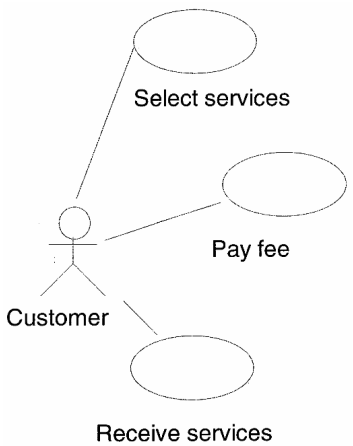
Chapter 6: Designing the Modules

1. Answers to this question may vary, since some details for the specific scenarios will be filled in by the reader. The number of distinct use cases is also subjective, as long as the use cases in combination describe all of the car wash functionality specified. One solution is as follows:



The "select services" scenario begins when the customer drives his car up to the control panel. The system displays on the control panel is listing of the different levels of service available and the price for each. The customer initiates a transaction by pushing a button on the control panel corresponding to a particular service. The system then prompts the customer to indicate (by pushing a button on the control panel) the type of car he or she is driving (e.g. compact, midsize, minivan, truck), or to cancel. if the customer selects "cancel", the system resets and the customer can select a different service, if desired.

The "pay fee" scenario begins when the customer has finished selecting the services. The system then computes the fee and displays the amount on the control panel. If the customer does nothing, then after a short period of time the system resets itself. Otherwise, the customer inserts money into the money reader, which is responsible for reporting to the system the value of the money just entered. [Note that because the car wash is to be an automated system, we assume the existence of an external system for accepting customer payment.] The system compares the value of the money just entered to the fee remaining to be paid. If the amount just entered is greater, then the system instructs the money reader to return the proper amount of change; otherwise, the system calculates the remaining fee and instructs the money reader to display this amount on the control panel. When the fee has been completely paid the system is ready to dispense services.

The "receive services" scenario begins when the customer has completely paid the fee. If the car wash is currently busy, the control panel indicates that the customer

should wait. If the car wash is empty, or when the car wash becomes empty, the control panel instructs the customer to drive forward. When the sensors indicate that the customer has driven forward to the appropriate position, a “stop” light is illuminated inside the car wash. The system then begins to dispense services. Depending on the type of vehicle indicated by the customer, particular hoses and nozzles may or may not be activated. Once services have been dispensed, the “stop” light is no longer illuminated. The customer drives out of the car wash.

2. Lorenz and Kidd’s specialization index is a measure of how much of the functionality of a class is specific to that class rather than inherited by the class from elsewhere in the inheritance hierarchy. It may help to think of the metric being defined as (NOO/total class methods) * the level of the class in the inheritance hierarchy. The first term then measures what percentage of the methods available to the class are specifically defined in the class itself. A low value for this term means that very few of the class’ inherited methods were overridden with behavior specific to the class itself, that is, that most of the methods used by this class were defined elsewhere and inherited as-is. Conversely, a high value indicates that most of the methods in the class were overridden with specific behavior. The second term weights the index by increasing the value the deeper the class is in the inheritance tree, i.e. classes with more superclasses are assumed to be more specialized. This makes sense both semantically (each new level of an inheritance tree should represent a further refinement of some organizing concept) and quantitatively (the value of the index should be increased to allow for the fact that some methods may have been overridden in the parent classes). A major change in the index should indicate that the level of inheritance should be questioned. If the index becomes lower and lower, there is less and less specific functionality in this subclass (and perhaps less and less reason to keep the extra layer of the inheritance hierarchy). If the index becomes higher and higher, there is less and less functionality from the superclasses still present in the class (and perhaps less and less reason to keep this class as part of the hierarchy rather than an independent class).

3. Any system can be expressed using an object-oriented approach, but that does not mean this approach is always the right one. OO has many strengths: It allows a consistent vocabulary to be used across different phases of the development process, facilitating traceability. Many claim that it helps designers understand the problem better by allowing the design of classes that mirror real problem components. By providing a number of different diagrams, representing both dynamic and static aspects of a system, OO facilitates the analysis of various aspects of the system being designed. And, OO facilitates information hiding and encapsulation, which are widely understood to be beneficial design strategies. However, OO does have its weaknesses; maintenance of an OO system can be a difficult task because information about a particular aspect of the system can be spread across several different types of diagrams. Information hiding can help developers gain a broad understanding of the design but can make other tasks harder, such as impact analysis. Features such as inheritance and polymorphism can be helpful in the hands of an experienced designer but can create hard-to-understand systems if used poorly. One type of system for

which OO might not be the best choice is a rule-based system. When a system is mostly concerned with algorithmic processing or rule look-up (i.e. there are few discernable entities and the real complexity comes from computation), OO does not provide many applicable benefits.

4. One way to refine the design to avoid stamp coupling is to use interfaces. By designing interfaces to access the necessary data contained in the complex data structure, it is possible to reduce the stamp coupling to a data coupling since now the component depends on atomic data instead of complex data types.

5. Clearly the designers of the Ariane-5 system made a serious error when they made this decision, and making another decision would have prevented the disaster. However, equally clearly, the designers did not intend to cause such a disaster. The error was due to incompetence, not malice. The designers share responsibility for the disaster with everyone involved in the development of the software, although the decision they made seems to be at the root of the chain of events that led to the disaster.

6. Coincidental: a component that prints the current time or gives a directory listing, depending on the user input.
   Logical: a component that prints a document or writes it to a file.
   Temporal: a component that logs users into the system, check their mail and shows them the calendar for the day.
   Procedural: a component that reads a user database query, checks the availability of the database, then searches for the requested information.
   Communicational: a component that collects disk access housekeeping data while accessing user-requested data.
   Sequential: a component that requests a user password, reads the password, checks the password and initiates the user's session.
   Functional: a component that validates a user password and does nothing else

7. Content: component 1 reads in a stream of characters and updates a variable, which is internal to component 2, that keeps track of the number of lines that have been read. Component 2 uses this variable to calculate when a new page needs to be started.
   Common: components 1 and 2 are as above, but the line counter variable is in a shared data space.
   Control: component 1 reads in a stream of characters, keeping track of the number of lines read, and invokes component 2 which performs a calculation based on the data in that structure.
   Stamp: component 1 performs a validity check on a complicated data structure, then passes that structure to component 2 which performs a calculation based on the data in that structure.

Data: component 1 performs a validity check on a complicated data structure, then passes individual elements of the data structure that are needed by component 2 to complete its calculation.

8. Answers to this question may vary.

9. Probably not completely. For example, consider a system that provides a number of independent services in response to a user request. The components providing the services could be decoupled, but they would each be connected to the component which reads the user request and then calls the appropriate service.

10. Any system design could be made cohesive by making it monolithic. That is, one way to ensure that each component is self-contained is to have only on component that does everything in the system. This would be a functionally cohesive design. The trade-off would be that such monolithic designs are generally much harder to maintain and test.

11. Some examples of quality attributes and the effect of good design:
    reliability: system designs that exhibit high modularity and low coupling will generally have higher reliability because they will be easier to implement and test and because the use of modularity and low coupling avoids many types of errors that are difficult to find and fix.
    traceability: system designs that exhibit high cohesion enhance traceability because the relationships between functions in the requirements, design, and code are clearer when each component corresponds to just one functions.
    maintainability: modularity, low coupling, and high cohesion all contribute to maintainability by facilitating understanding of the code and ease of modification.

12. In many situations, a recursive component is a very good idea because it is the most straightforward way to implement some types of algorithms. It preserves good design principles if the recursion is straightforward and direct. That it, the component calls only itself directly and does not involve other components indirectly in the recursion. Otherwise, it greatly increases and complicates the coupling between components.

13. Answers to this question may vary.

14.
```
addword(french: WORD, english: WORD)
//adds English and French words and their associations to the
//dictionary.
requires: 'french' is a French word and 'english' is an English word
          not associated in the dictionary.
ensures: adds 'french' and 'english' if not in dictionary,
          associates 'french' and 'english'.

getPronunciation(french: WORD)
//retrieves the pronunciation of a French word from the dictionary.
requires: 'french' is a French word in the dictionary.
returns: the pronunciation of the word specified in 'french'.
```

```
translate(english: WORD)
//retrieves the French translation of an English word form the
dictionary.
requires: 'english' is an English word associated with a French word
         in the dictionary.
returns: a French word that is associated with the word specified in
         'english'.
```

15. This module could be redesign in different ways to improve its generality. One way would be to allow input of types other than INTEGER, such as DOUBLE, or FLOAT. This would allow the module to be used in sorting different types of number representations. A second way would be to include a second parameter to indicate if the sorting should be done in decreasing or non-decreasing order.

16. This module might fail if one of the arrays is null, or if at least one of the arrays does not contain integers, or if the arrays are of different sizes. This module could recover from these failures by raising one or more exceptions dealing with the failure. The responsibility of passing correct values, as well as how to deal with improper inputs, would rest with the calling module.

17. This is a bad use of inheritance because Stack is not a specialization of a List. There is no overlapping functionality between the two data structures and the methods provided by the List do not apply to the way a Stack works.

18. Specification (a) is not substituable by any other specification. Having val not in the list is not required in (b) nor (d) as it is in (a), (c) does not require anything and is therefore looser than (a). Specification (b) is substituable by (a) and (d) since both specify the same behavior without requiring that val is not in the list, even though specification (d) is more strict than (b). It is not substituable by (a) nor (d)

19. The specification a' is better than a since it uses active fault detection. This means that the responsibility of ensuring passing correct input parameters, or ensuring that the system recovers from an invalid state, belongs to the calling module and not the module being called.

20.

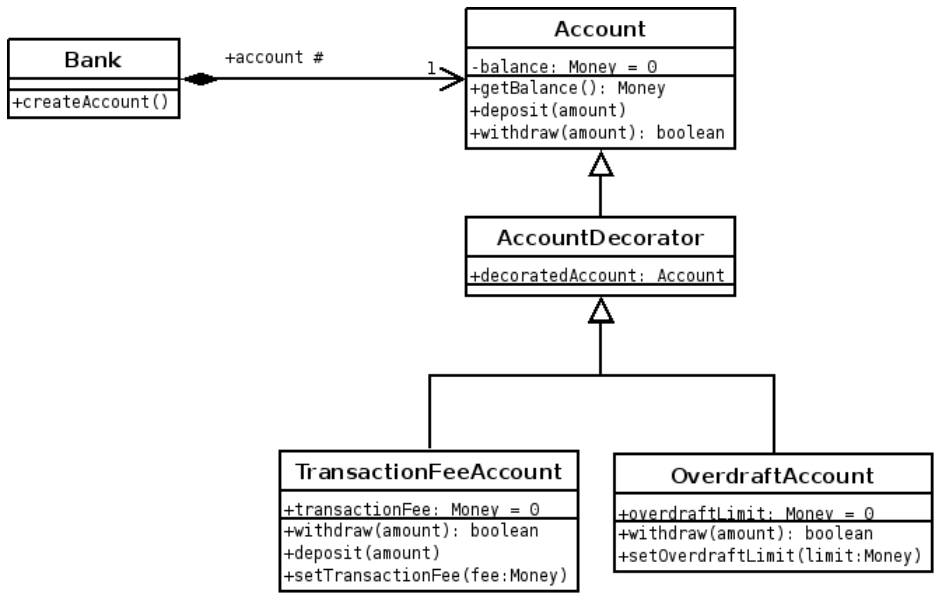
```
local
   count: INTEGER
   capacity: INTEGER
   dictionary: function that maps Key to Element
   insert(elem: Element; key: String)
   // Insert elem into dictionary
         ensure: has(key) and retrieve(key) == elem,
         if count > capacity then raise CAPACITYEXCEEDEDEXCEPTION,
         if has(key) then raise DUPLICATEKEYEXCEPTION
         if not key.valid() then raise INVALIDKEYEXCEPTION
   retrieve(key: String): Element
   // Returns the element indexed by key
   ensure: result = dictionary(key),
```

```
         if not has(key) then raise ELEMENTNOTINDEXEDEXCEPTION
end


local
   gauge: INTEGER
   capacity: INTEGER
   fill()
      //Fill reservoir with water
      ensure: in_valve.closed and out_valve.closed and is_full,
            if in_valve.closed then raise INVALVECLOSEDEXCEPTION,
            if out_valve.open then raise OUTVALVEOPENEXCEPTION
   is_full(): BOOLEAN
      // Tests whether reservoir is full
      ensure: result == (0.95*capacity <= gauge)
end
```

21. The second one is better. Returning the smallest representable integer does not ensure that the input parameter is valid, and therefore can result in unexpected behavior. For example, passing an array containing the smallest representable integer and passing an empty array would both result in the same output. By requiring that the input is not empty, the module ensures that the output correlates to the input.

22. The figure below shows both the overdraft protection and transaction fee modifications to the banking system.



23. The figure below shows the application of the Observer pattern to the banking system.