Managing Scope Creep with Design Patterns in Formal Specifications

Marc Parisi
Marc.parisi@gmail.com

Abstract

Scope creep can affect the timeliness of the software engineering process. Scope creep often occurs when features are changed or added. Software specification languages provide a formal method for extrapolating and expanding requirements. Several design patterns provide object oriented constructs whose goal are to mitigate changes in code. Applying these design patterns to formal specifications can help mitigate changes.

Structural patterns realize communication between objects, thus identifying how these objects and communications can change. Using a combination of these pattern types, specifications can be redesigned with the focus of mitigating scope creep. This paper will focus on the patterns which fall into these two categories to design specifications, even if the requirements don't specify this behavior.

This paper will identify a template of specifications which will aide developers in identifying how specifications can be designed with mitigating scope creep in mind.

Keywords*: requirements, formal specifications, design patterns, scope creep*

1. Introduction

Software specification languages, such as Z, typically leave details out of the underlying development constructs to the design phase [2]. The specification languages extrapolate and expound requirements, but do little to help design phase issues and developmental decisions. Specifying requirements correctly is just as important as correctly implementing the design as they both lead to mitigating issues in the design phase, particularly scope creep [4]. Software design patterns can be applied during the software design phase to improve development and reduce issues associated with poor design decisions [4,5]. This paper asserts that using design patterns to develop specifications can mitigate the costs of scope creep by creating specifications in a way that closer matches the ultimate design.

The specification language used throughout this paper will be Object-Z. Object-Z is an extended Z specification language which allows object oriented constructs, including classes, inheritance, and polymorphism. Figure 1, below, depicts a schema class formed using Object-Z

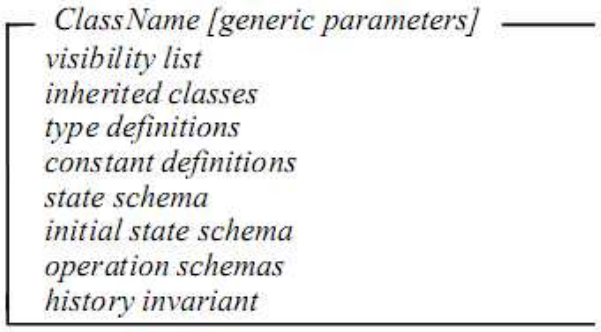


Figure 1 -- Object-Z class definition

Under no circumstance does this paper attempt to sway readers into using Object-Z over another specification language, as the ultimate choice is up to the specification designer. As we will see, the purpose of this paper is not to design an object oriented specification, but rather to use object oriented design patterns to further refine specifications. [6]

2. Usage of design patterns in specifications

As stated, the intent of this paper is to help design specifications for the purpose of mitigating scope creep by incorporating abstraction into the specification language. Object-Z already includes functionality for this purpose; however, in order to achieve this goal we implicitly perform the following tasks: include object oriented design practices into the requirements phase and refining requirements. As we will see in the following sections, these tasks are often performed as result of using design patterns in specifications. [6]

2.1 Object oriented design practices

As we will see shortly, most design patterns abstract elements of a software package to create a more vertical design. By doing this in specification phase we can fully and correctly abstract requirements. This will cause the specifications to be clearer and more open to change [1]. The *Gang of Four* described the open/closed principle, which states that software should be open to extension, but closed to modification [3]. This paper functions upon the same principle to mitigate the scope creep by using proper design techniques, which ensure changes do not affect the system.

2.2 Refining requirements

In the event that requirements do not fully support the processes outlined in this paper, they are to be refined, but not necessarily refactored. To better explain this principle, think of a simple Queue software package, which accepts a particular data type ***T***. The package en-queues and de-queues only elements of type ***T***. A specification may define the queue elements as a particular set of type ***T*** elements, modeling the requirements. However, as per the mantra of object oriented design, we may wish to modify the queue specifications to be a set of type ***O***,

which is in fact an abstract class that defines required constructs to be used within queue. In this example, we refined the requirements to be more generic and object oriented.

3. Design patterns

3.1 Structural patterns

Structural patterns help developers to realize relationships between entities. Often, these relationships can be better defined with the designation defined within each pattern. In this section we will discuss the façade, composite, and bridge patterns. While many other structural patterns exist, these three will be the most suited to the intent of this paper. [1, 3]

3.1.1 Façade pattern

The façade pattern provides a unified interface for a class [1] The façade can be conflated to the adapter patter, as they differ only slightly; however we will limit our discussion to only the façade pattern, depicted in figure 1, below [1]. ClassX interacts with the subsystem(s) through the Façade class. The general structure of the Object-Z template will not be dissimilar.

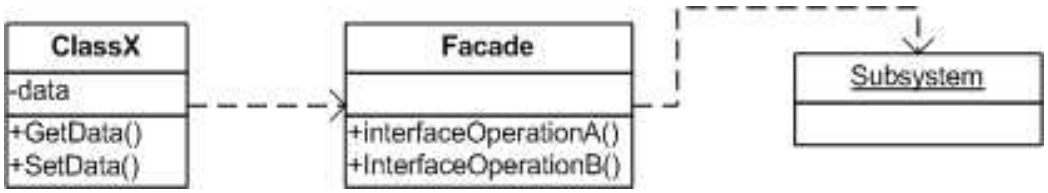


Figure 2 -- Facade Pattern

The façade pattern can be used in specifications to provide the same advantage it has in design: unifying interfaces. When writing specifications, we will use the class diagram in Figure 1 as a template for our specifications.

3.1.2 Composite Pattern

The composite pattern, as defined by the Gang of Four, is a design pattern which partitions a group of objects and treats them as a single instance. The composite pattern partitions the objects into a tree structure to show part and whole relationships [1]. Figure 4 depicts the composite pattern's UML class diagram. When designing specifications using Object-Z, we will use the composite pattern in our specifications to encapsulate hierarchies within our requirements. This is especially helpful if we can structure the requirements in a hierarchy or refine them to construct a hierarchy of requirements, thus reducing the necessary specifications.

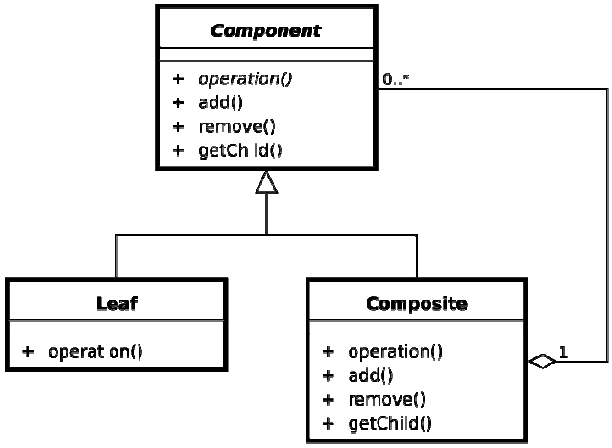


Figure 3 -- Composite Pattern

3.1.3 Bridge Pattern

The bridge pattern decouples abstraction from the implementation [1]. The class diagram in Figure 4 depicts the relationships between the abstractions and implementations. Using this same strategy when designing specifications can accomplish the same goal by refining requirements so that they explicitly allow for abstractions in the specifications. Since Object-Z can identify classes and implementing classes, we can depict composition and inheritance within the specifications.

The bridge pattern is meant to be linked the abstractions with the implementations, so if the specifications define an abstract class, we can move abstraction elements to concrete abstractions and implementation details to the associated concrete implementations. To do this, developers may need to refine the requirements so that we can visualize the abstractions within the specification.

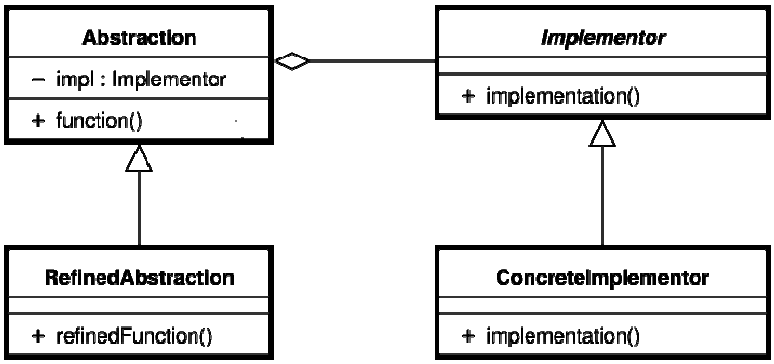


Figure 4 -- Bridge Pattern

4. Implementing Design Patterns in Object-Z

Now we will begin the process of implementing these design patterns into a specification. To do this, we will identify requirements in each section that correspond to a typical requirements issue that the pattern can help solve. Throughout the discussion we will use a prevailing example of a date book. Informally, we specify our date book system with the following statement:

A system which will allow the reservation and listing of time slots

To fulfill this requirement we have defined the specification in Listing 1, below. In the following sections, we will discuss the deficiencies in this specification along with how the corresponding design pattern addresses these problems.

DateBook

$$reservee : \mathbb{P}\ Person$$
$$reserved : slot \pfun reservee$$
$$reservations : \mathbb{P}\ slots$$
$$\text{dom}\ reserved \subseteq reservations$$

InitDateBook

$$DateBook$$
$$reservations = \phi$$

ReserveSlot

$$\Delta DateBook$$
$$p? : reservee$$
$$s? : slot$$
$$r! : Response$$
$$s? \in reservations \quad \text{dom}\ reserved$$
$$reserved' = reserved \cup \{s? \mapsto p?\}$$
$$reservations' = reservations$$
$$r! =' Success'$$

AlreadyReserved

$$\Xi DateBook$$
$$p? : reservee$$
$$s? : slot$$
$$r! : Response$$
$$s? \in \text{dom}\ reserved$$
$$r =' SlotUnavailable'$$

ListReservations

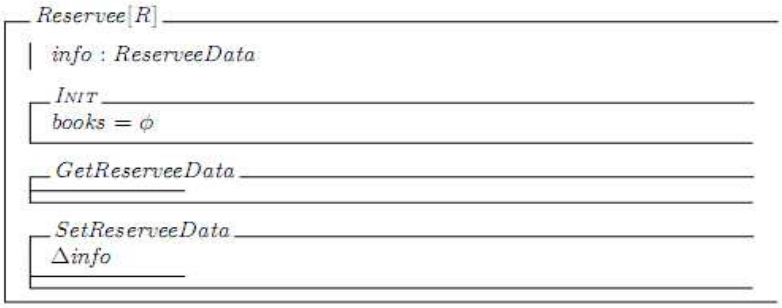
$$\Xi DateBook$$
$$s? : slot$$
$$p! : reservee$$
$$s \in reservations$$
$$p! = reserved(s?)$$

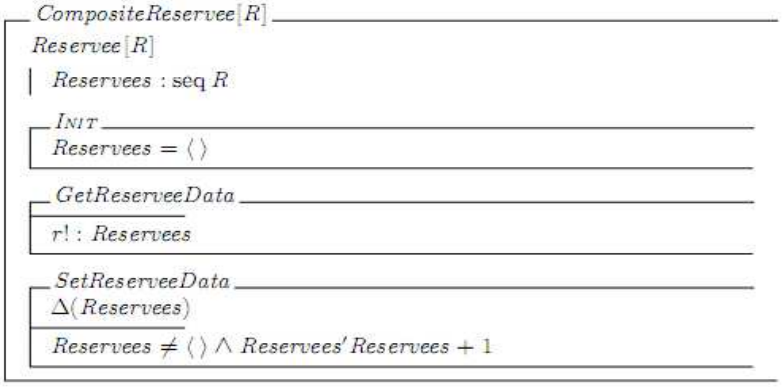
$$CreateReservation \mathrel{\widehat=} ReserveSlot \lor AlreadyReserved$$

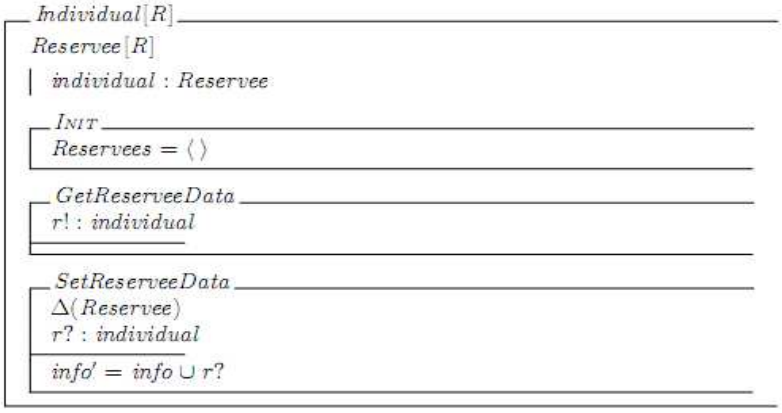
Listing 1 – DateBook

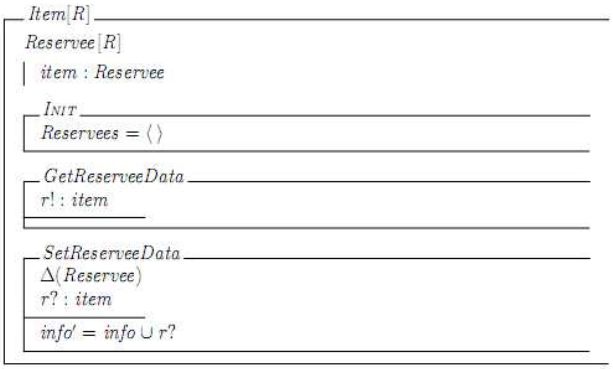
4.1 Modifying the specification

The above specification does not indicate whether reservee is a person or a group of people. Further, the DateBook specification does not allow for variations in the design. To remedy this, we will couple the composite pattern with the bridge pattern. First, we will create a reservee specification in Object-Z with a composite pattern. This will allow us to interact with a myriad of reservee objects in the same manner. Listing 2, below depicts the abstract definition of Reservee, along with the concrete specifications.









Listing 2 – Bridge Pattern

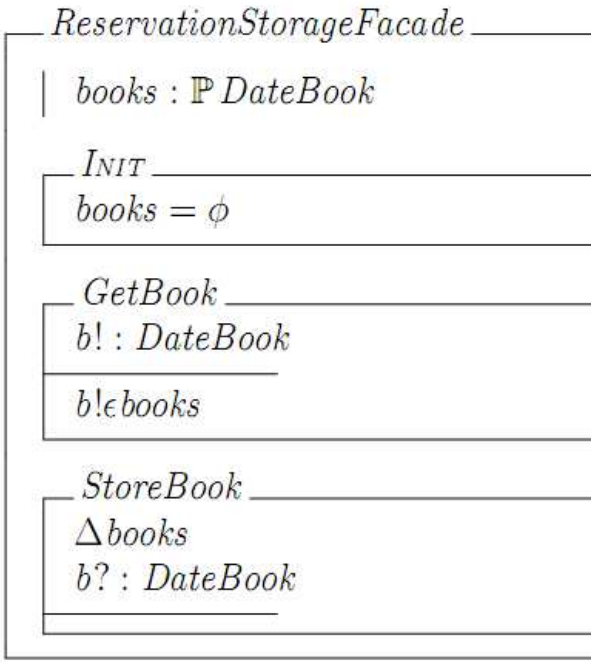
The CompositeReservee specification is the composite class, which contains a sequence of abstract Reservee classes. Since CompositeReservee is derived from Reservee, the sequence can contain the composite classes as well, thus allowing us to have a hierarchy. This hierarchical structure is represented by the composition of similar classes. With this new specification, we can access all Reservees with the same interfaces, whether it is a group of Reservees, an item, or an individual.

Now that we have an object oriented specification for Reservee, we will modify how we reserve and list a reservation. To do this, we will abstract the initial Datebook specification and create implementations for each abstraction. The changes will be represented by the bridge and composite patterns. In our example below, we have DateBook as our refined abstraction and CalendarReservation as our concrete implementation.

Listing 3 -- Composite Pattern

With the composite pattern we have an abstract reservation, AbstractReservation, which defines the type of reservation, which can be extended at any time. The abstract implementation, specified by AbstractImpl defines an implementation that DateBook will use to implement its methods. In the above listing, we implemented a calendar reservation system. If we wish to extend the types of Reservees or the type of reservation system, we must simply specify another type. By implementing the bridge and composite patterns with Object-Z, we strictly follow the Open/Closed principle.

Listing 1 does not address storage constraints, leaving the method in which the data is stored up to the design phase; however, the façade pattern can be used to limit ramifications of design phase changes by specifying a façade to the storage medium. Specification 2, below, depicts a sample façade specification. In this example we create a storage façade with two methods: GetBook and StoreBook.



Listing 4 -- Facade

Creating a façade abstracts the storage of the date books to an outside interface. The date book specification can be changed to incorporate the façade above, which will contain storage implementation details. Therefore, if the storage medium were to change, the date book specification need not be changed.

The above patterns, when employed with the example specification, allow us to incorporate object oriented principles into our requirements. We can easily extend the system by adding our refined abstractions and implementations. Any extension would not require refactoring of already established specifications.

5. Conclusion

As we've seen with our DateBook example, we can effectively modify a specification with Object-Z and several structural design patterns to help eliminate scope creep. Scope creep, particularly the addition of features can slow development; however, if the specifications already establish the Open/Closed principle in our requirements, we can eliminate scope creep before and during the design phase [4]. The modifications are relatively easy and follow Object-Z and object oriented principles.

References

[1] A. Shalloway, J. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison Wesley, Boston, Massachusetts, 2005.

[2] S. Diller, *Z: An Introduction to Formal Methods*, Wiley, New York, NY, 2004.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Boston, Massachusetts, 1994.

[4] K. Ewusi-Mensah, *Software development failures : anatomy of abandoned projects,*MIT Press, Boston, Massachusetts, 2003.

[5] S. McConnell, *Code Complete*, Microsoft Press, Redmond, Washington, 2004.

[6] G. Smith, "Object-Z", http://www.itee.uq.edu.au/~smith/objectz.html, Current: April 12, 2009.

Authors

Marc Parisi is a graduate from Mississippi State University, and works in Maryland as a Software Engineer. He studied Computer Science, with emphases in Digital Forensics, Operating System Design, and Software Engineering.