4

Input Space Partitioning

In a very fundamental way, all testing is about choosing elements from the input space of the software being tested. The criteria presented previously can be viewed as defining ways to divide the input space according to the test requirements. The assumption is that any collection of values that satisfies the same test requirement will be "just as good." Input space partitioning takes that view in a much more direct way. The input *domain* is defined in terms of the possible values that the input parameters can have. The input parameters can be method parameters and global variables, objects representing current state, or user-level inputs to a program, depending on what kind of software artifact is being analyzed. The input domain is then partitioned into regions that are assumed to contain equally useful values from a testing perspective, and values are selected from each region.

This way of testing has several advantages. It is fairly easy to get started because it can be applied with no automation and very little training. The tester does not need to understand the implementation; everything is based on a description of the inputs. It is also simple to "tune" the technique to get more or fewer tests.

Consider an abstract partition q over some domain D. The partition q defines a set of equivalence classes, which we simply call *blocks*, B_q.[1] The blocks are pairwise disjoint, that is

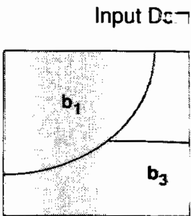
$$b_i \cap b_j = \emptyset, i \neq j; b_i, b_j \in B_q$$

and together the blocks cover the domain D, that is

$$\bigcup_{b \in B_q} b = D$$

This is illustrated in Figure 4.1. The input domain D is partitioned into three blocks, b_1, b_2, and b_3. The partition defines the values contained in each block and is usually designed from knowledge of what the software is supposed to do.

The idea in partition coverage is that any test in a block is as good as any other for testing. Several partitions are sometimes considered together, which, if not done carefully, leads to a combinatorial explosion of test cases.



Input Domain D

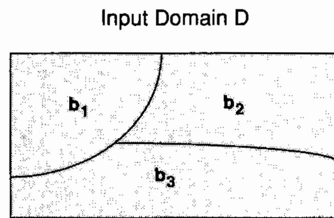


Figure 4.1. Partitioning of input domain D into three blocks.

A common way to apply input space partitioning is to start by considering the domain of each parameter separately, partitioning each domain's possible values into blocks, and then combining the variables for each parameter. Sometimes the parameters are considered completely independently, and sometimes they are considered in conjunction, usually by taking the semantics of the program into account. This process is called *input domain modeling* and the next section gives more details.

Each partition is usually based on some *characteristic* C of the program, the program's inputs, or the program's environment. Some possible characteristic examples are:

- Input X is null
- Order of file F (sorted, inverse sorted, arbitrary)
- Min separation distance of two aircraft

Each characteristic C allows the tester to define a partition. Formally, a partition *must* satisfy two properties:

1. The partition must cover the entire domain (completeness)
2. The blocks must not overlap (disjoint)

As an example, consider the characteristic "order of file F" mentioned above. This could be used to create the following (defective) partitioning:

- Order of file F
  - b_1 = Sorted in ascending order
  - b_2 = Sorted in descending order
  - b_3 = Arbitrary order

However, this is **not** a valid partitioning. Specifically, if the file is of length 0 or 1, then the file will belong in all three blocks. That is, the blocks are not disjoint. The easiest strategy to address this problem is to make sure that each characteristic addresses only one property. The problem above is that the notions of being sorted into ascending order and being sorted into descending order are lumped into the same characteristic. Splitting into two characteristics, namely sorted ascending and sorted descending, solves the problem. The result is the following (valid) partitioning of two characteristics.

- File F sorted ascending
  - b_1 = True
  - b_2 = False

- File F sorted descending
  - b_1 = True
  - b_2 = False

With these blocks, files of length 0 or 1 are in the True block for both characteristics.

The completeness and disjointness properties are formalized for pretty pragmatic reasons, and not just to be mathematically fashionable. Partitions that are not complete or disjoint probably reflect a lack of clarity in the rationale for the partition. In particular, if a partition actually encodes two or three rationales, the partition is likely to be quite messy, and it is also likely to violate either the completeness or the disjointness property (or both!). Identifying and correcting completeness or disjointness errors typically results in esthetically more pleasing partitions. Further, formally objectionable "partitions" cause unnecessary problems when generating tests, as discussed below. The rest of this chapter assumes that the partitions are both complete and disjoint.

4.1 INPUT DOMAIN MODELING

The first step in input domain modeling is identification of testable functions. Consider the TriTyp program from Chapter 3. TriTyp clearly has only one testable function with three parameters. The situation is more complex for Java class APIs. Each public method is typically a testable function that should be tested individually. However, the characteristics are often the same for several methods, so it helps to develop a common set of characteristics for the entire class and then develop specific tests for each method. Finally, large systems are certainly amenable to the input space partition approach, and such systems often supply complex functionality. Tools like UML use cases can be used to identify testable functions. Each use case is associated with a specific intended functionality of the system, so it is very likely that the use case designers have useful characteristics in mind that are relevant to developing test cases. For example, a "withdrawal" use case for an ATM identifies "withdrawing cash" as a testable function. Further, it suggests useful categories such as "Is Card Valid?" and "Relation of Withdrawal Policy to Withdrawal Request."

The second step is to identify all of the parameters that can affect the behavior of a given testable function. This step isn't particularly creative, but it is important to carry it out completely. In the simple case of testing a stateless method, the parameters are simply the formal parameters to the method. If the method has state, which is common in many object-oriented classes, then the state must be included as a parameter. For example, the insert(`Comparable obj`) method for a binary tree class behaves differently depending on whether or not `obj` is already in the tree. Hence, the current state of the tree needs to be explicitly identified as a parameter to the insert() method. In a slightly more complex example, a method find(String str) that finds the location of str in a file depends, obviously, on the particular file being searched. Hence, the test engineer explicitly identifies the file as a parameter to the find() method. Together, all of the parameters form the *input domain* of the function under test.

The third step, and the key creative engineering step, is modeling the input domain articulated in the prior step. An *input domain model* (*IDM*) represents the input space of the system under test in an abstract way. A test engineer describes

the structure of the input domain in terms of input *characteristics*. The test engineer creates a *partition* for each characteristic. The partition is a set of *blocks*, each of which contains a set of *values*. From the perspective of that particular characteristic, the values in each block are considered equivalent.

A test input is a tuple of values, one for each parameter. By definition, the test input belongs to exactly one block from each characteristic. Thus, if we have even a modest number of characteristics, the number of possible combinations may be infeasible. In particular, adding another characteristic with n blocks increases the number of combinations by a factor of n. Hence, controlling the total number of combinations is a key feature of any practical approach to input domain testing. In our view, this is the job of the coverage criteria, which we address in Section 4.2.

Different testers will come up with different models, depending on creativity and experience. These differences create a potential for variance in the quality of the resulting tests. The structured method to support input domain modeling presented in this chapter can decrease this variance and increase the overall quality of the IDM.

Once the IDM is built and values are identified, some combinations of the values may be invalid. The IDM must include information to help the tester identify and avoid or remove invalid sub-combinations. The model needs a way to represent these restrictions. Constraints are discussed further in Section 4.3.

The next section provides two different approaches to input domain modeling. The *interface-based* approach develops characteristics directly from input parameters to the program under test. The *functionality-based* approach develops characteristics from a functional or behavioral view of the program under test. The tester must choose which approach to use. Once the IDM is developed, several coverage criteria are available to decide which combinations of values to use to test the software. These are discussed in Section 4.2.

4.1.1 Interface-Based Input Domain Modeling

The interface-based approach considers each particular parameter in isolation. This approach is almost mechanical to follow, but the resulting tests are surprisingly good.

An obvious strength of using the interface-based approach is that it is easy to identify characteristics. The fact that each characteristic limits itself to a single parameter also makes it easy to translate the abstract tests into executable test cases.

A weakness of this approach is that not all the information available to the test engineer will be reflected in the interface domain model. This means that the IDM may be incomplete and hence additional characteristics are needed.

Another weakness is that some parts of the functionality may depend on combinations of specific values of several interface parameters. In the interface-based approach each parameter is analyzed in isolation with the effect that important subcombinations may be missed.

Consider the TriTyp program from Chapter 3. It has three integer parameters that represent the lengths of three sides of a triangle. In an interface-based IDM, Side1 will have a number of characteristics, as will Side2 and Side3. Since the three variables are all of the same type, the interface-based characteristics for each will

likely be identical. For example, since Side1 is an integer, and zero is often a special value for integers, Relation of Side1 to zero is a reasonable interface-based characteristic.

4.1.2 Functionality-Based Input Domain Modeling

The idea of the functionality-based approach is to identify characteristics that correspond to the intended functionality of the system under test rather than using the actual interface. This allows the tester to incorporate some semantics or domain knowledge into the IDM.

Some members of the community believe that a functionality-based approach yields better test cases than the interface-based approach because the input domain models include more semantic information. Transferring more semantic information from the specification to the IDM makes it more likely to generate expected results for the test cases, an important goal.

Another important strength of the functionality-based approach is that the requirements are available before the software is implemented. This means that input domain modeling and test case generation can start early in development.

In the functionality-based approach, identifying characteristics and values may be far from trivial. If the system is large and complex, or the specifications are informal and incomplete, it can be very hard to design reasonable characteristics. The next section gives practical suggestions for designing characteristics.

The functionality-based approach also makes it harder to generate tests. The characteristics of the IDM often do not map to single parameters of the software interface. Translating the values into executable test cases is harder because constraints of a single IDM characteristic may affect multiple parameters in the interface.

Returning to the TriTyp program from Chapter 3, a functionality-based approach will recognize that instead of simply three integers, the input to the method is a triangle. This leads to the characteristic of a triangle, which can be partitioned into different types of triangles (as discussed below).

4.1.3 Identifying Characteristics

Identifying characteristics in an interface-based approach is simple. There is a mechanical translation from the parameters to characteristics. Developing a functionality-based IDM is more challenging.

Preconditions are excellent sources for functionality-based characteristics. They may be explicit or encoded in the software as exceptional behavior. Preconditions explicitly separate defined (or normal) behavior from undefined (or exceptional) behavior. For example, if a method choose() is supposed to select a value, it needs a precondition that a value must be available to select. A characteristic may be whether the value is available or not.

Postconditions are also good sources for characteristics. In the case of TriTyp, the different kinds of triangles are based on the postcondition of the method.

The test engineer should also look for other relationships between variables. These may be explicit or implicit. For example, a curious test engineer given a

method m() with two object parameters x and y might wonder what happens if x and y point to the same object (aliasing), or to logically equal objects.

Another possible idea is to check for missing factors, that is, factors that may impact the execution but do not have an associated IDM parameter.

It is usually better to have many characteristics with few blocks than the reverse. It is also true that characteristics with small numbers of blocks are more likely to satisfy the disjointness and completeness properties.

Generally, it is preferable for the test engineer to use specifications or other documentation instead of program code to develop characteristics. The idea is that the tester should apply input space partitioning by using *domain knowledge* about the problem, not the implementation. However, in practice, the code may be all that is available. Overall, the more semantic information the test engineer can incorporate into characteristics, the better the resulting test set is likely to be.

The two approaches generally result in different IDM characteristics. The following method illustrates this difference:

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//   else returns true if element is in the list, false otherwise
```

If the interface-based approach is used, the IDM will have characteristics for list and characteristics for element. For example, here are two interface-based characteristics for list, including blocks and values, which are discussed in detail in the next section:

- list is null
  - b_1 = True
  - b_2 = False
- list is empty
  - b_1 = True
  - b_2 = False

The functionality-based approach results in more complex IDM characteristics. As mentioned earlier, the functionality-based approach requires more thinking on the part of the test engineer, but can result in better tests. Two possibilities for the example are listed below, again including blocks and values.

- number of occurrences of element in list
  - $b_1 = 0$
  - $b_2 = 1$
  - b_3 = More than 1
- element occurs first in list
  - b_1 = True
  - b_2 = False

4.1.4 Choosing Blocks and Values

After choosing characteristics, the test engineer partitions the domains of the characteristics into sets of values called *blocks*. A key issue in any partition approach is how partitions should be identified and how representative values should be selected from each block. This is another creative design step that allows the tester to tune the test process. More blocks will result in more tests, requiring more resources but possibly finding more faults. Fewer blocks will result in fewer tests, saving resources but possibly reducing test effectiveness. Several general strategies for identifying values are as follows:

- **Valid values**: Include at least one group of valid values.
- **Sub-partition**: A range of valid values can often be partitioned into sub-partitions, such that each sub-partition exercises a somewhat different part of the functionality.
- **Boundaries**: Values at or close to boundaries often cause problems.
- **Normal use**: If the operational profile focuses heavily on "normal use," the failure rate depends on values that are not boundary conditions.
- **Invalid values**: Include at least one group of invalid values.
- **Balance**: From a cost perspective, it may be cheap or even free to add more blocks to characteristics that have fewer blocks. In Section 4.2, we will see that the number of tests sometimes depends on the characteristic with the maximum number of blocks.
- **Missing partitions**: Check that the union of all blocks of a characteristic completely covers the input space of that characteristic.
- **Overlapping partitions**: Check that no value belongs to more than one block.

Special values can often be used. Consider a Java reference variable; null is typically a special case that needs to be treated differently from non null values. If the reference is to a container structure such as a Set or List, then whether the container is empty or not is often a useful characteristic.

Consider the TriTyp program from Chapter 3. It has three integer parameters that represent the lengths of three sides of a triangle. One common partitioning for an integer variable considers the relation of the variable's value to some special value in the testable function's domain, such as zero.

Table 4.1 shows a partitioning for the interface-based IDM for the TriTyp program. It has three characteristics, q_1, q_2, and q_3.

The first row in the table should be read as "Block $q_1.b_1$ is that Side 1 is greater than zero," "Block $q_1.b_2$ is that Side 1 is equal to zero," and "Block $q_1.b_3$ is that Side 1 is less than zero."

Table 4.1. First partitioning of TriTyp's inputs (interface-based)

Partition	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	*greater than 0*	*equal to 0*	*less than 0*
q_2 = "Relation of Side 2 to 0"	*greater than 0*	*equal to 0*	*less than 0*
q_3 = "Relation of Side 3 to 0"	*greater than 0*	*equal to 0*	*less than 0*

Table 4.2. Second partitioning of TriTyp's inputs (interface-based).

Partition	b_1	b_2	b_3	b_4
q_1 = "Length of Side 1"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_2 = "Length of Side 2"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_3 = "Length of Side 3"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*

Consider the partition q_1 for Side 1. If one value is chosen from each block, the result is three tests. For example, we might choose Side 1 to have the value 7 in test 1, 0 in test 2, and −3 in test 3. Of course, we also need values for Side 2 and Side 3 of the triangle to complete the test case values. Notice that some of the blocks represent valid triangles and some represent invalid triangles. For example, no valid triangle can have a side of negative length.

It is easy to refine this categorization to get more fine grained testing if the budget allows. For example, more blocks can be created by separating inputs with value 1. This decision leads to a partitioning with four blocks, as shown in Table 4.2.

Notice that if the value for Side 1 were floating point rather than integer, the second categorization would **not** yield valid partitions. None of the blocks would include values between 0 and 1 (noninclusive), so the blocks would not cover the domain (not be complete). However, the domain D contains integers so the partitions are valid.

While partitioning, it is often useful for the tester to identify candidate values for each block to be used in testing. The reason to identify values now is that choosing specific values can help the test engineer think more concretely about the predicates that describe each block. While these values may not prove sufficient when refining test requirements to test cases, they do form a good starting point. Table 4.3 shows values that can satisfy the second partitioning.

The above partitioning is interface based and only uses syntactic information about the program (it has three integer inputs). A functionality-based approach can use the semantic information of the traditional geometric classification of triangles, as shown in Table 4.4.

Of course, the tester has to know what makes a triangle scalene, equilateral, isosceles, and invalid to choose possible values (this may be simple middle school geometry, but many of us have probably forgotten). An equilateral triangle is one in which all sides are the same length. An isosceles triangle is one in which at least two sides are the same length. A scalene triangle is any other valid triangle. This brings up a subtle problem, Table 4.4 does **not** form a valid partitioning. An equilateral triangle is also isosceles, thus we must first correct the partitions, as shown in Table 4.5.

Table 4.3. Possible values for blocks in the second partitioning in Table 4.2

Param	b_1	b_2	b_3	b_4
Side 1	2	1	0	−1
Side 2	2	1	0	−1
Side 3	2	1	0	−1

Table 4.4. Geometric partitioning of TriTyp's inputs (functionality-based)

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	*scalene*	*isosceles*	*equilateral*	*invalid*

Now values for Table 4.5 can be chosen as shown in Table 4.6. The triplets represent the three sides of the triangle.

A different approach to the equilateral/isosceles problem above is to break the characteristic Geometric Partitioning into four separate characteristics, namely Scalene, Isosceles, Equilateral, and Valid. The partition for each of these characteristics is boolean, and the fact that choosing Equilateral = true also means choosing Isosceles = true is then simply a constraint. Such an approach is highly recommended, and it invariably satisfies the disjointness and completeness properties.

4.1.5 Using More than One Input Domain Model

For a complex program it might be better to have several small IDMs than one large. This approach allows for a divide-and-conquer strategy when modeling characteristics and blocks. Another advantage with multiple IDMs for the same software is that it allows varying levels of coverage.

For instance, one IDM may contain only valid values and another IDM may contain invalid values to focus on error handling. The valid value IDM may be covered using a higher level of coverage. The invalid value IDM may use a lower level of coverage.

Multiple IDMs may be overlapping as long as the test cases generated make sense. However, overlapping IDMs are likely to have more constraints.

4.1.6 Checking the Input Domain Model

It is important to check the input domain model. In terms of characteristics, the test engineer should ask whether there is any information about how the function behaves that is not incorporated in some characteristics. This is necessarily an informal process.

The tester should also explicitly check each characteristic for the completeness and disjointness properties. The purpose of this check is to make sure that, for each characteristic, not only do the blocks cover the complete input space, but selecting a particular block implies excluding all other blocks in that characteristic.

If multiple IDMs are used, completeness should be relative to the portion of the input domain that is modeled in each IDM. When the tester is satisfied with the

Table 4.5. Correct geometric partitioning of TriTyp's inputs (functionality-based)

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	*scalene*	*isosceles, not equilateral*	*equilateral*	*invalid*

Table 4.6. Possible values for blocks in geometric partitioning in Table 4.5.

Param	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

characteristics and their blocks, it is time to choose which combinations of values to test with and identify constraints among the blocks.

EXERCISES

Section 4.1.

1. Answer the following questions for the method search() below:

```
public static int search (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//   else if element is in the list, return an index
//   of element in the list; else return -1
//   for example, search ([3,3,1], 3) = either 0 or 1
//      search ([1,7,5], 2) = -1
```

Base your answer on the following characteristic partitioning:

```
Characteristic: Location of element in list
   Block 1: element is first entry in list
   Block 2: element is last entry in list
   Block 3: element is in some position other than first or last
```

(a) "Location of element in list" fails the disjointness property. Give an example that illustrates this.
(b) "Location of element in list" fails the completeness property. Give an example that illustrates this.
(c) Supply one or more new partitions that capture the intent of "Location of e in list" but do not suffer from completeness or disjointness problems.

2. Derive input space partitioning tests for the **GenericStack** class with the following method signatures:
   - public GenericStack ();
   - public void Push (Object X);
   - public Object Pop ();
   - public boolean IsEmt ();

   Assume the usual semantics for the stack. Try to keep your partitioning simple, choose a small number of partitions and blocks.
   (a) Define characteristics of inputs
   (b) Partition the characteristics into blocks
   (c) Define values for the blocks