

  If one or more choices are missing, the software will not perform according to the specifications. Testing may indicate that the software is in error; however, since the error may remain hidden, the user may be the one to discover the error at some (often embarrassing) later date.

- have the optimal choices been made?

  In large programming projects, it is possible to overlook duplicate or redundant software; this is especially true in situations involving complex decisions. Although a machine may not object to poor software design, a systems analyst should always try to write the best software possible.

Ever since the late 1950s, a great deal of effort has been put into developing techniques that allow the systems analyst to develop software that addresses these two issues. One such technique is known as *decision tables*, in which all possible *conditions* (i.e., guards) and all possible *actions* (i.e., statements) are enumerated. Decision tables are based upon the observation that conditional selection consists of a condition resulting in one or more actions. The purpose of the decision table is to connect the various conditions to their specific actions.

A decision table is one that consists of four distinct fields:

Condition Stubs	Condition Entries
Action Stubs	Action Entries

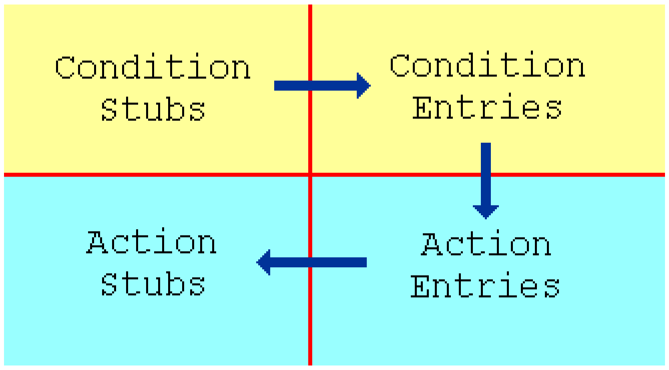
where the *Condition Stubs* are the list of the different conditions that make up the application and the *Action Stubs* are the different actions required by the application. There are two distinct types of *Condition Entries*:

- a *limited entry* format, in which the entire condition is written in the *Condition Stubs* and each condition entry is used to show whether the condition is `TRUE`, `FALSE`, or not pertinent; usually denoted '`Y`', '`N`', or '-', respectively.

- an *extended entry* format, in which part of the *Condition Stub* is written directly into the *Condition Entry*. For example, the entry could contain an indication of whether a certain variable was less than, equal to, or greater than a particular value.

An *Action Entry* indicates whether the associated *Action Stub* should be performed ('`X`' denotes that the action is to be performed, while a blank means the action should be ignored).

Condition stubs are connected to their related action stubs via *rules*. A rule is a vertical column that includes one or more condition entries and one or more action entries. When the condition stub(s) associated with a given rule are met, the action entries included in the rule denote the action stub(s) to be performed. The cycle of a decision table can be shown as follows:



Example

A certain water company charges its clients by the amount of water used; the more used, the greater the charge. Water billing is as follows: $5.00 minimum charge for 1 m^3; $5.00 per m^3 for a demand between 1 and 5 m^3; $5.50 per m^3 for usage between 5 and 10 m^3; and $6.00 per m^3 for water usage above 10 m^3. Write two decision tables, one with limited entry, the other with extended entry.

Before attempting to write the decision tables, it is necessary to determine the conditions and the associated actions. In this example, the conditions are:

- Demand of less than 1 m^3
- Demand between 1 and 5 m^3
- Demand between 5 and 10 m^3
- Demand greater than 10 m^3

The actions can be listed as follows:

- Apply minimum charge of $5.00
- Charge $5.00 per m^3
- Charge $5.50 per m^3
- Charge $6.00 per m^3

In a limited entry decision table, the condition entries are listed as '`Y`', '`N`', and '`-`'. The water meter limited entry decision table could be written as follows:

	1	2	3	4
Demand of less than 1 m^3	Y	-	-	-
Demand between 1 and 5 m^3	-	Y	-	-
Demand between 5 and 10 m^3	-	-	Y	-
Demand greater than 10 m^3	-	-	-	Y
Apply minimum charge of $5.00	X			

Charge $5.00 per m^3		X		
Charge $5.50 per m^3			X	
Charge $6.00 per m^3				X

An extended entry table contains less condition stubs because the condition stubs are extended into the condition entries. In this example, one could consider the condition stub to be "Demand of", and the extended entries "Less than 1", "1 to 5", "5 to 10", and "More than 10".

The entire table could be written as follows:

	1	**2**	**3**	**4**
Demand of (in m^3)	Less than 1	1 to 5	5 to 10	More than 10
Apply minimum charge of $5.00	X			
Charge $5.00 per m^3		X		
Charge $5.50 per m^3			X	
Charge $6.00 per m^3				X

Translating Decision Tables

Decision tables can be translated into `IF-ELSE` statements by observing that the condition stub is a guard and the actions dictated by the rules are statements. For example, consider the following two rules from a particular decision table (the rules are R1 and R2):

	R1	**R2**
Condition 1	-	Y
Condition 2	Y	-
Condition 3	-	Y
Condition 4	-	Y
Action 1	X	X
Action 2	X	
Action 3		X

For rule R1 to come into effect, *Condition 2* must be `TRUE`; if this is the case, *Action 1* and *Action 2* are performed. This can be written as a structured English `IF-ENDIF` statement:

```
IF Condition 2
     Action 1;
     Action 2;
END IF
```

However, for the actions associated with rule R2 to be performed, three conditions must be `TRUE`: *Condition 1*, *Condition 3*, and *Condition 4*. The `IF-ENDIF` statement for this rule can be written as follows:

```
IF Condition 1 AND Condition 3 AND Condition 4
     Action 1;
     Action 3;
END IF
```

Example

As an example, consider a train reservation system in which passengers can request either a sleeping compartment or a seat in coach class. Since there are a finite number of sleeping compartments and seats in coach class, either a ticket will be issued or the passenger will be placed on a waiting list. The conditions in this example are:

- *Passenger requests a sleeping compartment (Yes or No).*
- *Passenger requests a coach class seat (Yes or No).*
- *Sleeping compartments are available (Yes or No).*
- *Coach class seats are available (Yes or No).*

There are three possible actions:

- *Issue a sleeping compartment ticket.*
- *Issue a coach class ticket.*
- *Place passenger on waiting list.*

A first attempt at the decision table can now be made:

	1	**2**	**3**	**4**
Passenger requests a sleeping compartment	Y	Y	-	-
Passenger requests a coach class seat	-	-	Y	Y
Sleeping compartments are available	Y	N	-	-
Coach class seats are available	-	-	Y	N
Issue a sleeping compartment ticket	X			
Issue a coach class ticket			X	
Place passenger on waiting list		X		X

This table consists of four rules. For example, rule 1 states that *if* a passenger requests a sleeping compartment *and* there are sleeping compartments available, *then* issue a sleeping compartment ticket. Note that in each rule there are a number of conditions that are not pertinent; for example, in rule 1, the passenger cannot request a coach seat (because a sleeping compartment is being requested), nor does it matter whether there are coach class seats available.

Decision tables also help in the development of optimal code. Consider, for example, the above ticket issuing software; one possible implementation could be to check for each rule in turn:

```
IF Sleeping compartment requested AND Sleeping compartments available
   /* Rule 1 */
   Issue a sleeping compartment ticket
ELSE
   IF Sleeping compartment requested AND No sleeping compartments available
      /* Rule 2 */
      Place passenger on waiting list
   ELSE
      IF Coach seat requested AND Coach seats available
         /* Rule 3 */
         Issue a coach seat ticket
      ELSE
         IF Coach seat requested AND No coach seats available
            /* Rule 4 */
           Place passenger on waiting list
         END IF
      END IF
    END IF
END IF
```

An examination of the above code fragment reveals that, for example, two checks are made on whether the passenger requested a sleeping compartment. Re-examining the decision table shows that rules 1 and 2 are performed if a sleeping compartment request has been made:

	1	**2**
Passenger requests a sleeping compartment	Y	Y
Sleeping compartments are available	Y	N
Issue a sleeping compartment ticket	X	
Place passenger on waiting list		X

Furthermore, from this table, one can see that *if* a sleeping compartment is available, *then* a sleeping compartment ticket is issued, *otherwise* the passenger is placed on the waiting list.

This means that a considerably abbreviated version of the decision table implementation is possible:

```
IF Passenger requests a sleeping compartment
   /* This is either rule 1 or rule 2 */
   IF Sleeping compartments are available
      /* Rule 1 */
      Issue a sleeping compartment ticket
   ELSE
      /* Rule 2 */
      Place Passenger on waiting list
   END IF
ELSE
   IF Passenger requests a coach class seat
      /* This is either rule 3 or rule 4 */
      IF Coach class seats are available
         /* Rule 3 */
         Issue a coach class ticket
      ELSE
         /* Rule 4 */
         Place Passenger on waiting list
      END IF
   END IF
END IF
```

There are some important differences between this code fragment and the previous. First, the total number of relational expressions that are required drops from eight to four. Second, the number of times the same relational expression is written drops from two to zero. Third, in the worst case, a total of three relational expressions are executed as compared to eight in the previous code fragment. The reason this optimization could take place is due to the decision table. By searching for patterns in the table, one can develop better software.

Types of Decision Tables

A decision table in which the condition stubs associated with each rule are listed in full is known as a *skinny* `IF-ELSE`; whereas, using selected condition stubs to build up several rules is known as a *network* `IF-ELSE`. The first code fragment is an example of a skinny `IF-ELSE`, while the second is an example of a network `IF-ELSE`.

Many applications require considerably more complex conditions than those shown in the previous example. It may reach the point where it is next to impossible to keep track of the various conditions and their associated actions, let alone try to write optimal code from the decision table. This being the case, techniques are available to *compress* the decision table, thereby removing redundant or unnecessary rules. Consider the following example:

An eco-tourism company offers tours of the high-Arctic in a double-decker snowmobile. Tourists can choose to either sit in the open, top deck or in the enclosed, lower deck. Once a seat has been selected for either deck, the total number of available seats for the deck in question is decremented. If there are no seats available on the requested deck, the tourist is offered the choice of either waiting for a seat to become available or to accept a space on the other deck (assuming one is available).

Design the decision table for the eco-tourism company, then show the `IF` statement for the decision table.

As with all decision tables, it is necessary to determine the conditions and actions. In this case, the conditions are:

- Upper deck requested
- Lower deck requested
- Upper deck available
- Lower deck available
- Alternate deck acceptable

and the actions are:

- Issue upper deck ticket
- Issue lower deck ticket
- Decrease upper deck total
- Decrease lower deck total
- Onto upper deck waiting list
- Onto lower deck waiting list

The next step is to *enumerate* all possible conditions. Since there is a total of two possible choices per condition stub and there are five condition stubs, then the total number of rules are 2^5 or 32; resulting in the following condition entries:

```
Upper requested  Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y N N N N N N N N N N N N N N N N
Lower requested  Y Y Y Y Y Y Y Y N N N N N N N N Y Y Y Y Y Y Y Y N N N N N N N N
Upper available  Y Y Y Y N N N N Y Y Y Y N N N N Y Y Y Y N N N N Y Y Y Y N N N N
Lower available  Y Y N N Y Y N N Y Y N N Y Y N N Y Y N N Y Y N N Y Y N N Y Y N N
Accept Alternate Y N Y N Y N Y N Y N Y N Y N Y N Y N Y N Y N Y N Y N Y N Y N Y N
```

Tables this size quickly become unmanageable; fortunately, the number of rules can be halved simply by observing that a tourist asks for either an upper deck seat or a lower deck seat. Since the choice is either one or the other (not both), they are said to be *mutually exclusive*; in other words, the 'N' (i.e., `FALSE`) of an "upper deck request" *must* be a "lower deck request". Applying this knowledge to the table one finds a considerably more manageable table:

Upper deck requested	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Upper deck available	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Lower deck available	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Alternate deck acceptable	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N

Now that the enumeration of the various condition stub combinations has been completed, it is necessary to apply these combinations to determine which actions are envoked. This is done by looking at each condition entry and determining which actions apply. For example, in rule 1 (all condition entries are 'Y'), the actions to be performed are "issue upper deck ticket" and "decrease upper deck total".

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Upper deck requested	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Upper deck available	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Lower deck available	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Alternate deck acceptable	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Issue upper deck ticket	X	X	X	X							X					
Issue lower deck ticket					X				X	X			X	X		
Decrease upper deck total	X	X	X	X							X					
Decrease lower deck total					X				X	X			X	X		
Onto upper deck waiting list						X	X	X							X	
Onto lower deck waiting list							X					X			X	X

At this point one could start to implement the decision table; however, it is worth taking a closer look at the various rules. For example, rules `1`, `2`, `3`, and `4` all result in the same actions (i.e., "issue upper deck ticket" and "decrease upper deck total"). With these four rules, once it has been established that the request is for an upper deck ticket and that upper deck seats are available, there is no need to determine if a lower deck seat is available or if the tourist is willing to take an alternate deck. In other words, for these rules, it doesn't matter what the results of the last two conditions are (i.e., "lower deck available" and "alternate deck acceptable") since their values cannot affect the decision. These four rules can be *compressed* into one rule in which only the first two conditions are evaluated (the 'New' rule):

	1	**2**	**3**	**4**	**New**
Upper deck requested	Y	Y	Y	Y	Y
Upper deck available	Y	Y	Y	Y	Y
Lower deck available	Y	Y	N	N	-
Alternate deck acceptable	Y	N	Y	N	-
Issue upper deck ticket	X	X	X	X	X
Issue lower deck ticket					
Decrease upper deck total	X	X	X	X	X
Decrease lower deck total					
Onto upper deck waiting list					

Onto lower deck waiting list					

the '-' in the last two condition entries means *don't care*. In other words, this rule is performed if the first two entries are `TRUE`; the remaining entries are ignored.

Two or more rules can be considered for compression when they have the *same* action entries and one or more identical condition entries. Those condition entries that are the same for different rules remain unchanged; however, those that are different become don't care entries. Don't care entries are formed when a condition entry can be *either* a 'Y' and a 'N' entry.

The other compressions that can take place on this table are as follows:

- Rules `6` and `8` which have the same action "onto upper deck waiting list", can be compressed into a single rule. Condition stub "lower deck available" is a don't care because the tourist has requested an upper deck and will not accept a lower deck:

	6	**8**	**New**
Upper deck requested	Y	Y	Y
Upper deck available	N	N	N
Lower deck available	Y	N	-
Alternate deck acceptable	N	N	N
Issue upper deck ticket			
Issue lower deck ticket			
Decrease upper deck total			
Decrease lower deck total			
Onto upper deck waiting list	X	X	X
Onto lower deck waiting list			

- Rules `9`, `10`, `13`, and `14`, all with actions "issue lower deck ticket" and "decrease lower deck total", can be compressed into a single rule. Condition stubs "upper deck available" and "lower deck available" are don't cares in this new rule because a lower deck seat is requested and one is available:

	9	**10**	**13**	**14**	**New**
Upper deck requested	N	N	N	N	N
Upper deck available	Y	Y	N	N	-
Lower deck available	Y	Y	Y	Y	Y
Alternate deck acceptable	Y	N	Y	N	-
Issue upper deck ticket					
Issue lower deck ticket	X	X	X	X	X
Decrease upper deck total					
Decrease lower deck total	X	X	X	X	X
Onto upper deck waiting list					
Onto lower deck waiting list					

- Rules 12 and 16, sharing action "onto lower deck waiting list", can be compressed into a single rule with condition stub "upper deck available" as a don't care because the tourist will only accept a lower deck seat:

	12	**16**	**New**
Upper deck requested	N	N	N
Upper deck available	Y	N	-
Lower deck available	N	N	N
Alternate deck acceptable	N	N	N
Issue upper deck ticket			
Issue lower deck ticket			
Decrease upper deck total			
Decrease lower deck total			
Onto upper deck waiting list			
Onto lower deck waiting list	X	X	X

- Rules 7 and 15, with actions "onto upper deck waiting list" and "onto lower deck waiting list" can also be compressed. This new rule states that it doesn't matter what seat has been selected, there are none available, but the tourist is willing to take either an upper or lower deck seat:

	7	**15**	**New**
Upper deck requested	Y	N	-
Upper deck available	N	N	N
Lower deck available	N	N	N
Alternate deck acceptable	Y	Y	Y
Issue upper deck ticket			
Issue lower deck ticket			
Decrease upper deck total			
Decrease lower deck total			
Onto upper deck waiting list	X	X	X
Onto lower deck waiting list	X	X	X

- Rules 5 and 11 remain unchanged because although they share actions with other rules, including them with other rules would make the compressed rule ambiguous. For example, had rule 5 been compressed with rules 1 through 4, the new rule would have been concerned with condition stub "upper deck available" only; ignoring all other conditions. This would result in, for example, upper deck seats being allocated to people who wanted lower deck seats.

  Rules 5 and 11 are therefore:

	5	**11**
Upper deck requested	Y	N
Upper deck available	N	Y

Lower deck available	Y	N
Alternate deck acceptable	Y	Y
Issue upper deck ticket		X
Issue lower deck ticket	X	
Decrease upper deck total		X
Decrease lower deck total	X	
Onto upper deck waiting list		
Onto lower deck waiting list		

Once the rules have been compressed, the compressed version of the decision table can be written. As a rule of thumb, newly compressed rules should be listed by their similarities, thereby emphasizing the patterns and potentially making the implementation easier.

The compressed decision table for the eco-tourism problem can be written as follows (rules `1` though `4` become rule `1`; rules `6` and `8` become rule `2`; rule `5` becomes rule `3`; rules `9`, `10`, `13`, and `14` become rule `4`; rules `12` and `16` become `5`; rule `11` becomes rule `6`; and rules `7` and `15` become rule `7`):

	1	**2**	**3**	**4**	**5**	**6**	**7**
Upper deck requested	Y	Y	Y	N	N	N	-
Upper deck available	Y	N	N	-	-	Y	N
Lower deck available	-	-	Y	Y	N	N	N
Alternate deck acceptable	-	N	Y	-	N	Y	Y
Issue upper deck ticket	X					X	
Issue lower deck ticket			X	X			
Decrease upper deck total	X					X	
Decrease lower deck total			X	X			
Onto upper deck waiting list		X					X
Onto lower deck waiting list					X		X

The decision table has been compressed from 32 possible rules down to 7. Now all that remains is to implement these rules.

Rule `7` will be implemented first, as a skinny `IF-ELSE`, since this rule is not concerned with the type of deck requested:

```
IF NOT(Upper deck available) AND NOT(Lower deck available) AND Alternate deck acceptable
   /* Rule 7 */
   Onto upper deck waiting list;
   Onto lower deck waiting list;
ELSE
   /* Remaining rules */
END IF
```

The remaining rules can be implemented using a series of networked `IF-ELSE` statements:

```
IF Upper deck requested
   /* Rules 1, 2, and 3 */
   IF Upper deck available
      /* Rule 1 */
      Issue upper deck ticket;
      Decrease upper deck total;
   ELSE /* Upper deck NOT available */
      /* Rules 2 and 3 */
      IF Alternate deck acceptable
         IF Lower deck available
            /* Rule 3 */
            Issue lower deck ticket;
            Decrease lower deck total;
         END IF
      ELSE /* Alternate deck NOT acceptable */
         /* Rule 2 */
         Onto upper deck waiting list;
      END IF
   END IF
ELSE  /* Upper deck NOT requested -- lower deck */
   IF Lower deck available
      /* Rule 4 */
      Issue lower deck ticket;
      Decrease lower deck total;
   ELSE /* Lower deck NOT available */
      /* Rules 5 and 6 */
      IF Alternate deck acceptable
         IF Upper deck available
            /* Rule 6 */
            Issue upper deck ticket;
            Decrease upper deck total;
         END IF
      ELSE /* Alternate deck NOT acceptable */
         /* Rule 5 */
         Onto lower deck waiting list;
      END IF
   END IF
END IF
```

This example has shown the power of decision tables in helping design optimal guards and their associated statements. By putting one's effort into the *analysis* of the problem, the total design, implementation, and testing time can be reduced.

© 2005 -- Whale Lake Press