11

The PSP Design Templates

At this point in learning the PSP, you are probably making 30% to 40% fewer coding mistakes than you did at the beginning of the course. Figure 11.1 shows that, for many developers, this improvement happens after Program 3. That is when they become aware of the number and kinds of mistakes they make during program development. This awareness did not just happen, however; it resulted from the defect analysis study the students in this class made after completing their third program. After seeing how many mistakes they make, most developers take more care during coding. This substantially reduces the number of defects they inject. These data are for the original PSP course with ten exercises. Because PSP courses based on this current text may have a different number of exercises, the defect analysis is generally made with report R4, which comes after three to six of the programs. That is when the sharp drop in defect injection should occur.

Avoiding design defects requires more than just awareness and care. It requires better designs. The PSP addresses design issues from a defect prevention point of view. The objective is better and more precise designs. This chapter describes how the PSP design templates address this need and how to use them.

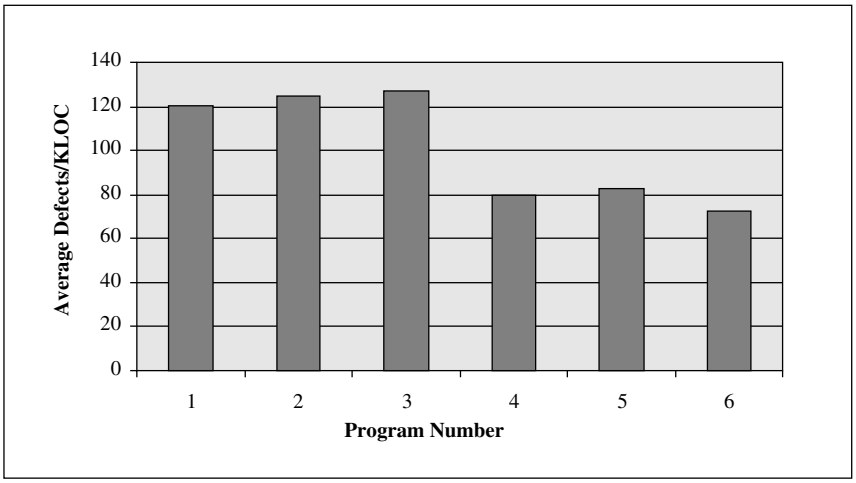

FIGURE 11.1 AVERAGE DEFECTS/KLOC (810 DEVELOPERS, PSP PROGRAMS 1 THROUGH 6)

11.1 Design Representation

The Unified Modeling Language (UML) provides a comprehensive framework for representing designs (Booch et al. 1999). Rather than require that all developers who use the PSP also learn UML, this book provides a simplified set of generalized design templates for precisely recording relatively simple designs. These PSP design templates complement UML and other design methods by providing a simple summary format for documenting small program designs.

The lack of a precise design is the source of many implementation errors. If you reviewed your design defect data, you would likely find that many of these defects were caused by simple errors. The reasons for these errors would vary, but a frequent cause would likely be the lack of a properly specified design. A principal objective of a software design is to provide a concise and precise statement of exactly what the program is to do and how it is to do it. Doing this requires three things: producing a sound design, recording all of the required design information, and recording it in a precise and understandable way. Then, of course, you must faithfully implement that design.

Precise Logic Notation

Once you have created the design, you must analyze it for correctness, completeness, and consistency. This requires that the design be precisely documented. With spoken languages and pictures, it is often hard to be sufficiently precise in describing exactly what a program is to do and how it is to be constructed. Both language and pictures should be used to communicate designs, but the design must be recorded in a concise and precise notation that completely and accurately defines it for the implementation work to follow. This notation should meet the following criteria:

- It should be capable of precisely and completely representing the design.
- It must be understood and used by the people who use the design.
- It should help you to verify that the design does precisely what you intend.

This chapter describes how to represent designs with a set of templates and a precise logic notation. This notation is based on propositional logic and uses a simplified version of the Z notation that is easy to learn and to use (Jacky 1996). If you are not familiar with propositional logic, Table 11.1 provides a brief summary of the key terms, and the following paragraphs present a simple example of their use. Additional examples of this notation are given with some of the template descriptions in this and the next chapter.

Although you may choose not to use this notation after PSP training, you should use it for documenting the designs of the remaining PSP exercise programs. The design method is your servant. If, after using it in the PSP course, it does not help you to produce clearer designs, it is probably not right for you, at least not at this time.

Using the Logic Notation

Although this notation may seem new and unfamiliar, it is easy to use and very concise and expressive. Suppose, for example, you had a large volume of PSP data that you wanted to analyze. Suppose also that in order to do this analysis, you wanted a list of the names of all the developers of the PSP programs. One way to define a NameList that contains the names of all the developers of the PSPData would be as follows:

$$\forall \text{ dname} : \text{DeveloperName} \mid \text{dname} \in \text{NameList} \bullet \exists \text{ pdata} : \text{PSPData} \bullet \text{pdata.Name} = \text{dname}$$

This statement says that for all developer names (dname) in the list DeveloperName, where dname is a member of NameList, there exists a pdata in PSPData such that the name field of pdata equals dname.

TABLE 11.1 LOGIC NOTATION

Symbol	Function	Example
:=	assignment	A := B – Assign the value B to variable A.
!	not	!A
=	equal	A = B
≠	not equal	A ≠ B
>=	greater than or equal	A >= B
<=	less than or equal	A <= B
>	greater than	A > B
<	less than	A < B
∈	member of	d ∈ CData – d is a member of CData.
∉	not a member of	d ∉ CData – d is not a member of CData.
true	true	
false	false	
•	it is true that	If A = x • B = C – If A = x, it is true that B = C.
∀	for all	∀ a : A • a ∈ CData – Every member a in set A is a member of CData.
∃	there exists	∃ a : A • a ∈ CData – At least one member a of set A is a member of CData.
∧	intersection, logical and	A ∧ B – A and B
∨	union, logical or	A ∨ B – A or B
A → B	guarded command	If A = x → B := C – A = x causes B to be set equal to C.
;	sequence	A; B; C – A followed by B followed by C

In this case, it would have been easier to just say you want to ensure that all names in NameList also have at least one entry in PSPData. However, in many cases the relationships are much more complex, and the language description would become so convoluted that it would be almost impossible to understand. After a little practice using this notation with a few simple cases, you will likely get the hang of it and find it very convenient.

11.2 The Design Templates

The design representation described in the remainder of this chapter uses four templates to represent a program's design. These templates are used to describe the essential properties and structure of module-size programs. The PSP templates also help to minimize duplication. Each item is recorded in only one place and its location is referenced when needed. This saves time, reduces the likelihood of error, and provides reliable references. A good design should have minimum redundancy.

The PSP templates are designed to provide a complete and precise design representation for the basic program building blocks of software products and systems. Although many of the large-scale properties of software systems are not covered by the templates, these larger-scale properties must ultimately be reflected in the design and code of one or more of the program's modules. These templates help to ensure that these larger properties are completely and properly implemented at the module level. As with all PSP elements, you should modify these templates to suit your needs, and augment them with whatever other design tool, method, or graphical aid you find helpful.

The elements of a complete design can be visualized with the two-dimensional specification structure shown in Figure 11.2 (de Champeaux et al. 1993). Here, the elements of the module's design are divided into four categories:

1. **External-static:** This category defines the static relationships of this part to other parts or system elements. Examples are the call-return behavior and the inheritance hierarchy.

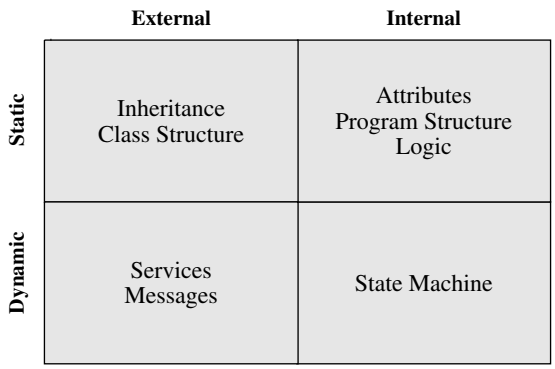

FIGURE 11.2 DESIGN SPECIFICATION STRUCTURE

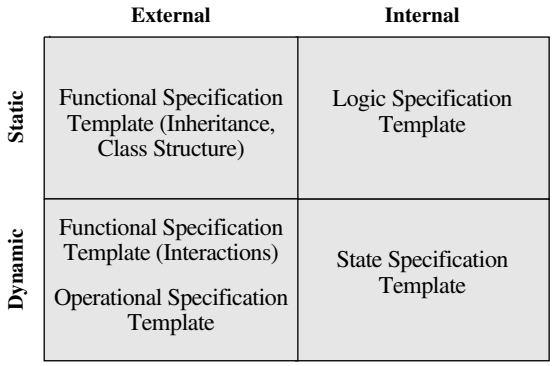

FIGURE 11.3 PSP TEMPLATE STRUCTURE

2. **External-dynamic:** This category defines the interactions of this part with other parts or system elements. Examples would be user scenarios, system- or regression-testing scripts, or real-time interrupt-response behavior.
3. **Internal-static:** This category contains a static description of a module or part, such as its detailed logical structure.
4. **Internal-dynamic:** This category defines the part's dynamic characteristics. Examples of internal-dynamic behavior are state machines, response-time specifications, and interrupt-handling performance.

With the four templates described in the following sections, you can precisely specify the design of just about any small program. The four PSP design templates roughly correspond to de Champeaux's four quadrants as shown in Figure 11.3. These templates and the information that they contain are described in the following sections, together with examples of how they are used.

11.3 The Operational Specification Template (OST)

The example Operational Specification Template (OST) and the OST Instructions are shown in Tables 11.2 and 11.3. The OST is a simplified form of use case that is used to describe a program's operational behavior (Schneider and Winters 1998). The OST helps you to visualize how the program is supposed to react under various usage scenarios. When faced with a design decision that involves an actor, produce

a trial scenario to see how each action would appear to the actor. This will help you to visualize the program's behavior and to make proper design choices. Although no set of scenarios can cover all usage combinations for even a moderately complex system, include examples of the key functions and exception conditions. With careful planning, you can often cover most important use cases with just a few scenarios. In addition to aiding in program design, these scenarios also provide a sound basis for test design and development. If you don't construct and test such scenarios, you are likely to overlook some key program actions.

The OST provides a space for the scenario number at the top. Numbering the scenarios helps when you use several scenarios for one program. In the next space, *User Objective,* briefly describe the user's expected purpose for the scenario. It is often hard for program designers to see the product from the user's perspective. By writing a brief statement of what the user is trying to accomplish with the scenario, you can better visualize the user's viewpoint. Next, enter a brief statement of your objective in constructing the scenario in the *Scenario Objective* space. Examples would be to illustrate error behavior or to define overflow exception handling. In the *Source* column, enter the originator of each action; in *Step*, enter the action

TABLE 11.2 EXAMPLE OPERATIONAL SPECIFICATION TEMPLATE (OST)

Student J. Developer Date 10/26

Program LogIn Program # _____

Instructor Humphrey Language C++

Scenario Number	1	**User Objective**	To log onto the system
Scenario Objective		To illustrate normal LogIn operation	
Source	**Step**	**Action**	**Comments**
User	1	Call the system	
System	2	Request UserId	Check for timeout
User	3	Supplies UserId	Check for data errors
System	4	Check UserId	Check the user's ID
CheckId	5	UserId is proper	Also check error case
System	6	Request UserPW	Check for timeout
User	7	Supplies UserPW	Check for data errors
System	8	CheckPW	Check the user's PW
CheckPW	9	PW is proper	Also check error case
System	10	LogInUser	Log user onto system, stop

TABLE 11.3 OPERATIONAL SPECIFICATION TEMPLATE (OST) INSTRUCTIONS

Purpose	• To hold descriptions of the likely operational scenarios followed during program use • To ensure that all significant usage issues are considered during program design • To specify test scenarios
General	• Use this template for complete programs, subsystems, or systems. • Group multiple small scenarios on a single template, as long as they are clearly distinguished and have related objectives. • List the major scenarios and reference other exception, error, or special cases under comments. • Use this template to document the operational specifications during planning, design, test development, implementation, and test. • After implementation and testing, update the template to reflect the actual implemented product.
Header	• Enter your name and the date. • Enter the program name and number. • Enter the instructor's name and the programming language you are using.
Scenario Number	Where several scenarios are involved, reference numbers are needed.
User Objective	List the users' likely purpose for the scenario, for example, to log onto the system or to handle an error condition.
Scenario Objective	List the designer's purpose for the scenario, for example, to define common user errors or to detail a test scenario.
Source	• Enter the source of the scenario action. • Example sources could be user, program, and system.
Step	Provide sequence numbers for the scenario steps. These facilitate reviews and inspections.
Action	Describe the action taken, such as • Enter incorrect mode selection. • Provide error message.
Comments	List significant information relating to the action, such as • User enters an incorrect value. • An error is possible with this action.

sequence number; in *Action*, describe the action taken; and in *Comments*, note results, exception conditions, or other useful information.

The OST example in Table 11.2 is for a simple LogIn routine. It shows the user and system actions to verify the user's ID and password (PW). OSTs should also be completed to define how the error conditions shown in OST #1 are handled: timeout errors where the user fails to respond within a specified time, data errors where the password or user ID strings have an unacceptable format, or incorrect IDs or passwords. An unacceptable string might be excessively long or it could contain improper characters. Such strings are the source of over half of the

security vulnerabilities on the Internet. All input character strings should be checked for security problems before being passed to the LogIn class or any of its methods. Such checks should be made for every publicly exposed input on any Internet-connected system.

One OST should describe the program's normal flow, noting all possible error or other abnormal cases. OSTs should also define the tests needed to verify the program's normal and abnormal operation. Complete this portion of test development during program design. When you design the tests for a program while you are designing it, you will often find design problems. For larger systems, it is also desirable to define the external behavior of the lower-level subsystems and components with OSTs.

11.4 The Functional Specification Template (FST)

The Functional Specification Template (FST) provides a simple way to document much of the material in a UML class diagram. An example FST and its instructions are shown in Tables 11.4 and 11.5. The FST describes a part, including its methods, its relationships, and its constraints. The part could be a class, a program module, or even a large program or system. As shown in Table 11.4, the part or class name is listed at the top, together with the classes or other parts from which it directly descends. The attributes are listed next, with their declarations and descriptions. The third section lists the part's methods or items, with their declarations, descriptions, and returns.

The *Attributes* section is also where you list the relationships of this class or part with other classes or parts. A suggested simple attribute notation is as follows (Rumbaugh et al. 1999):

Name[multiplicity] : type = initial-value {constraint}

where

- **Name** is the name of the attribute.
- **Multiplicity** indicates the number or range of occurrences.
- **Type** is a standard type or class, with normal font for a standard type and italics for a class.
- **Initial value** is the value assigned when the attribute is created.
- **Constraint** defines any constraints on the class.

For example, a UserClass with 0 to *n* instances might be listed as UserClass[0..n] : *UserClass.* The italicized type indicates that this attribute concerns a

TABLE 11.4 EXAMPLE FUNCTIONAL SPECIFICATION TEMPLATE (FST)

Student	J. Developer	Date	10/26
Program	LogIn	Program #	
Instructor	Humphrey	Language	C++

Class Name	LogIn
Parent Class	

Attributes

Declaration	**Description**
MaxTime: Integer, minutes	Maximum time-out minutes - adjustable
n: Integer	Count of user ID and password errors
nMax: Integer	Maximum of n error count
ValidIdSet	Set of valid user IDs with passwords

Items

Declaration	**Description**
Void LogIn.Start(n: Int)	Initialize system Handle LogIn transaction, error count, and time-outs
Int LogIn.GetId(ID: String)	Get UserId and check it for problems. Return true for an acceptable string, false for timeout or an unacceptable string
Int LogIn.CheckId(ID: String)	UserId $\in$ ValidIdSet $\rightarrow$ Valid ID UserId $\notin$ ValidIdSet $\rightarrow$!Valid ID
Int LogIn.GetPW(PW: String)	Get password and check for problems. Return true for an acceptable string, false for timeout or an unacceptable string
Int LogIn.CheckPW(PW: String)	PW = UserId.PW $\rightarrow$ Valid PW PW $\neq$ UserId.PW $\rightarrow$!Valid PW
Void LogIn.LogInUser(ID: String, n: Int)	n >= nMax $\rightarrow$ Reject user, deactivate ID Valid ID $\wedge$ Valid PW $\rightarrow$ Log in user

class, rather than a standard type. The multiplicity could be [n], [n..i], [n..*], [*], etc., for example. Example attribute constraints would be n : integer {even} for an integer *n* that must have only even values. Similarly, you could list UserClass-Member : *UserClass* {UserClassMember.ID $\in$ UserIDList}, specifying that the members of the UserClass have IDs that are in the UserIDList. An example of some attribute categories is shown in Table 11.6.

The FST also defines the program's items, their calls, and their returns. For example, CheckId returns true when ID is a member of the ValidIdSet, so the ID is valid. If the return is false, the program should still request a PW even though the ID is incorrect. It is good security practice to get both the ID and PW before telling the user that one of them is incorrect. This makes it harder for intruders to identify valid user IDs and passwords.

The FST provides a concise description of the class's external appearance and actions. It also provides a central description of the class's attributes. However, it does not describe the internal workings of the class. The Logic Specification Template (LST) and the State Specification Template (SST) do that.

In the example FST, the Start item initializes the system and manages the LogIn activities. However, until you produce the SST and LST, you will not know precisely how the program works. Though it might seem desirable to include enough of a description on the FST to explain its internal behavior, that would duplicate the SST and LST. Then they would either be unnecessary or you would have to make duplicate updates whenever you changed the design. To the extent that you can, try to keep the OST and FST focused on what the program does, rather than how it works.

TABLE 11.5 FUNCTIONAL SPECIFICATION TEMPLATE (FST) INSTRUCTIONS

Purpose	• To hold a part's functional specifications • To describe classes, program modules, or entire programs
General	• Use this template for complete programs, subsystems, or systems. • Use this template to document the functional specifications during planning, design, test development, implementation, and test. • After implementation and testing, update the template to reflect the actual implemented product.
Header	• Enter your name and the date. • Enter the program name and number. • Enter the instructor's name and the programming language you are using.
Class Name	• Enter the part or class name and the classes from which it directly inherits. • List the class names starting with the most immediate. • Where practical, list the full inheritance hierarchy.
Attributes	• Provide the declaration and description for each global or externally visible variable or parameter, together with any constraints. • List pertinent relationships of this part with other parts together with the multiplicity and constraints.
Items	• Provide the declaration and description for each item. • Precisely describe the conditions that govern each item's return values. • Describe any initialization or other key item responsibilities.
Example Items	An item could be a class method, procedure, function, or database query, for example.

TABLE 11.6 EXAMPLE FST ATTRIBUTES

Class Name	User

Attributes	
Declaration	**Description**
UserID : string {unique}	A unique user ID
UserPassword : string	User's password
MemberOf[0..1] : *Group*	The group class this user is a member of

11.5 The State Specification Template (SST)

A state machine's behavior is defined by its current inputs and the system's state. The State Specification Template provides a simple way to precisely describe a part's state behavior. Programs with multiple states are called **state machines** and their behavior can be enormously complex. A typical state machine is a thermostat, automobile cruise control, or stopwatch. The actions of a state machine depend both on the input and on the current system state. Any system is a state machine if it has memory and its behavior depends on the contents of its memory.

In the programming world, state machines can be helpful in designing counters, sorts, interrupt handlers, security systems, web sites, and many other products. For example, every page on a web site can be viewed as a state, and the transitions among the web pages can be represented as state transitions. The growing importance of Internet security now requires that the transitions among web pages be controlled. It also means that the security authorization state of the user must be considered and authenticated on every web page. Without a clear understanding of states and state behavior, designing an effective security management system can be difficult, if not impossible.

In the PSP, state-machine behavior is described with the aid of the State Specification Template (SST). The state machine for the LogIn function is shown in Figure 11.4. The SST and its instructions are shown in Tables 11.7 and 11.8. This is the LogIn class described in the OST and FST templates in Tables 11.2 and 11.4.

A simple scenario illustrates how this state machine works. From Start, the system transitions to the CheckID state, sets n := 0, Fail := false, ID and PW to !Valid, and requests an ID from the user. From the CheckID state, whether the ID is valid or not, the system transitions to the CheckPW state and requests a PW from the user. If PW and ID are valid and Fail is false, the user is logged in. If PW

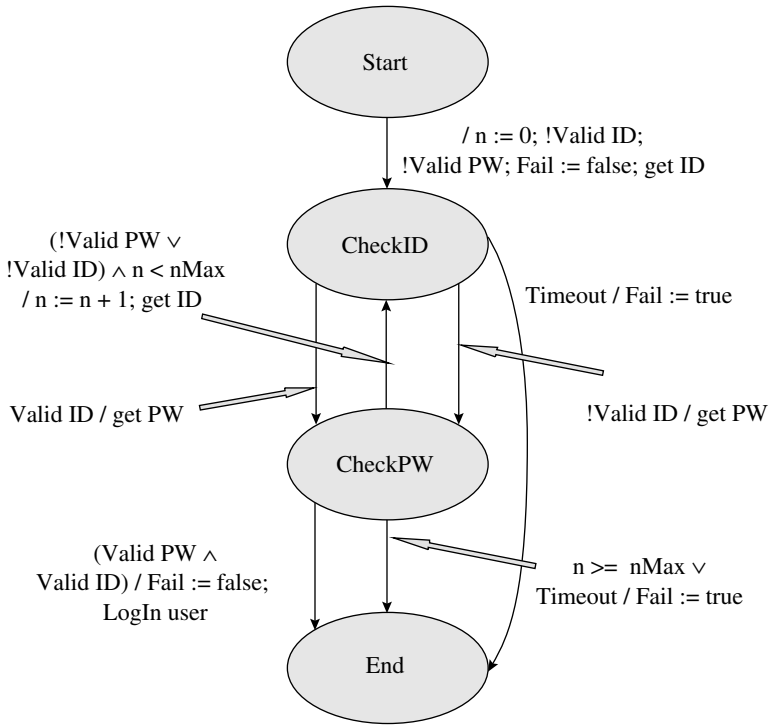

FIGURE 11.4 LOGIN STATE-MACHINE DIAGRAM

or ID is not valid, the system returns to CheckID, requesting an ID. The error count n is increased by 1, and if it reaches nMax, the system exists with Fail := true. Although this is a simple state machine, many are too complex to precisely describe in a figure. This complexity can easily be captured on the SST, as shown in Table 11.7. Note, however, that no design should be considered correct until it is verified.

The SST example in Table 11.7 shows all of the LogIn state-machine states and the transitions among them. These states are all listed at the top of the form followed by a section for each state and its next states. Each state should be briefly described, including the conditions that cause transition to each next state and the transition actions. Only the possible next states need be shown unless some impossible transition needs verification. If so, list the impossible transition and why it is impossible. It is then relatively easy to check these impossible conditions in a design review, inspection, or test.

TABLE 11.7 EXAMPLE STATE SPECIFICATION TEMPLATE (SST)

Student J. Developer Date 10/27

Program LogIn Program # _____

Instructor Humphrey Language C++

State Name	**Description**	
Start	Start condition for system	
CheckD	The state of the system after a user ID is requested	
CheckPW	The state of the system after a user password is requested	
End	The final state: LogIn either logs in or cuts off the user.	
Function/Parameter	**Description**	
ID	User identification: ID is Valid or !Valid	
PW	User password: PW is Valid or !Valid	
n	Integer count of ID and password errors	
nMax	Maximum value of ID and password errors: n >= nMax is rejected.	
Fail	Error count or timeout indicator: Fail = true is failure, Fail = false is ok.	
States/Next States	**Transition Condition**	**Action**
Start		
Start	No transitions from Start to Start	
CheckID	True	Get ID, n := 0; ID and PW !Valid
CheckPW	No transitions from Start to CheckPW	
End	No transitions from Start to End	
CheckID		
Start	No transitions from CheckID to Start	
CheckID	No transitions from CheckID to CheckID	
CheckPW	Valid ID	Get password
CheckPW	!Valid ID	Get password
End	Timeout	Fail := true
CheckPW		
Start	No transitions from CheckPW to Start	
CheckID	(!Valid PW ∨ !Valid ID) ∧ n < nMax ∧ !Timeout	Get ID, n := n + 1
CheckPW	No transitions from CheckPW to CheckPW	
End	Valid PW ∧ Valid ID	Fail := false, login user
End	n >= nMax ∨ Timeout	Fail := true, cut off user
End		
	No transitions from End to any state	

TABLE 11.8 STATE SPECIFICATION TEMPLATE INSTRUCTIONS

Purpose	• To hold the state and state transition specifications for a system, class, or program • To support state-machine analysis during design, design reviews, and design inspections
General	• This form shows each system, program, or routine state, the attributes of that state, and the transition conditions among the states. • Use this template to document the state specifications during planning, design, test development, implementation, and test. • After implementation and testing, update the template to reflect the actual implemented product.
Header	• Enter your name and the date. • Enter the program name and number. • Enter the instructor's name and the programming language you are using.
State Name	• Name all of the program's states. • Also enter each state name in the header space at the top of each *States/Next States* section of the template.
State Name Description	• Describe each state and any parameter values that characterize it. • For example, if a state is described by SetSize=10 and SetPosition=3, list SetSize=10 and SetPosition=3.
Function/ Parameter	• List the principal functions and parameters. • Include all key variables or methods used to define state transitions or actions.
Function/ Parameter Description	• For each function, provide its declaration, parameters, and returns. • For each parameter, define its type and significant values.
Next State	• For each state, list the names of all possible next states. • Include the state itself.
Transition Condition	List the conditions for transition to each next state. • Use a mathematical or otherwise precise notation. • If the transition is impossible, list "impossible," with a note saying why.
Action	List the actions taken with each state transition.

11.6 The Logic Specification Template (LST)

An example of the Logic Specification Template (LST) is shown in Table 11.9. It shows how to use a simple programming design language (PDL) to describe the internal logic of the LogIn class. PDLs are often called **pseudocode**.

The LST holds a pseudocode description of a program. Its objective is to clearly and concisely explain what the program is to do and how. Pseudocode

TABLE 11.9 EXAMPLE LOGIC SPECIFICATION TEMPLATE (LST)

Student	J. Developer	Date	10/27
Program	LogIn	Program #	
Instructor	Humphrey	Language	C++

Design References
- Operational Specification – page 231
- Functional Specification – page 234
- State Specification – page 238

Parameters
- n : the error counter, maximum value nMax
- ID : Boolean indicator of ID Valid and ID !Valid
- PW : Boolean indicator of PW Valid and PW !Valid
- Fail: Boolean indicator of failure condition, end session

Log a user onto the system.
Start by initializing the n error counter, set ID: = !Valid, PW := !Valid, and Fail := false.
Get user ID.
Repeat the main loop until a valid ID and password or Fail.
Check ID for validity. {CheckID state}
If no ID response in MaxTime, set Fail := true.
Get password and check for validity. {CheckPW state}
If no password response in MaxTime, set Fail := true.
If PW !Valid or ID !Valid, step the n counter.
If n exceeds nMax, set Fail := true.
Until ID and PW Valid or Fail = true.
Otherwise, repeat the main loop.
If Fail = true, cut off user, otherwise, log in the user. {End state}

should be written in ordinary human language and, to the extent practical, it should not use programming language constructs (McConnell 1993). When developers use programming expressions in their pseudocode, they often end up writing the program in pseudocode instead of the programming language. This is programming and not design.

In writing the pseudocode, don't bother with design details that would be obvious to an experienced implementer. Appropriate pseudocode statements might be "initialize the class" or "repeat main loop until ID and password valid." However, before completing the design, make sure the pseudocode is sufficiently detailed that the implementation is obvious without further design.

Pseudocode can be written in stages, much like the outline of a paper or a book. Start with an overall statement of what the program does. Then, as you refine the design, expand this initial statement to include added pseudocode detail. A common approach is to use your development environment to write the pseudocode as comments and then, during implementation, insert the code in the appropriate places to implement the pseudocode. Assuming that the pseudocode is clear and sufficiently detailed, it will provide the comments needed to produce a clear, understandable, and correct program.

11.7 A State-Machine Design Example

Although designing a state machine may seem simple, it can also be tricky. The problem is devising just the right states to completely and efficiently do the job. Because there are many possible state machines for any given problem, it is easy to produce a large and unnecessarily complex structure. Such designs are error-prone and expensive, however, so it is worthwhile taking enough time to produce a good design. A few minutes of design and design verification can save hours in implementation and testing. It will also produce a neater and higher-quality product.

LogIn State-Machine Example

A common program design approach is to think in scenarios. Although this is effective for applications, it can lead to overly complex state machines. For example, in designing the LogIn state machine in this way, you might begin with a Start state and then get and verify the ID. This implies two more states: ValidID and !ValidID. Then, starting in ValidID, the next step would be to get and verify the password. If the password were valid, the program would end, implying an End state, but if the password were not valid, you would need a ValidID-!ValidPW-1 state for this first PW error. After another invalid password, you would have a ValidID-!ValidPW-2

state, and so forth. Similarly, starting in !ValidID, you might end up with !ValidID-1, !ValidID-2, and so forth. You might also consider the sequence: !Valid ID, Valid ID, !Valid PW, and so forth. This approach typically produces a large number of states that must be condensed into a reasonable design. However, because there is no general method for simplifying state machines, you might not end up with a very clean design.

A generally effective state-machine design strategy is as follows:

1. Think through the logical flow of the system to understand it.
2. Devise a solution strategy.
3. Define the decisions that must be made by the program to implement this strategy.
4. Identify the information required to make these decisions.
5. Identify the subset of this information that must be remembered.
6. Determine the minimum number of states required to hold all valid combinations of the remembered information.

For example, in the LogIn case, the strategy is to always request an ID and a password, whether or not the submitted ID is valid. This makes it more difficult for potential attackers to identify valid IDs and passwords. Therefore, the program must make only two determinations: after requesting an ID, it must determine whether to transition to the CheckPW state or the End state; and after requesting a password, it must determine whether to end the session or check another ID. The only information needed for the first decision is whether a timeout has occurred. For the second decision, the program must know if the most recent ID was correct, how many ID and password errors have occurred, whether the current password is correct, and if there has been a password timeout error. Because this information can be held in a few Boolean parameters, only four states are needed: Start, CheckID, CheckPW, and End.

Search State-Machine Example

For a slightly different and more complex case, consider the state machine to conduct a search. The objective is to find the value of x for which some function F(x) = M, a user-specified value, where F(x) is assumed to monotonically increase or decrease with x. That is, for each x, there is a unique value of F(x) and the reverse. In addition, assume that you have a program that will calculate F(x) for a given value of x. One solution to this problem is a simple searching procedure that makes successive trial calculations until the value of F(x) is within an acceptable error range of the desired value M. An example of this search procedure is shown in Figure 11.5. For steps 1 and 2 of the state-machine design process, a possible solution strategy would be as follows:

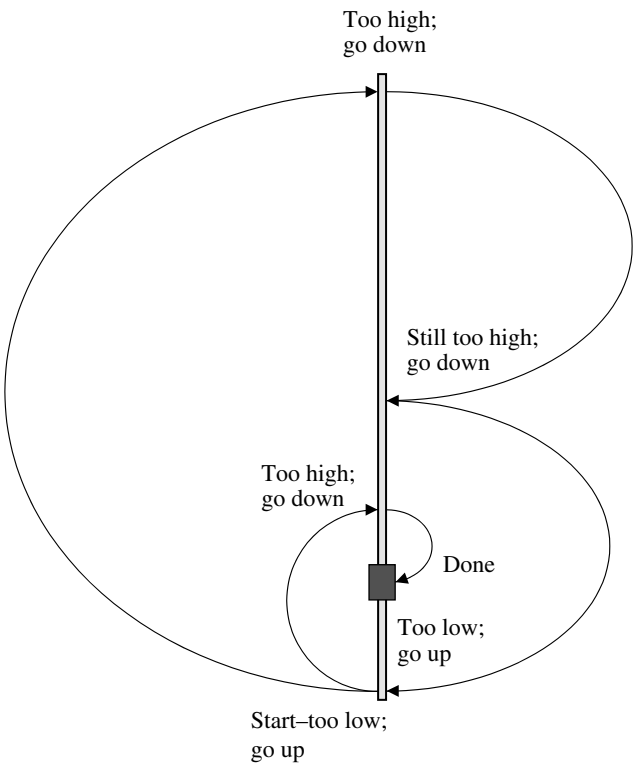

FIGURE 11.5 STATE-MACHINE SEARCH STRATEGY

1. Determine the desired final result M, select an arbitrary positive number for Delta, and define an acceptable error range, Range. The program should then find a value of x for which: M – Range < F(x) <= M + Range.
2. Assume an initial value of x_{init} and calculate F(x_{init}).
3. Also calculate values for F(x_{init} + Delta) and F(x_{init} – Delta).
4. Set error values e+ := F(x_{init} + Delta) – M and e- := F(x_{init} – Delta) – M.
5. If e+ >= e-, it means that the value of F(x) increases with x, so set Sign := +1. If e+ < e-, set Sign := –1.
6. Test to determine whether F(x_{init}) is within, above, or below the target range.
7. If the initial test was too low, increase x by the increment Sign*Delta and repeat.
8. Test to determine whether this value is within, above, or below the target range.

9. If, this time, the result is too high, go down, but by a smaller step. For a binary search, set Delta := –Delta/2 and $x := x$ + Sign*Delta.
10. Calculate the new F(x) value.
11. Test to determine whether this value is within, above, or below the target range.
12. Continue this procedure until the result x is within the target range.

With this design strategy, consider design step 3: List the decisions to be made by the program. The two decisions are whether to use a positive (Up) Delta or a negative (Down) one and when to divide Delta by 2. The next step in the state-machine design strategy is to identify the state-unique information needed to make these decisions. The decisions are whether to add or subtract Delta and whether the search is completed. The information needed is whether the current trial is too high, too low, or within the error range and whether the previous change in x was an increase or a decrease. The states don't need to remember the results of the current trial but they do need to remember whether the prior test increased or decreased x.

To decide whether or not to divide Delta by 2, look at Figure 11.5: Delta is halved every time the search reverses direction. If Delta > 0, the search was going up, so a high guess requires a reversal and halving Delta, giving Delta = –Delta/2. Similarly, if Delta < 0, the search was going down, so a low guess would require setting Delta = –Delta and dividing it by 2. Therefore, every reversal results in Delta = –Delta/2. Where there is no reversal, Delta is not changed. This suggests that the states only need to remember whether the prior search step was upward (Up – positive Delta) or downward (Down – negative Delta).

In the final design step, identify the possible states. The information to be remembered requires only the following two states: Up and Down. In addition, the program needs to know whether F(x) increases or decreases with increasing x. If a larger x results in a smaller F(x), the Delta should be subtracted instead of added. This requires a state for setting DeltaSign. Start and End states are also needed. With these states defined, it is relatively easy to complete the State Specification Template in Table 11.10, providing the transition conditions and functions for each state.

As part of state-machine design, it is usually a good idea to draw a state diagram. This will provide a clear and understandable picture of state-machine operation as well as a useful test of the design. Put the transition conditions and state functions on the diagram if they can be included without overcomplicating the picture. This is not essential, however, because the State Specification Template has all of the required information. If you have trouble producing the state diagram, either the state specification is incomplete or the design is too complex to represent clearly in a simple picture. Then you must use the SST to define the program's behavior. The state-machine diagram for the search example is shown in Figure 11.6.

TABLE 11.10 STATE SPECIFICATION TEMPLATE FOR SEARCH

Student J. Developer Date 10/30

Program Search Program #

Instructor Humphrey Language C++

State Name	**Description**	
Start	Start condition for program	
DeltaSign	For determining if F(x) increases or decreases with increasing x	
Up	Last test was below the target range	
Down	Last test was above the target range	
End	Last test was within the target range	
Function/Parameter	**Description**	
x	Initial value set for calculations, later adjusted to find x for F(x) = M.	
Target, M	The final target value = M, with M – Range < F(x) <= M + Range	
Delta	The amount by which the search range is adjusted for each test	
Sign	Indicates whether F(x) increases with x (+ 1), or decreases with x (– 1)	
e	Error in F(x) test: e = F(x) – M	
e+	Error for x + Delta: e+ = F(x + Delta) – M	
e-	Error for x – Delta: e- = F(x – Delta) – M	
x+	x+ = x + Sign*Delta	
x-	x- = x – Sign*Delta	
x+/2	x+/2 = x + Sign*Delta/2	
x-/2	x-/2 = x – Sign*Delta/2	
States/Next States	**Transition Condition**	**Action**
Start		
DeltaSign	e+ < e-	Sign := -1, Test F(x)
DeltaSign	e+ >= e-	Sign := +1, Test F(x)
End	In Range, M – Range < F(x) <= M + Range	
DeltaSign		
Up	F(x) < Target	x := x+, Test F(x)
Down	F(x) > Target	x := x-, Test F(x)
End	In Range, M – Range < F(x) <= M + Range	
Up		
Up	F(x) < Target	x := x+, Test F(x)
Down	F(x) > Target	x := x-/2, Test F(x)
End	In Range, M – Range < F(x) <= M + Range	
Down		
Up	F(x) < Target	x := x+/2, Test F(x)
Down	F(x) > Target	x := x-, Test F(x)
End	In Range, M – Range < F(x) <= M + Range	
End		

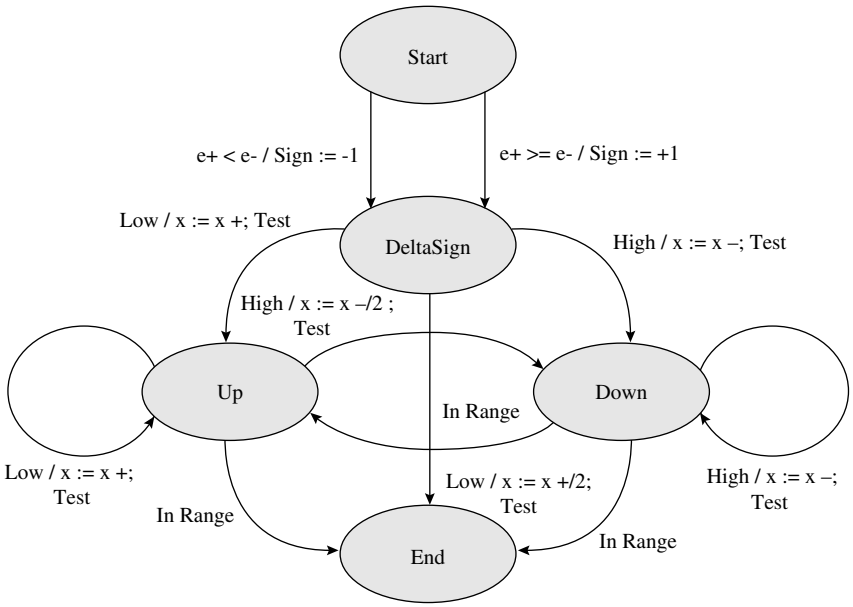

FIGURE 11.6 SEARCH STATE DIAGRAM

11.8 Using the PSP Design Templates

The principal intent of the PSP design templates is to record the design, not to aid you in producing that design. The templates precisely define what the program is to do and how it is to be implemented. Your choice of design method is particularly important because design is an intellectual process and it is hard to do good intellectual work in a language or with methods with which you are not fluent. Therefore, at least until you have used these templates for a while and are familiar with them, continue producing designs as you have in the past but record them on the templates. By completing the templates, you will often identify design problems. As you gain experience with the templates, you may also find that they help you to produce the design. The PSP2.1 exit criteria for the design phase calls for the completed templates.

The PSP design templates were made as generic as possible so they could be used with any design method. If your design method already produces the information required by one or more of the templates, you need not bother duplicating

that material. Before deciding to omit one or more of these templates, however, ensure that your design method captures all of the required information in the detail specified by the templates.

Comparison with UML

Figure 11.7 compares the PSP design templates and the UML. It shows that the OST contents are provided by a combination of the UML use case diagrams, use case descriptions, activity diagrams, and sequence diagrams. The SST is covered by the UML state diagram, but only if you include the detailed specifications for the states and state transitions. FST is partially covered by the class diagram. Here again, the shortcomings concern the precision of the functional definition. The LST has no UML equivalent, although most UML tools provide a way to enter and store the pseudocode with a design.

Redesign

Most industrial software development work, and all software maintenance work, involves modifying and enhancing existing systems. Whether making modifications or adding functions to an old product, you can still use the PSP design templates. They are particularly helpful when delving deeply into an old product to make fixes or add enhancements. Because you must understand the product to have any chance of making defect-free changes, the condition of the product's design is

	External	Internal
Static	Class Diagram with Object Constraint Language descriptions	No UML equivalent to the Logic Specification Template
Dynamic	Use Case Diagram Use Case Description Activity Diagram Sequence Diagram	State Diagram

FIGURE 11.7 UML EQUIVALENTS TO THE PSP DESIGN TEMPLATES

important. If suitable design documentation is available, the job will be easier. However, software designs are rarely documented, and when they are the documentation is usually not current, correct, or understandable. Then you must either work without a design or start with the source code, if you have it, and reconstruct the design. Although the methods for doing this depend on the product, the PSP design templates can help by recording the parts of the design that you must understand.

11.9 Using the Design Templates in Large-Scale Design

You can also use the PSP design templates to specify large software products or systems. You could specify the system's external functional and operation behavior with FST and OST. Then, as you decompose the system into multiple smaller subsystems or components, you could specify their external operational and functional behavior with additional FST and OST specifications. Where needed, you could use the SST to define the state behavior of the system and its component parts. Then, at the next level, you could repeat the same steps for the lower-level parts.

One way to visualize this design process is shown in Figure 11.8. The system requirements statement describes what the users need. The functional and

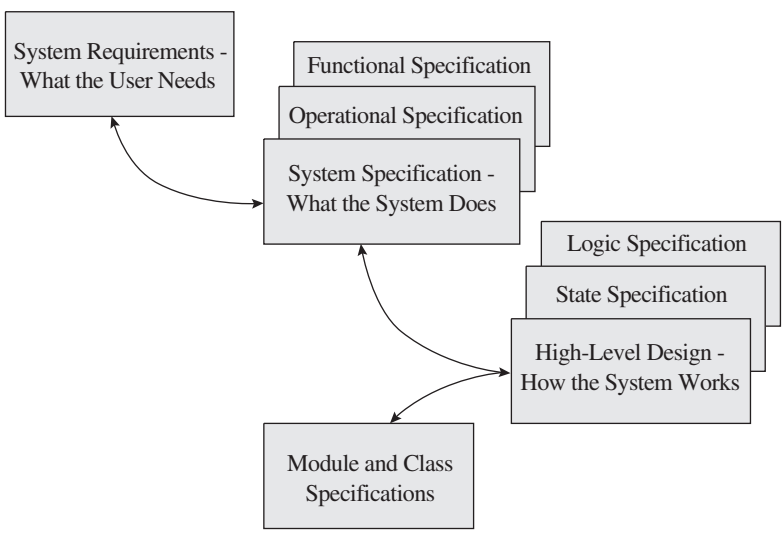

FIGURE 11.8 THE DESIGN HIERARCHY

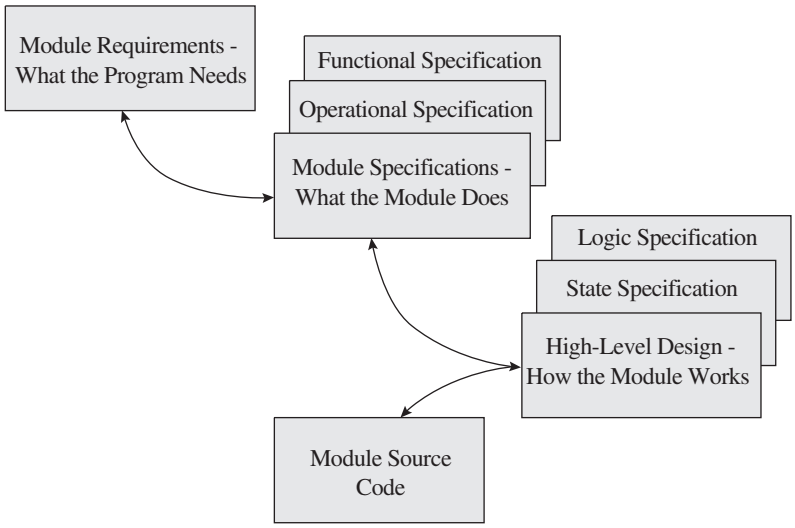

FIGURE 11.9 THE IMPLEMENTATION HIERARCHY

operational specifications then describe the system's external behavior. The system's high-level design would include an SST for its state behavior and an LST for any high-level control logic. An example of this logic would be the interrupt-control design of a real-time system. Next, postulate the services needed from the lower-level parts and capture their behavior in FST and OST specifications. For large systems, you could repeat this process several times, completing OST, FST, SST, and LST specifications at every design level. At the end, as shown in Figure 11.9, you would have the detailed design for the lowest-level modules.

This top-down strategy is only one approach. You might instead require some critical low-level function that you did not fully understand, and need to create a prototype or even complete its detailed design and code before you can count on using it. You could thus bounce from the highest design level down to implementation and back many times during design. Regardless of the strategy, do not consider any element's design complete until you have fully defined all of the logic connecting it to the higher-level program elements that use its services. You must also have specified all of the services it requires from lower-level parts. The PSP design templates can help you to do this.

11.10 Summary

In the PSP, designs are described with the aid of four templates. The Operational Specification Template describes the program's operational behavior via one or more scenarios. The Functional Specification Template describes the functions performed by a program, a class, or a procedure. The Logic Specification Template uses pseudocode to precisely describe a program's logic. Finally, the State Specification Template describes the program's state behavior.

You can use these templates to specify the internal and external behavior of both large systems and small programs, as well as the behavior of their parts. The PSP templates are designed to capture the precise content of a design and are intended to complement existing design methods. The intent is for developers to produce designs with whatever methods work best for them and then to record these designs in the PSP templates. Doing this will usually identify errors, oversights, and inconsistencies, as well as provide a useful design record.

11.11 Exercises

The assignment for this chapter is to use PSP2.1 to write one or more programs. Use the PSP design templates for these programs and, if the program has a search function, consider using a search machine like the one described in Table 11.10 (see p. 245) and Figure 11.6 (see p. 246). The PSP2.1 process and the program specifications are provided in the assignment kits, which you can get from your instructor or at **www.sei.cmu.edu/tsp/psp**. The kits contain the assignment specifications, an example of the required calculations, and the PSP2.1 process scripts. In completing this assignment, faithfully follow the PSP2.1 process, record all required data, and produce the program report according to the specifications given in the assignment kit.

References

Booch, G., J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Reading: Addison-Wesley, 1999.

de Champeaux, D., D. Lea, and P. Faure. *Object-Oriented System Development.* Reading, MA: Addison-Wesley, 1993.

Jacky, J. *The Way of Z: Practical Programming with Formal Methods.* Cambridge, UK: Cambridge University Press, 1996.

McConnell, S. *Code Complete.* Redmond, WA: Microsoft Press, 1993.

Rumbaugh, J., I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Reading, MA: Addison-Wesley, 1999.

Schneider, G., and J. P. Winters. *Applying Use Cases.* Redmond, WA: Microsoft Press, 1998.