to handle such an explosion in the number of equivalence classes are discussed in Sections 2.2, 2.6, and in Chapter 4.

4. *Identify infeasible equivalence classes*: An infeasible equivalence class is one that contains a combination of input data that cannot be generated during test. Such an equivalence class might arise due to several reasons. For example, suppose that an application is tested via its GUI, that is data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering a palette of valid inputs only. There might also be constraints in the requirements that render certain equivalence class infeasible.

   Infeasible data is a combination of input values that cannot be input to the application under test. Again, this might happen due to the filtering of the invalid input combinations by a GUI. While it is possible that some equivalence classes are wholly infeasible, the likely case is that each equivalence class contains a mix of testable and infeasible data.

   In this step we remove from E equivalence classes that contain infeasible inputs. The resulting set of equivalence classes is then used for the selection of tests. Care needs to be exercised while selecting tests as some data in the reduced E might also be infeasible.

Example 2.8: Let us now apply the equivalence partitioning procedure described earlier to generate equivalence classes for the software control portion of a boiler controller. A simplified set of requirements for the control portion are as follows:

> *Consider a boiler control system (BCS). The control software of BCS, abbreviated as CS, is required to offer one of several options. One of the options, C (for control), is used by a human operator to give one of the three commands (cmd): change the boiler temperature (temp), shut down the boiler (shut), and cancel the request (cancel). Command temp causes CS to ask the operator to enter the amount by which the temperature is to be changed (tempch). Values of tempch are in the range [−10...10] in increments of 5 degrees Fahrenheit. A temperature change of 0 is not an option.*
>
> *Selection of option C forces the BCS to examine V. If V is set to* GUI, *the operator is asked to enter one of the three commands via a GUI. However, if V is set to* file, *BCS obtains the command from a command file.*
>
> *The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is* temp. *The file name is obtained from variable F. Values of V and F can be altered by a different module in BCS. In response to*

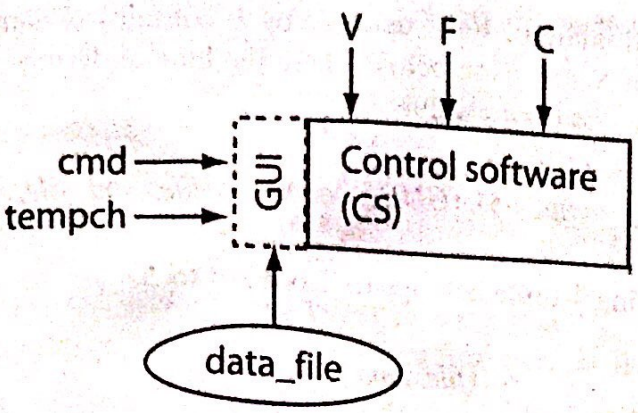

Fig. 2.5 Inputs for the boiler-control software. *V* and *F* are environment variables. Values of *cmd* (command) and *tempch* (temperature change) are input via the GUI or a data file depending on *V*. *F* specifies the data file.

temp and shut commands, the control software is required to generate appropriate signals to be sent to the boiler heating system.

We assume that the control software is to be tested in a simulated environment. The tester takes on the role of an operator and interacts with the CS via a GUI. The GUI forces the tester to select from a limited set of values as specified in the requirements. For example, the only options available for the value of *tempch* are −10, −5, 5, and 10. We refer to these four values of *tempch* as *t_valid* while all other values as *t_invalid*. Figure 2.5 is a schematic of the GUI, the control software under test, and the input variables.

Identify the input domain: The first step in generating equivalence partitions is to identify the (approximate) input domain. Recall that the domain identified in this step is likely to be a superset of the true input domain of the control software. First we examine the requirements, identify input variables, their types, and values. These are listed in the following table:

Variable	Kind	Type	Value(s)
V	Environment	Enumerated	{*GUI*, *file*}
F	Environment	String	A file name
cmd	Input via GUI or file	Enumerated	{*temp*, *cancel*, *shut*}
tempch	Input via GUI or file	Enumerated	{−10, −5, 5, 10 }

Considering that each variable name defines a set, we derive the following set as a product of the four variables listed above:

$$S = V \times F \times cmd \times tempch.$$

The input domain of *BCS*, denoted by $\mathcal{I}$, contains S. Sample values in $\mathcal{I}$, and in S, are given below where the lone underscore character (_) denotes a *don't care* value.

$$(GUI,\_, temp, -5), (GUI,\_, cancel,\_), (file, cmd\_file, shut,\_)$$

The following 4-tuple belongs to $\mathcal{I}$ but not to S:

$$(file, cmd\_file, temp, 0)$$

Equivalence classing: Equivalence classes for each variable are given in the table below. Recall that for variables that are of an enumerated type, each value belongs to a distinct class.

Variable	Partition
V	{{*GUI*}, {*file*}, {*undefined*}}
F	f_valid, $f\_invalid$
cmd	{{*temp*}, {*cancel*}, {*shut*}, {*c_invalid*}
tempch	{*t_valid*}, {*t_invalid*}

f_valid denotes a set of names of files that exist, *f_invalid* denotes the set of names of nonexistent files, *c_invalid* denotes the set of invalid commands specified in F, *t_invalid* denotes the set of out-of-range values of *tempch* in the file, and *undefined* indicates that the environment variable V is undefined. Note that *f_valid*, *f_invalid*, *c_invalid*, and *t_invalid* denote sets of values whereas *undefined* is a singleton indicating that V is undefined.

Combine equivalence classes: Variables V, F, *cmd*, and *tempch* have been partitioned into 3, 2, 4, and 2 subsets, respectively. Set product of these four variables leads to a total of $3 \times 2 \times 4 \times 5 = 120$ equivalence classes. A sample follows:

{(*GUI*, *f_valid*, *temp*, −10)},
{(*file*, *f_invalid*, *c_invalid*, 5)},
{(*file*, *f_valid*, *temp*, −10)},
{(*GUI*, *f_valid*, *temp*, *t_invalid*)},
{(*undefined*, *f_valid*, *temp*, *t_invalid*)},
{(*file*, *f_valid*, *temp*, −5)}

Note that each of the classes listed above represents an infinite number of input values for the control software. For example, {(*GUI*, *f_valid*, *temp*, −10)} denotes an infinite set of values obtained by replacing *f_valid* by a string that corresponds to the name of an existing file. As we shall see later, each value in an equivalence class is a potential test input for the control software.

Discard infeasible equivalence classes: Note that the amount by which the boiler temperature is to be changed is needed only when the operator selects *temp* for *cmd*. Thus all equivalence classes that match the following template are infeasible.

$$\{(V, F, \{cancel, shut, c\_invalid\}, t\_valid \cup t\_invalid)\}$$

This parent–child relationship between *cmd* and *tempch* renders infeasible a total of $3 \times 2 \times 3 \times 5 = 90$ equivalence classes.

Next, we observe that the GUI does not allow invalid values of temperature change to be input. This leads to two more infeasible equivalence classes given below.

$\{(GUI, f\_valid, temp, t\_invalid)\}$ *The GUI does not allow invalid values of temperature change to be input.*

$\{(GUI, f\_invalid, temp, t\_invalid)\}$

Continuing with similar argument, we observe that a carefully designed application might not ask for the values of *cmd* and *tempch* when $V = file$ and F contains a file name that does not exist. In this case, five additional equivalence classes are rendered infeasible. Each of these five classes is described by the following template:

$$\{(file, f\_invalid, temp, t\_valid \cup t\_invalid)\}.$$

Along similar lines as above, we can argue that the application will not allow values of *cmd* and *tempch* to be input when V is undefined. Thus, yet another five equivalence classes that match the following template are rendered infeasible.

$$\{(undefined, \_, temp, t\_valid \cup t\_invalid)\}$$

Note that even when V is undefined, F can be set to a string that may or may not represent a valid file name.

The above discussion leads us to discard a total of $90 + 2 + 5 + 5 = 102$ equivalence classes. We are now left with only 18 testable equivalence classes. Of course, our assumption in discarding 102 classes is that the application is designed so that certain combinations of input values are impossible to achieve while testing the control software. In the absence of this assumption, all 120 equivalence classes are potentially testable.

The set of 18 testable equivalence classes match the seven templates listed below. The "_" symbol indicates, as before, that the data can be input but is not used by the control software, and the "*NA*"

indicates that the data cannot be input to the control software because the GUI does not ask for it.

$\{(GUI, f\_valid, temp, t\_valid)\}$	*four equivalence classes.*
$\{(GUI, f\_invalid, temp, t\_valid)\}$	*four equivalence classes.*
$\{(GUI, \_, cancel, NA)\}$	*two equivalence classes.*
$\{(file, f\_valid, temp, t\_valid \cup t\_invalid)\}$	*five equivalence classes.*
$\{(file, f\_valid, shut, NA)\}$	*one equivalence class.*
$\{(file, f\_invalid, NA, NA)\}$	*one equivalence class.*
$\{(undefined, NA, NA, NA)\}$	*one equivalence classes.*

There are several input data tuples that contain don't care values. For example, if $V = GUI$, then the value of F is not expected to have any impact on the behavior of the control software. However, as explained in the following section, a tester must interpret such requirements carefully.

2.3.6 Test Selection Based on Equivalence Classes

Given a set of equivalence classes that form a partition of the input domain, it is relatively straightforward to select tests. However, complications could arise in the presence of infeasible data and don't care values. In the most general case, a tester simply selects one test that serves as a representative of each equivalence class. Thus, for example, we select four tests, one from each equivalence class defined by the *pCat* relation in Example 2.4. Each test is a string that denotes the make and model of a printer belonging to one of the three classes or is an invalid string. Thus, the following is a set of four tests selected from the input domain of program *pTest*.

```
T = {HP cp1700, Canon Laser Shot LBP 3200,
     Epson Stylus Photo RX600, My Printer
    }
```

While testing `pTest` against tests in T we assume that if `pTest` correctly selects a script for each printer in T, then it will select correct scripts for all printers in its database. Similarly, from the six equivalence classes in Example 2.5, we generate the following set T consisting of six tests each of the form (w, f), where w denotes the value of an input word and f a filename.

```
T = {(Love, my-dict), (Hello, does-not-exist),
     (Bye, empty-file), (ϵ, my-dict),
     (ϵ, does-not-exist), (ϵ, empty-file)
    }
```

In the test above, ϵ denotes an empty, or a null, string, implying that the input word is a null string. Following values of *f my-dict, does-not-exist,*

and *empty-file* correspond to names of files that are, respectively, *exists*, *does-not-exist*, and *exists* but is empty.

Selection of tests for the boiler-control software in Example 2.8 is a bit more tricky due to the presence of infeasible data and don't care values. The next example illustrates how to proceed with test selection for the boiler-control software.

Example 2.9: Recall that we began by partitioning the input domain of the boiler-control software into 120 equivalence classes. A straightforward way to select tests would have been to pick one representative of each class. However, due to the existence of infeasible equivalence classes, this process would lead to tests that are infeasible. We, therefore, derive tests from the reduced set of equivalence classes. This set contains only feasible classes. Table 2.3 lists

Table 2.3 Test data for the control software of a boiler control system in Example 2.8.

ID	Equivalence class[a] {(*V, F, cmd, tempch*)}	Test data[b] (*V, F, cmd, tempch*)
E1	{(*GUI, f_valid, temp, t_valid*)}	(*GUI, a_file, temp*, −10)
E2	{(*GUI, f_valid, temp, t_valid*)}	(*GUI, a_file, temp*, −5)
E3	{(*GUI, f_valid, temp, t_valid*)}	(*GUI, a_file, temp*, 5)
E4	{(*GUI, f_valid, temp, t_valid*)}	(*GUI, a_file, temp*,10)
E5	{(*GUI, f_invalid temp, t_valid*)}	(*GUI, no_file, temp*, −10)
E6	{(*GUI, f_invalid temp, t_valid*)}	(*GUI, no_file, temp*, −10)
E7	{(*GUI, f_invalid temp, t_valid*)}	(*GUI, no_file, temp*, −10)
E8	{(*GUI, f_invalid temp, t_valid*)}	(*GUI, no_file, temp*, −10)
E9	{(*GUI*,_, *cancel*, NA)}	(*GUI, a_file, cancel*, −5)
E10	{(*GUI*,_, *cancel*, NA)}	(*GUI, no_file, cancel*, −5)
E11	{(*file, f_valid, temp, t_valid*)}	(*file, a_file, temp*, −10)
E12	{(*file, f_valid, temp, t_valid*)}	(*file, a_file, temp*, −5)
E13	{(*file, f_valid, temp, t_valid*)}	(*file, a_file, temp*, 5)
E14	{(*file, f_valid, temp, t_valid*)}	(*file, a_file, temp*, 10)
E15	{(*file, f_valid, temp, t_invalid*)}	(*file, a_file, temp*, −25)
E16	{(*file, f_valid, temp*, NA)}	(*file, a_file, shut*, 10)
E17	{(*file, f_invalid*, NA, ,NA)}	(*file, no_file, shut*, 10)
E18	{(*undefined*, _, NA, NA)}	(*undefined, no_file, shut*, 10)

[a] —, don't care; NA, input not allowed.
[b] *a_file*, file exists; *no_file*, file does not exist.

18 tests derived from the 18 equivalence classes listed in Example 2.8. The equivalence classes are labeled as E1, E2, and so on for ease of reference.

While deriving the test data in Table 2.3, we have specified arbitrary values of variables that are not expected to be used by the control software. For example, in E9, the value of F is not used. However, to generate a complete test data item, we have arbitrarily assigned to F a string that represents the name of an existing file. Similarly, the value of *tempch* is arbitrarily set to 10.

The don't care values in an equivalence class must be treated carefully. It is from the requirements that we conclude that a variable is or is not don't care. However, an incorrect implementation might actually make use of the value of a don't care variable. In fact, even a correct implementation might use the value of a variable marked as don't care. This latter case could arise if the requirements are incorrect or ambiguous. In any case, it is recommended that one assign data to don't care variables, given the possibility that the program may be erroneous and hence make use of the assigned data value.

In Step 3 of the procedure to generate equivalence classes of an input domain, we suggested the product of the equivalence classes derived for each input variable. This procedure may lead to a large number of equivalence classes. One approach to test selection that avoids the explosion in the number of equivalence classes is to cover each equivalence class of each variable by a small number of tests. We say that a test input covers an equivalence class E for some variable V, if it contains a representative of E. Thus, one test input may cover multiple equivalence classes, one for each input variable. The next example illustrates this method for the boiler example.

Example 2.10: In Example 2.8 we derived 3, 2, 4, and 5 equivalence classes, respectively, for variables V, F, *cmd*, and *tempch*. The total number of equivalence classes is only 15; compare this with 120 derived using the product of individual equivalence classes. Note that we have five equivalence classes, and not just two, for variable *tempch*, because it is an enumerated type. The following set T of 5 tests cover each of the 14 equivalence classes.

```
T={(GUI, a file, temp, -10), (GUI, no_ file, temp, -5),
  (file, a_ file, temp, 5), (file, a_ file, cancel, 10),
  (undefined, a_ file, shut, -10)
  }
```

You may verify that the tests listed above cover each equivalence classes of each variable. Though small, the above test set has several

weaknesses. While the test covers all individual equivalence classes, it fails to consider the semantic relations among the different variables. For example, the last of the tests listed above will not be able to test the *shut* command assuming that the value *undefined* of the environment variable V is processed correctly.

Several other weaknesses can be identified in T of Example 2.10. The lesson to be learned from this example is that one must be careful, that is consider the relationships among variables, while deriving tests that cover equivalence classes for each variable. In fact, it is easy to show that a small subset of tests from Table 2.3 will cover all equivalence classes of each variable and satisfy the desired relationships among the variables (see Exercise 2.13).

2.3.7 GUI Design and Equivalence Classes

Test selection is usually affected by the overall design of the application under test. Prior to the development of GUI, data was input to most programs in textual forms through typing on a keyboard, and many years ago, via punched cards. Most applications of today, either new or refurbished, exploit the advancements in GUIs to make interaction with programs much easier and safer than before. Test design must account for the constraints imposed on data by the front-end application GUI.

While designing equivalence classes for programs that obtain input exclusively from a keyboard, one must account for the possibility of errors in data entry. For example, the requirement for an application A places a constraint on an input variable X such that it can assume integral values in the range $0 \ldots 4$. However, a user of this application may inadvertently enter a value for X that is out of range. While the application is supposed to check for incorrect inputs, it might fail to do so. Hence, in test selection using equivalence partitioning, one needs to test the application using at least three values of X, one that falls in the required range and two that fall outside the range on either side. Thus three equivalence classes for X are needed. Figure 2.6(a) illustrates this situation where the *incorrect values* portion of the input domain contains the out-of-range values for X.

However, suppose that all data entry to application A is via a GUI front-end. Suppose also that the GUI offers exactly five correct choices to the user for X. In such a situation, it is impossible to test A with a value of X that is out of range. Hence, only the correct values of X will be input to A. This situation is illustrated in Figure 2.6(b).

Of course, one could dissect the GUI from the core application and test the core separately with correct and incorrect values of X. But any error so discovered in processing incorrect values of X will be

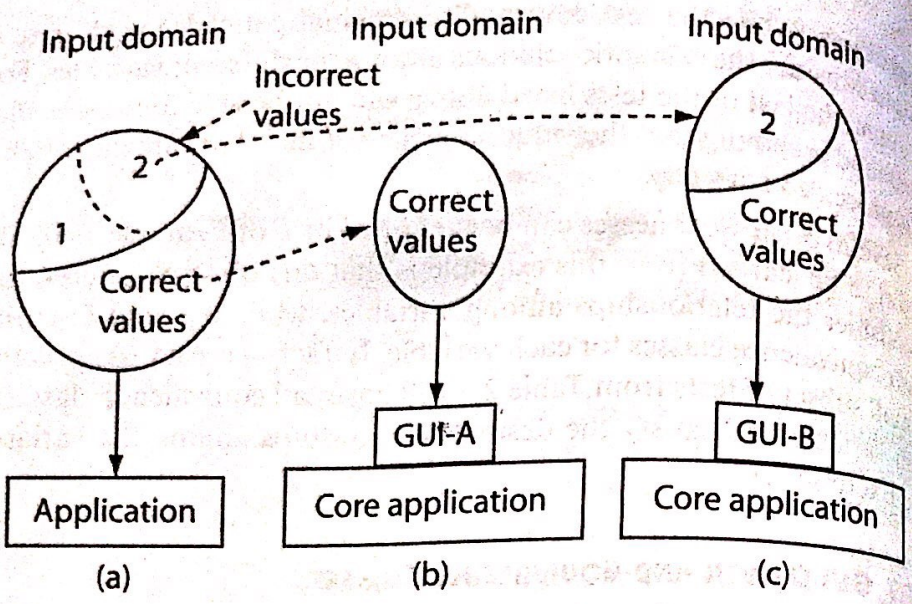

Fig. 2.6 Restriction of the input domain through careful design of the GUI. Partitioning of the input domain into equivalence classes must account for the presence of GUI as shown in (b) and (c). GUI-A protects all variables against incorrect input while GUI-B does allow the possibility of incorrect input for some variables.

meaningless because, as a developer may argue, in practice the GUI would prevent the core of A from receiving an invalid input for X. In such a situation, there is no need to define an equivalence class that contains incorrect values of an input variable.

In some cases a GUI might ask the user to type in the value of a variable in a text box. In a situation like this, one must certainly test the application with one or more incorrect values of the input variable. For example, while testing it is recommended that one use at least three values for X. In case the behavior of A is expected to be different for each integer within the range $0 \ldots 4$, then A should be tested separately for each value in the range as well as at least two values outside the range. However, one need not include incorrect values of variables in a test case if the variable is protected by the GUI against incorrect input. This situation is illustrated in Figure 2.6(c) where the subset labeled 1 in the set of incorrect values need not be included in the test data while values from subset labeled 2 must be included.

The discussion above leads to the conclusion that test design must take into account GUI design. In some cases, GUI design requirements could be dictated by test design. For example, the test design process might require the GUI to offer only valid values of input variables whenever possible. Certainly, GUI needs to be tested separately against such requirements. The boiler control example given earlier shows that the

number of tests can be reduced significantly if the GUI prevents invalid inputs from getting into the application under test.

Note that the tests derived in the boiler example make the assumption that the GUI has been correctly implemented and does prohibit incorrect values from entering the control software.

2.4 BOUNDARY-VALUE ANALYSIS

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes. For example, suppose that method M is required to compute a function f_1 when the condition $x \leq 0$ is satisfied by input x and function f_2 otherwise. However, M has an error due to which it computes f_1 for $x < 0$ and f_2 otherwise. Obviously, this fault is revealed, though not necessarily, when M is tested against $x = 0$ but not if the input test set is, for example, $\{-4, 7\}$ derived using equivalence partitioning. In this example, the value $x = 0$, lies at the boundary of the equivalence classes $x \leq 0$ and $x > 0$.

Boundary-value analysis is a test-selection technique that targets faults in applications at the boundaries of equivalence classes. While equivalence partitioning selects tests from within equivalence classes, boundary-value analysis focuses on tests at and near the boundaries of equivalence classes. Certainly, tests derived using either of the two techniques may overlap.

Test selection using boundary-value analysis is generally done in conjunction with equivalence partitioning. However, in addition to identifying boundaries using equivalence classes, it is also possible and recommended that boundaries be identified based on the relations among the input variables. Once the input domain has been identified, test selection using boundary-value analysis proceeds as follows:

1. Partition the input domain using unidimensional partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning.
2. Identify the boundaries for each partition. Boundaries may also be identified using special relationships among the inputs.
3. Select test data such that each boundary value occurs in at least one test input. A procedure to generate tests using boundary value-analysis and an illustrative example follows.

Example 2.11: This simple example is to illustrate the notion of boundaries and selection of tests at and near the boundaries. Consider a method *fP*, brief for *findPrice*, that takes two inputs *code*