A Domain-Specific Architectural Foundation for Engineering of Numerical Software Systems

ARBI GHAZARIAN
Department of Engineering
Arizona State University
7171 E. Sonoran Arroyo Mall, Peralta Hall, Rm 230R, Mesa, AZ 85212
USA
Arbi.Ghazarian@asu.edu http://www.cs.toronto.edu/~arbi/

Abstract: - Numerical computations have found vast applications in numerous areas of science and engineering. Many scientific and engineering advances rely on the capability to build computational models in the form of numerical software systems. A review of the numerical computation literature of the past few decades reveals that research and practice in this area have been largely focused on designing efficient numerical algorithms to carry out numerical computations, but less attention has been paid to the architectural design of large-scale numerical software systems. On the other hand, the field of software engineering has made tremendous advances in the past few decades in various areas of software development including programming languages, development methodologies, and design techniques. Given the current mature status of the algorithmic design aspects of numerical computations, a change in research direction from lower-level algorithm design of numerical computations to higher-level architectural design of numerical software systems can yield fruitful results. In line with this objective, this paper aims to address the architectural level issues of numerical systems by taking advantage of the latest advances in software engineering and adapting them for the domain of numerical systems. This paper presents the design of a domain-specific architecture, which can serve as an architectural foundation for developing a large family of numerical software systems. A number of design strategies are presented and the rationale behind each strategy is explained. Using the Java programming language, we demonstrate how each of the strategies can be implemented in an object-oriented language. We further demonstrate that a set of basic design principles underlie a variety of design techniques.

Key-Words: - Numerical Software, Software Engineering, Software Architecture, Architectural Patterns, Design Patterns, Design Principles

1 Introduction

The cost of developing large-scale software systems is immense. Finding an economical approach to developing software systems has always been a great challenge to the field of software engineering. A review of the software engineering literature reveals that research and practice in software engineering have been focused on two major areas to address this cost problem: maximizing software reuse, and minimizing software maintenance costs. The significance of software architecture in achieving both of these goals is well understood. The specific characteristics of the classes of problems that are to be solved by a software system – the application domain of the system – can have a strong bearing on the architectural design of systems. For instance, the domains of enterprise computing, numerical systems, and embedded systems each deal with different types of requirements. Domain-specific architectures can provide a solution to demands for decreasing development costs while increasing quality and productivity by constricting the gap between the problem space and the solution space in a domain.

The rest of this paper is organized as follows: In Section 2, we will design a software architecture

that can be reused across the numerical computation domain to develop a large family of numerical systems. We will use a wide array of software engineering best practices in the form of a set of architectural decisions or strategies. We will explain the rationale for each architectural decision and provide guidelines for implementing each strategy. In Section 3, we will uncover the basic design principles that underpin the architectural decisions made during the design of the system. Conclusions and directions for future work follow in Section 4.

2 Designing a Software Architecture for the Numerical Domain

In what follows, we will discuss nine design strategies that have been adapted to meet the requirements that arise from the specific characteristics of numerical software systems. We will take a uniform approach in discussing these design strategies. For each design strategy used in the design of the numerical system, we will first present the design strategy. We will then discuss the justification or the rationale behind using that design strategy. Finally, we will demonstrate how the design strategy can be implemented in an object-oriented language like Java.

Throughout our discussion, we will use a general quadratic equation as a motivating example to illustrate some of the design principles. Our focus here is on the architectural aspects of the numerical systems rather than the mathematical aspects so we deliberately chose a simple mathematical topic like quadratics to ensure that the mathematical complexities of the example will not distract us from our main discussion.

2.1 Strategy No. 1

2.1.1 Using a Layered Architectural Style

Selecting an architectural style is one of the most high-level decisions to be made in the early stages of developing a software system. The nature of the layers is dependent on the domain of the problem for which a software solution is desired. An efficient method for identifying these layers is to think of typical usage scenarios of the software system. For instance, a numerical software system might perform various numerical computations such as evaluating functions, solving linear and non-linear equations, integrating, differentiating and so on. Regardless of all the differences in the various types of computations performed by a numerical software, a common input-computation-output pattern can be immediately identified. As a simple example, consider the case where the user of a hypothetical numerical software system wants to solve quadratic equations of the form

$$(1)\ x^2 + bx + c = 0$$

The user enters as input the coefficient values *b* and *c*. The numerical software system computes the roots r_1 and r_2 of the quadratic equation based on some known algorithm and displays the results to the user. This suggests that at the highest level, a typical numerical software system is composed of three layers: the input layer, the computation layer, and the output layer. Strategy No. 1 suggests that these three layers should be explicitly reflected in the architecture of the numerical software system.

2.1.2 Rationale for Strategy No. 1

A layered or tiered architecture is composed of a number of layers, each in charge of a logically cohesive functionality. This means that each layer is responsible for one single aspect of the software system. Thinking of a software system as a set of layers makes it easier to comprehend the system.

Layers of a typical programming language compiler or the OSI seven-layer reference model for computer network communication [4] are classical examples of layered architectures. For instance, thinking about a compiler as a number of layers namely lexical analyzer, syntax analyzer, intermediate code generator, optimizer, and code generator, at the highest level, makes it easier to comprehend the whole system.

Decomposing a software system into a number of cohesive layers contributes towards achieving a system that is less complex and more understandable and thus easier to maintain. Moreover, future changes to the functionalities of the layers, such as corrective maintenance changes or new features requests are isolated inside layers.

2.1.3 Implementation of Strategy No. 1

At the highest level, we will represent the entire numerical software system as a package called *numerical*. Then we will partition the numerical system into its architectural layers. Each

architectural layer can be implemented as a sub-package so we will create three packages under the *numerical* package.

a) *numerical.input*

b) *numerical.computation*

c) *numerical.output*

Each of these packages corresponds to a layer in the software architecture. Note that throughout this paper, we adhere to Java's naming conventions.

2.2 Strategy No. 2

2.2.1 Decomposing the Numerical System into Subsystems

Distinct functional areas of a numerical software system (i.e., mathematical subject areas) determine the scope of the system. For instance, a numerical software package might be specifically designated to solve linear and nonlinear equations, while another package may deal with differentiating, integrating and solving differential equations. Each of these topics is considered as a separate area in mathematics. The idea here is to divide the numerical software system into a number of subsystems, each corresponding to a single mathematical subject area. This means that the natural partitioning that decomposes the problem space into various subject matters is also used to partition the solution space.

2.2.2 Rationale for Strategy No. 2:

Here we apply the functional decomposition philosophy at a high level to make it easier to understand, develop, and maintain the system. This form of functional decomposition assures that each subsystem or module of the numerical software system focuses on one single class of numerical problems. This will reduce the overall complexity of the numerical software system by decomposing it into a number of simpler subsystems. As we move on, we will further divide each subsystem into smaller modules to make it easier to deal with. At this level of decomposition, it is reasonable to divide the numerical software into a number of subsystems based on various categories of numerical problems for which a software solution is desired.

2.2.3 Implementation of Strategy No. 2:

Similar to layers, subsystems can also be represented using packages. For each subsystem representing a specific category of numerical problems, we will create a sub-package under each of the input, computation, and output packages. As an example, if we want to design a numerical package, specifically for solving linear and non-linear equations, we will divide each architectural layer of the numerical system into two subsystems: the linear subsystem, and the non-linear subsystem. For instance, as shown below, we will extend the packing structure resulted from implementing Strategy No. 1 to support the nonlinear subsystem:

a) *numerical.input.nonlinear*

b) *numerical.computation.nonlinear*

c) *numerical. output.nonlinear.*

This process effectively partitions each architectural layer into a number of subsystems. Further strategies can be defined to deal with cases where a component can be shared among subsystems. For instance, we can dedicate a common package to such cases.

2.3 Strategy No. 3

2.3.1 Creating Abstract Data Types to Represent the Input and Output Data Aspects of the System

The idea here is to bundle all the required input or output data of each numerical use case along with the operations that manipulate or access that specific data into an abstract data type. The resulting input and output data objects will facilitate working with the underlying data.

The internal data structure is private to the containing object and is only accessible through public accessor or mutator methods. As an example, the quadratic equation (1) can be represented by the values of its coefficients *b* and *c*. We can create an abstract data type encapsulating these two coefficients to represent quadratic equations. Then we can define a number of useful operations on the data contained within this abstraction. As an example, we can define an operation that calculates the discriminant of the quadratic equations using the following formula:

$$(2)\ d = b^2 - 4c$$

From basic algebra we know that if the discriminant is less than zero then the two solutions are complex numbers so we can define another useful operation that tells us whether the roots are complex.

When the object representing the quadratic equation is passed to the computation layer, the operations – or methods - in the computation layer of the system that solve the equation will interact with the passed object using its accessor, mutator, and utility methods. It should be noted that we limit these utility methods to only computations that are directly related to the data aspects of the object. Capturing the input data representing the numerical problem to be solved, performing conversions and transformations, performing calculations based on the encapsulated data that reveals specific properties of the numerical problem, and data validations are behaviors that are appropriate to be encapsulated in these abstract data types.

Similarly, we can create an output abstract data type that encapsulates the roots of the quadratic equation. Here, we could add methods to format the output data.

2.3.2 Rationale for Strategy No. 3

By creating abstract data types, the implementation details are hidden from other parts of the software system. This helps to isolate future changes to data aspects of the system. In other words, the scope of the changes required to modify or extend data aspects of the system is limited to abstract data types rather than scattered across the software system. This leads to a system that is easier to maintain.

Another advantage of encapsulating related data inside objects is that we can pass them to different layers of the system as a whole. This prevents the need to pass numerous fine-grained data values around. We can think of different layers of the software architecture as consumers of these encapsulated data. In most cases, they need all the data corresponding to the numerical problem together to satisfy their computational needs. Therefore, it is desirable to deliver all the required data in one pass. Moreover, the object that encapsulates the data may provide a number of useful operations, which can be used by other layers consuming that object.

Abstract data types simplify the interfaces or signatures of the methods. The methods accept all the required data in the form of an object as their input parameter rather than numerous primitive data types. Yet another advantage of creating abstract data types is that software systems becomes more self-documenting, which can be invaluable for understanding and maintaining them. Of course, this requires the use of appropriate naming conventions for abstract data types.

2.3.3 Implementation of Strategy No. 3

A single layer of a numerical can be further divided into a number of other relevant sub-layers or partitions. We will create a layer called the data object layer inside the input layer of each of the subsystems to accommodate abstract data types that capture the input data of the system. This can be done using nested packages in Java. As an example, for the nonlinear sybsystem, we will have a package such as

numerical.input.nonlinear.dataobjects

Similarly, we will create a package called

numerical.output.nonlinear.dataobjects

that contains the output data objects. For each numerical problem to be solved, we will create input and output Java classes to represent the corresponding data objects. The following code listing shows a class for input abstract data type called QuadraticEquation that represents equation (1):

```
public class QuadraticEquation {
   private double b;
   private double c;
   public QuadraticEquation(double b, double c)  {
            this.b = b;
            this.c = c;
   }

   public double getB() {
      return b;
   }

   public void setB(double b) {
      this.b = b;
   }

   public double getC() {
      return c;
   }
```

```
    public void setC(double c) {
        this.c = c;
    }

    public double computeDiscriminant() {
        double d = b*b - 4*c;
        return d;
    }

    public boolean hasComplexRoots() {
        double d = computeDiscriminant();
        if ( d < 0 )
            return true;
        else
            return false;
    }
}
```

As an example of an output data object, the following code listing shows a class called QuadraticRoots that represents the roots of the equation (1):

```
public class QuadraticRoots {
    //r1 and r2 represent the two real roots
    private double r1;
    private double r2;
    private boolean isComplex;

    //p +/- iq represents the two complex roots
    private double p;
    private double q;

    public QuadraticRoots() {
    }

    public QuadraticRoots(double r1,
                          double r2,
                          double p,
                          double q,
                          boolean isComplex) {

        this.r1 = r1;
        this.r2 = r2;
        this.p = p;
        this.q = q;
        this.isComplex = isComplex;
    }

    public boolean isComplex() {
        return isComplex;
    }

    public void setComplex(boolean isComplex) {
        this.isComplex = isComplex;
    }
    public double getR1() {
        return r1;
    }
    public void setR1(double r1) {
        this.r1 = r1;
    }
    public double getR2() {
        return r2;
    }
    public void setR2(double r2) {
        this.r2 = r2;
    }

    public double getP() {
        return p;
    }

    public void setP(double p) {
        this.p = p;
    }

    public double getQ() {
        return q;
    }

    public void setQ(double q) {
        this.q = q;
    }

}
```

Figure 1 depicts the structure of the numerical software system resulted from applying the three architectural strategies discussed so far. Table 1 shows the architectural layers as depicted in Figure 1.

Table 1: Architectural Layers and Components

Layer	**Architectural Components**
1	Numerical System/Package
2	Architectural Layers
3	Numerical Subsystems
4	Partitions and Abstract Data Types

An alternative to the structure depicted in Figure 1 is to swap layers 2 and 3. This way, the numerical

package will be partitioned into a number of subsystems, each containing sub-packages that correspond to architectural layers. In the original structure of Figure 1, the numerical subsystem packages are repeated under the three main architectural layers of input, computation, and output, whereas in the alternative design, the architectural layer packages are repeated under each of the numerical subsystem packages. Both designs provide the benefits of complexity reduction and isolation of changes.

2.4 Strategy No. 4

2.4.1 Using a Uniform Mechanism for Data Validation

In strategy No. 3, we created abstract data types to capture and represent input data. The idea here is that each of these data objects should be able to validate the data encapsulated inside them. This can be accomplished by adding a validation operation to every single abstract data type. All the required validation logic will be placed inside this method, which will further contribute to the achievement of isolation of change as a design goal.

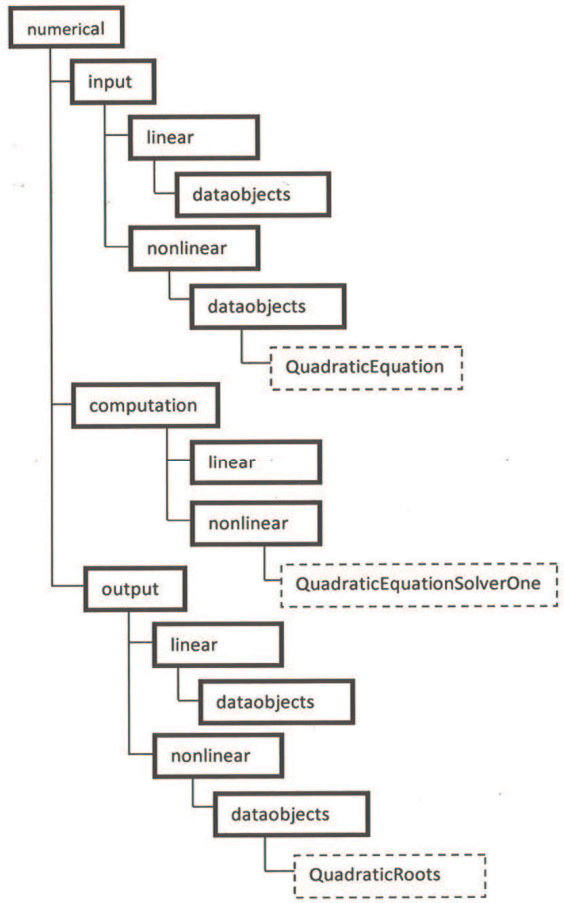

Figure 1. The Structure of the Numerical Software System After Applying Strategies 1, 2, and 3.

2.4.2 Rationale for Strategy No. 4

Input data validation is an important aspect of any software system. Numerical software is not an exception. Typically, data validation code is scattered throughout the entire software system, making it hard to maintain. Implementing a uniform mechanism for validating all user inputs leads to a software system that is easier to develop, change, and understand. This way, the developers of the system can always assume that all the data validation code for a specific use case or feature of the system is located in one single place, and that single location is an operation with a predefined name inside the abstract data type that captures the data of that specific use case. Being able to make such assumptions about a software system drastically reduces the complexity of the system. In a typical software system where validation code is scattered throughout the entire system, we cannot make any assumptions and therefore the only way to make a change to the data validation parts of the system is to search and read the source code and try to understand large parts of the system in order to locate the subset of the source code that is responsible for validating the input data of a specific use case of the system.

Note that we rely both on the structure of the system and naming conventions to facilitate source code navigation. This combined use of structural and naming rules allows the software engineer to reason about the properties of the numerical system.

2.4.3 Implementation of Strategy No. 4

Each abstract data type has a number of relevant operations that provide useful services regarding the underlying data structure. Depending on the concept represented by the abstract data types, a different set of such utility methods might be defined for a data type. Regardless of the conceptual differences among these abstract data types, they all share one common property - the need for validating the data contained within the abstract data types. Because we want to enforce a uniform implementation of data validation among all these abstract data types, we will create an interface called *validatable* in the

input package and require all abstract data types to implement this interface. The following code listing shows the *validatable* interface:

```
public interface Validatable {
      public void validate( ) throws
                              DataValidationException;
}
```

The *validate()* method notifies its callers of any errors found during the validation process by throwing an exception of type *DataValidationException.* The following code listing shows a partial implementation of the *validatable* interface by the *QuadraticEquation* class. Notice how all the validation logic is put in one single place: the *validate()* method:

```
public class QuadraticEquation implements
                                    Validatable{
      private double b;
      private double c;
      private String inputData[];

                .
                .
                .

      public void validate() throws
                          DataValidationException {
        try {
           double tempB =
           Double.parse Double(inputData[0]);
          this.b = tempB;
       }catch( NumberFormatException e){
        throw new DataValidationException("Error
        in   input data: Coefficient b must be a valid
       double number.");
      }

      try {
        double tempC =
        Double.parseDouble(inputData[1]);
        this.b = tempC;
     }catch( NumberFormatException e){
        throw new DataValidationException("Error
        in input data: Coefficient c must be a valid
       double number.");
    }
  }
}
```

As shown in the above code snippet, the user inputs (the values for the coefficients b and c) are read and stored as string variables in the *QuadraticEquation* data type. A call to the *validate*() method triggers the validation process. The validate() method checks the format of the string values to see if they are valid double number. If there are no validation errors and the strings can be successfully converted into double-precision floating point numbers, these converted numbers will be used to set the values of the *b* and *c* coefficient attributes in the *QuadraticEquation* data object, otherwise a *DataValidationException* is thrown. The code listing for the exception class follows:

```
public class DataValidationException extends
                                        Exception {
      Public DataValidationException(
                                String errorMessage){
          super(errorMessage);
      }
}
```

2.5. Strategy No. 5

2.5.1 Reify Numerical Algorithms and their variations into Separate Objects and make them Interchangeable: The Strategy Design Pattern

Numerical algorithms and their variations must be treated as first-class citizens in building numerical applications. This can be achieved through a technique called reification – the turning of important concepts of a domain into separate objects. One possible way to encapsulate each individual numerical algorithm inside a separate class is to use the "Strategy" design pattern [5]. Figure 2 depicts the classes that participate in the Strategy design pattern using a UML diagram.

2.5.2 Rationale for Strategy No. 5

The reification of numerical algorithm into strategy objects allows them to change independently from the clients that use them (the source code that calls the methods implementing the numerical algorithms). Using the Strategy design pattern to encapsulate these algorithms also makes them interchangeable. For instance, depending on the situation, we might want to solve the quadratic

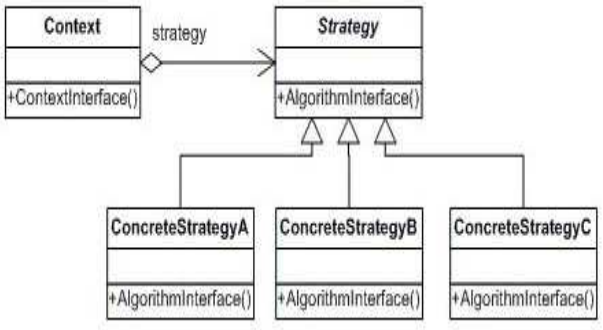

Figure 2. The Strategy Design Pattern. Figure Adopted from [7].

equation (1) using algorithms (1) and (2) shown below (both algorithms adopted from [6]). The client code does not need to change to solve the numerical problem using a different algorithm. The object encapsulating the appropriate algorithm is passed to the client, and the client code simply calls its *compute()* method. If, depending on the conditions of the numerical problem, we want to solve the problem using a different algorithm, all we need to do is to pass a different object (i.e., an object that uses another algorithm to solve the problem) to the client. Changing the algorithms used to solve numerical problems simply becomes a matter of passing different objects to the client. The client code itself does not change at all.

Algorithm (1) for Solving Quadratic Equations

$d = b^2 - 4c$
if $d \geq 0$
  $e = \sqrt{d}$
  $r_1 = \frac{-b+e}{2}$
  $r_2 = \frac{-b-e}{2}$
else
  $p = \frac{-b}{2}$
  $q = \frac{\sqrt{-d}}{2}$

Algorithm (2) for Solving Quadratic Equations

$d = b^2 - 4c$
if $d \geq 0$
  $e = \sqrt{d}$
  if $d \geq 0$
    $r_1 = \frac{-b+e}{2}$
  else
    $r_2 = \frac{-b-e}{2}$
  $r_2 = \frac{c}{r_1}$
else
  $p = \frac{-b}{2}$
  $q = \frac{\sqrt{-d}}{2}$

2.5.3 Implementation of Strategy No. 5

The strategy design pattern consists of three different types of participants: the strategy, the concrete strategy, and the context. The following code listing shows the strategy interface. All classes encapsulating a numerical algorithm must implement this interface. This leads to a uniform implementation of all numerical algorithms.

```
public interface NumericalAlgorithm {
    public Object compute(Object input);
}
```

The argument passed to the *compute* method and its return type are of generic type *Object*. The caller of this method, which plays the role of the context in the strategy design pattern, will prepare an input data object of the appropriate type and pass it to the compute method of the object that encapsulates the desired numerical algorithm. The *compute* method will perform the necessary calculations to compute the results and then will create an output data object that contains the result of the computation and will return it to the caller.

The method that implements the *compute* method must cast the *input* argument to the appropriate data object type. Similarly, the caller

method should cast the object returned by the *compute* method to the appropriate output data object type. The following code listing shows a typical implementation of this interface. The *QuadraticEquationSolverOne* class encapsulates the Algorithm (1) for solving quadratic equations.

```
public class QuadraticEquationSolverOne
                implements NumericalAlgorithm {

    public Object compute(Object input) {

        double d,e;
        double r1,r2;
        double p,q;
QuadraticRoots quadraticRoots = new
QuadraticRoots();

//access the input
double b = ((QuadraticEquation)input).getB();
double c = ((QuadraticEquation)input).getC();

    //do the calculations
    d = b*b - 4*c;
    if ( d >= 0 ){
        e = Math.sqrt(d);
        r1 = (-b + e)/2;
        r2 = (-b - e)/2;
        quadraticRoots.setR1(r1);
        quadraticRoots.setR2(r2);
        quadraticRoots.setComplex(false);
    }else {
        //complex roots
        p = -b/2;
        q = Math.sqrt(-d)/2;
        quadraticRoots.setP(p);
        quadraticRoots.setQ(q);
        quadraticRoots.setComplex(true);
    }

    //return the result
    return quadraticRoots;

    }
}
```

Similarly, we could encapsulate Algorithm (2) in a class called *QuadraticEquationSolverTwo*. Depending on the situation, we would pass one of the two objects to the client code.

2.6 Strategy No. 6

2.6.1 Creating a Single Access Point for Handling All the Input-Related Functionality for a Subsystem

High cohesion, modularity, isolation of change, and simplification of use are the main design goal behind this strategy. These all can be achieved by isolating all the input logic of a subsystem in a single module and creating a single point of access to that module. Each method in this module will be responsible for handling the input of one type of problem in that subsystem. This includes presenting the user interface, interacting with the user in order to capture the input data and instantiating and populating the related input data object.

2.6.2 Rationale for Strategy No. 6

Applying this strategy makes it possible to limit all future changes and extensions to the input-related aspects of a subsystem to a single module. In general, creating modules that are focused on one single aspect of a subsystem makes it possible to quickly locate the places in the source code that need to be changed in response to a change request or defect fix, which facilitate maintenance. Also, providing a single entry point to a module reduces the complexity of the system because it provides a unified way for all clients to access the module.

2.6.3 Implementation of Strategy No. 6

As an example, we can apply this strategy to the nonlinear subsystem in our hypothetical numerical software system by creating a class called *NonlinearInputHandler* in the *numerical.input.nonlinear* package. In order to provide a single access point to the services of this class, we will apply the singleton design pattern [5]. The sole instance of this class can always be reached by calling its static *instance*() method. Figure 3 depicts the Singleton design pattern.

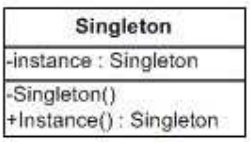

Figure3. Singleton Design Pattern. Figure Adopted From [7].

The following code listing shows a sample implementation of this class. The *handleQuadraticEquationInput*() method in this class handles all the input-related logic for the quadratic Equation (1).

```
public class NonlinearInputHandler {
    private static NonlinearInputHandler instance
                                            = null;
    BufferedReader inputReader = new
                            BufferedReader(new

    InputStreamReader(System.in));

    private NonlinearInputHandler() {

    }

public static NonlinearInputHandler Instance() {
    if ( instance == null ){
        instance = new NonlinearInputHandler();
    }

        return instance;
}
public QuadraticEquation
      handleQuadraticEquationInput() {
    String[] coeficients = new String[2];

    try {
        System.out.println("Enter b: ");
        coeficients[0] = inputReader.readLine();
        System.out.println("Enter c: ");
        coeficients[1] = inputReader.readLine();
        } catch (IOException ex) {
            System.err.println("error reading
                            from input stream.");
            System.exit(-1);
        }

        QuadraticEquation equation = new
                QuadraticEquation(coeficients);
        try {
            equation.validate();

        }catch(DataValidationException e) {
            System.out.println("Error in Input : "
                            + e.getMessage());
            System.exit(-1);
        }

        return equation;
  }
}
```

2.7 Strategy No. 7

2.7.1 Creating a Single Access Point for Handling All the Output-Related Functionality for a Subsystem

This strategy is similar to the previous one, but here it is applied to the output of the system. The goal is to isolate all the output logic of a subsystem in a single module and create a single point of access to that module. Each method in the module will be responsible for handling output of one type of problem in that subsystem. This includes formatting and presenting the result of the calculations.

2.7.2 Rationale for Strategy No. 7

All the benefits mentioned for the Strategy No. 6, which was related to the input aspects of the system, also applies here to the output aspects.

2.7.3 Implementation of Strategy No. 7

As an example, we can apply this strategy to the nonlinear subsystem in our hypothetical numerical software system by creating a class called *NonlinearOutputHandler* in the *numerical.output.nonlinear* package. In order to provide a single access point to the services of this class, we will apply the singleton design pattern so the sole instance of this class can always be obtained by calling its static *instance*() method. The following code listing shows a sample implementation of this class. The *handleQuadraticEquationOutput*() method in this class handles all the output-related logic for the quadratic Equation (1).

```
public class NonlinearOutputHandler {
 private static NonlinearOutputHandler instance =
                                            null;

    private NonlinearOutputHandler() {

    }

 public static NonlinearOutputHandler Instance() {
  if ( instance == null ){
    instance = new  NonlinearOutputHandler();
 }

        return instance;
```

```
    }

public void handleQuadraticEquationOutput(
                        QuadraticRoots output) {
  //print the result
  if ( output.isComplex() ){
    System.out.println("The Quadratic Eqution Has
    Complex Roots:");
    System.out.println("output - p = " +
                                    output.getP());
System.out.println("output - q = " + output.getQ());

}else {
 System.out.println("output - r1 = " +
                                   output.getR1());
 System.out.println("output - r2 = " +
                                   output.getR2()
        }
    }
}
```

2.8 Strategy No. 8

2.8.1 Compose Numerical Services by Creating a Façade For Each Subsystem of the Numerical Software System

This strategy represents each subsystem of the numerical software system to the clients of the subsystem, using a convenient higher-level interface. The Façade design pattern [5] is the right choice here. Each method in the façade class is responsible for solving one type of numerical problem. This method delegates responsibilities to various modules within the subsystem to carry out the requested task. Figure 4 depicts the Façade design pattern.

The previous strategies helped us to create a number of different modules each focusing on a single dimension of a numerical problem such as input, validation, computation, and output. Numerical problems, like problems in most other domains, are multi-dimensional. In order to perform useful computations, the parts that deal with individual problem dimensions should be put together to form numerical procedures. It is the responsibility of the methods of the façade class to compose numerical services from different modules to solve numerical problem. The methods of the façade class know which module is responsible for which dimension of the numerical problem.

2.8.2 Rationale for Strategy No. 8

The façade class separates the clients from the complexities of the underlying modules in a subsystem by providing a higher-level interface to those modules. The client code does not need to have the knowledge of the underlying modules. The façade methods are the only places where this knowledge is kept.

Note that while the previous strategies created independent and isolated service parts, Strategy No. 8 composes complete numerical services by sequencing and coordinating these service parts. This architecture provides three main benefits: First, the individual service parts can be reused for creating new services. Second, since service parts have minimum interaction with other parts of the system, they can be easily replaced, extended, or changed if the needs arise. Finally, they facilitate the overall comprehension of the system as each service part performs a well-defined and single task with a minimum level of dependence to other parts of the system.

2.8.3 Implementation of Strategy No. 8

As an example we can create a façade class to represent the nonlinear subsystem. We will create a class called *NonlinearFacade* in the *numerical* package. The *solveQuadraticEquation()* method in this class demonstrates how the responsibility of addressing various dimensions of solving the quadratic equation (1) is delegated to the appropriate modules within the numerical software system.

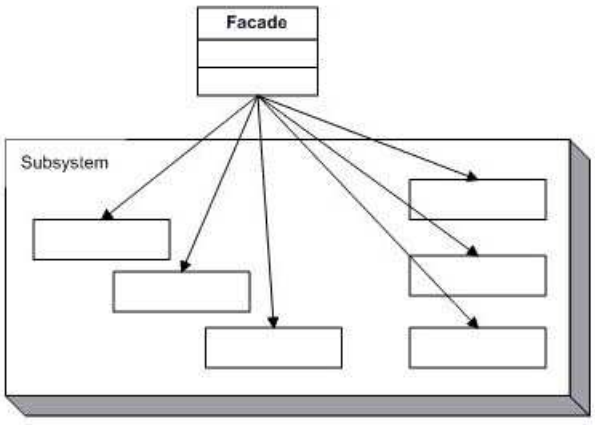

Figure 4. The Façade Design Pattern. Figure Adopted From [7].

```
public class NonlinearFacade {
     private static NonlinearFacade instance = null;

    private NonlinearFacade() {

     }

     public static NonlinearFacade instance() {
        if (instance == null ){
                instance = new NonlinearFacade();
          }
          return instance;
    }

//Servive method solving quadratic equations
public void solveQuadraticEquation() {
  /Step 1: capture and validate the input
 QuadraticEquation input =

NonlinearInputHandler.Instance().
                    handleQuadraticEquationInput();

 //Step 2: create an instance of the desired
 //algorithm to solve the equation
NumericalAlgorithm quadraticSolverOne = new
                    QuadraticEquationSolverOne( );

//3. Run the selected algorithm
QuadraticRoots output =

(QuadraticRoots)quadraticSolverOne.
                                    compute(input);
//Step 4: Dislay the results
NonlinearOutputHandler.Instance().
        handleQuadraticEquationOutput(output);

}
```

Here the *solveQuadraticEquation* () method is the only method that has the complete knowledge of modules and their responsibilities required to solve a quadratic equation. This knowledge includes sending a request to the appropriate module to capture the input data, validating the input data, creating an instance of the appropriate object that encapsulates the desired algorithm, capturing the result of performing the computation in the appropriate output object, and finally, sending a request to the appropriate module to display the results.

2.9 Strategy No. 9

2.9.1. Define Strict and Fixed Rules For Inter-Module Coupling

In general, there are three possible ways for a module A to be coupled with another module B in an object-oriented system: by argument, by instantiation, and by method call (We do not consider the case where a class is coupled with another class by inheritance). In argument-based coupling, an object of type A is passed to a method in class B. In instantiation-based coupling, an object of type A is instantiated inside the class B or a method of class B. In call-based coupling, an object of type A calls a method on an object of type B.

We used a layered architecture to partition a numerical software system into three main layers including the input layer, the output layer, and the computation layer. We further divided these main layers to a number of subsystems, each corresponding to a category of numerical problems such as linear and nonlinear problems. In addition to these decomposition rules, we might further define the following coupling or inter-module communication rules:

- The input module of a subsystem must not be coupled with any other subsystem or module within the same subsystem. It is a self-contained module representing one single dimension of the numerical subsystem. This means that the input module does not depend on any other module for its functionality. Other modules depend on the input module for input-related services.
- The output module of a subsystem must not be coupled with any other subsystem or module within the same subsystem. It is a self-contained module representing one single dimension of the numerical subsystem. This means that the output module does not depend on any other module for its functionality. Other modules depend on the output module for output-related services.
- The computation module is coupled with the input module by argument. It receives an input data object as an argument. It is also coupled with the output module by instantiation. The computation module returns an output

data object as the result of the computations.

- The subsystem façade is coupled with both the input and output modules by way of method call. It is also coupled with the computation module both by instantiation, and by method call.
- Communication between two different subsystems is only done through their facades. This means that the methods of a subsystem façade are the only places where can be freely coupled with other parts of the system. This way the modules of a subsystem have the lowest degree of coupling with other subsystems, and the subsystem façade is the only place in a subsystem that has the knowledge of inter-subsystem couplings. This gives us more control over coupling throughout the software system.

Table 2 depicts the Module Coupling Matrix (MCM) *within* each subsystem (i.e., the architectural components belonging to a subsystem) of the numerical architecture presented in this paper. An MCM can be used to express the coupling rules for all or parts of a system. The columns and rows of this matrix represent the modules that comprise each subsystem including the input, output, computation, and the façade. The symbols A, I, and C in the intersection of a row i and a column j indicate that the module i is coupled to module j through argument, instantiation, or method call respectively. A × symbol indicates that the module i is not allowed to couple with module j. The ✓ symbol indicates a "Don't Care" situation, where module i can be freely coupled with module j in all possible ways. The entries on the main diagonal of the matrix in Table 2 are all ✓ symbols indicating that each module can be freely coupled with itself within a subsystem (i.e., self-referencing is permissible). It is important to note that all communication *between* different subsystems is through subsystem facades.

Table 2: Intra-module coupling within a subsystem

	IN	**OUT**	**CMP**	**FCD**
IN	✓	×	×	×
OUT	×	✓	×	×
CMP	A	I	✓	×
FCD	C	C	I, C	✓

2.9.2 Rationale for Strategy No. 9

Following a strict and fixed set of rules for coupling modules gives us more control over coupling in a system. Systems with a high degree of coupling are hard to understand and maintain. The reason is that in tightly-coupled systems every part of the system can be freely coupled with some other part of the system, which increases the complexity of the system. All the strategies we have discussed help to minimize the coupling. It is impractical to completely remove the coupling between modules; modules need to interact with each other in order to perform useful tasks. However, the key to a good design is to minimize the degree of coupling so that changes to a module do not propagate to other modules. In places where coupling is not avoidable, we enforce strict rules so that each module can interact with other modules using predefined coupling patterns. This makes it easier to understand the system because one does not need to review the source code to discover how modules are connected together. Modules are following fixed patterns for their interactions.

2.9.3 Implementation of Strategy No. 9

If we look at our implementation of the quadratic equation solver, it can be seen that we have followed the coupling rules defined in strategy No. 9. For example, the input and output packages do not refer to anything outside their packages. The computation package, however, is dependent on both input and output data objects. This dependence is inherent and cannot be avoided. A numerical algorithm needs to perform its calculations on some input data and produce some output data. The key point here is that, through using the strategy design pattern, we have created a fixed coupling pattern between the computational classes such as the *QuadraticEquationSolverOne* and their input and output classes (e.g., the *QuadraticEquation* and *QuadraticRoots*).

We decided to implement the *compute* method of the *NumericalAlgorithm* interface by always accepting an input data object and returning back an output data object. This implies a fixed coupling pattern. Also, the solve*QuadraticEquation ()* method of the *NonlinearFacade* class has couplings with input, computation, and output modules. This method demonstrates a fixed sequence of coupling. It first calls a method on the singleton instance of

the *NonlinearInputHandler* class to prepare the input. It then creates an instance of the appropriate algorithm class (the *QuadraticEquationSolverOne class*), and then triggers the algorithm by providing the input data object. It then provides the output data object resulted by the algorithm to the appropriate method in the sole instance of the *NonlinearOutputHandler* class. We can expect to see the same coupling sequence in all of our façade methods. This repeating pattern of coupling makes it easier to understand the code.

3 The Underlying Design Principles

Table 3 summarizes the nine architectural high-level decisions - or strategies - that were used to build an architectural foundation for the domain of numerical software systems along with their rationale and implementation techniques. As it can be seen from this table, there are common design principles behind all these various architectural decisions. In particular, the following four principles are the key to high-quality software architectures and consequently high-quality software systems:

1. Decomposition
2. Isolation
3. Unification
4. Fixation

Decomposition of systems into smaller units such as layers, subsystems, use cases, features, and the like in conjunction with the isolation of service parts within these units help to reduce the overall system complexity. The unification of service mechanisms across a system (i.e., uniformity of design) along with the regulation of the ways in which system parts – or components – interact (i.e., as a set of fixed and predefined interaction patterns) further reduce system complexity. These principles can be encoded as a set of design rules governing software systems. Conformance of systems to these principles facilitates software comprehension and evolution. The consistent application of these principles also facilitates reasoning about the properties of software systems, which can be invaluable in performing various software engineering tasks.

4 Conclusion and Future Work

In this paper, we presented an architectural foundation for developing a family of numerical software systems. Numerical systems are viewed as multi-dimensional systems and as such each dimension of numerical problems is addressed through a mechanism devised in the software architecture. The knowledge of translating numerical software requirements into a set of architectural components and their relationships is captured as a series of architectural rules.

The reuse of the proposed architecture increases development productivity while improving the quality of the software system. This is because the time-consuming design process is not repeated for developing every single numerical system. Instead, the design is captured in the form of a domain-specific architecture and is reused for developing new numerical software systems. Moreover, since the strategies used in developing this architecture are based on software engineering best practices, they improve the quality of the resulting system.

The work presented in this paper can be continued in many directions. The approach presented in this paper can be used to develop domain specific architectures for other system domains such as enterprise information systems or embedded systems. This will allow us to better understand and identify the common and contrasting architectural patterns across domains.

Another interesting research avenue is to devise a language to express the design rules or regulations that underlie a system. Such a language can then be used to automate the verification of conformance of systems to their underlying design rules. Such a language can also be used to support automatic generation of architectural code.

Yet another interesting research question is how the architectural components in a rule-governed system trace back to their higher level software requirements? A follow up question is whether we can define fixed rules (i.e., traceability rules) to describe the relationships between requirements and their corresponding components in the design of systems. We are currently researching these problems. Some results have been reported in [1], [2], and [3]. a Advances in these areas can pave the way for automatic requirements allocation, which in turn can lead to more systematic approaches to software development.

References:

[1] Ghazarian, A., "*Coordinated Software Development: A Framework for Reasoning about Trace Links in Software Systems"*, Proceedings of the IEEE 13th International Conference on Intelligent Engineering Systems (INES 2009), Barbados, April 2009.

[2] Ghazarian, A., "*A Matrix-Less Model for Tracing Software Requirements to Source Code"*, International Journal of Computers, ISSN: 1998-4308, Issue 3, Volume 2, 2008.

[3] Ghazarian, A., "*Traceability Patterns: An Approach to Requirement-Component Traceability in Agile Software Development"*, Proceedings of the 8th WSEAS International Conference on Applied Computer Science (ACS'08), Venice, Italy, November 2008.

[4] Tanenbaum, A. S., "*Computer Networks*", Third Edition, Prentice Hall, 1996.

[5] Gamma E., Helm R., Johnson R., Vlissides J., "*Design Patterns: Elements ofReusable Object-Oriented Software*", Addison-Wesley, 1995.

[6] Miller W., "*The Engineering of Numerical Software*", Prentice-Hall, 1984

[7] Design Patterns, http://www.dofactory.com/Patterns/Patterns.aspx

Table 3: The Architectural Decisions, the Rationale Behind Them, and Their Implementation Techniques

Strategy	Design Goal/Rationale	Implementation
1: Layered Architecture	- Isolation of Change - Improving Comprehension - Reduction of Complexity	Decomposition Through Packaging Mechanisms
2: Subsystem-based Decomposition	- Isolation of Change - Improving Comprehension -Reduction of Complexity	Decomposition Through Packaging Mechanisms
3: Data Abstraction	- Isolation of Change - Data Hiding - Simplified Operations (Simplified Method Signatures)	Creation of Separate Classes for the Main Concepts of the Domain
4: Uniform Data Validation	- Isolation of Change -Improving Comprehension	Use of interfaces to enforce implementation uniformity
5: Reification of Numerical Algorithms	- Isolation of Change - Prevent Changes to Client Code	Creation of Separate Classes for the Main Concepts of the Domain (e.g., Using the Strategy Design Pattern to Create a Strategy class for each numerical algorithm)

6: Single Entry Points to Input-Related Functionality	- Ease of Use (Simplification of Client Code) - Isolation of Change	Singleton Design Pattern
7: Single Entry Points to Output-Related Functionalities	- Ease of Use (Simplification of Client Code) - Isolation of Change	Singleton Design Pattern
8: Single and simplified Entry Point to Subsystems	- Simplification of Client Code - Ease of Use - Isolation of Change - Reuse of Services - Controlled Inter-Subsystem Coupling	Façade Design Pattern
9: Fixed Inter-Module and Inter-Subsystem Coupling Rules	- Reduction of Complexity - Improving Comprehension	Enforcement of Predefined and fixed Coupling Rules