Reusable Software Components in Legacy Codes

Chaman Singh Verma and Ling Liu

PROJECT SUMMARY

Large software system evolve over the years and become increasingly complex. A large portion of the system may consists of legacy codes, which are mostly written in non-OOP languages like FORTRAN, C, COBOL etc, which are inflexible and difficult to extend. It is infeasible to rewrite the entire software with new design rules,or in new programming languages. Reusing is perhaps the best strategy to handle complexities in software development. Design patterns and generic programming are two very powerful techniques which emphasis upon reusing software. We believe that design patterns could be applied to develop very flexible systems in which old design and legacy codes could be hidden behind in some classes exposing only the new design to the end users. This will allow old design to gradually replace with some new design without breaking the system at any moment of time.

We study our new system with Mesh Generation Software which is a good representative of the other applications which have been evolved over the years. It uses legacy codes written in FORTRAN, C and Python. Using design patterns in C++, we should be able to demonstrate that the resulting system is much more powerful and flexible.

Intellectual Merit

Intellectually it is a challenging task to develop a software which is flexibility to adapt for future requirements. Also, now a days, software is developed by teams rather than individuals (possibly in different languages) and it is non-trivial task to integrate the pieces into single system.

The aim of this project is to explore a new simplified approach which can expose the system design at very high level which all the developers can understand with ease. Legacy codes or not-so-well designed modules could be hidden behind some design patterns which provide a common vocabulary for the software developers will help rapid prototyping of system. The resulting system will be much more flexible and loosely coupled to allow modification/replacement of the existing modules without breaking it.

We will experiment with our approach with a Mesh Generation Software (in-house developed code) and redesign the entire software with design patterns and generic programming and come out with a system which will be far easier to understand and flexible.

Broader Impact

As the P2P system and Grid computing are becoming increasing popular way of sharing information and computing resources, it is important that we develop software which are easy to understand , easily reusable, and adaptable to the changing environments or requirements. We need to develop system which has almost flat learning curves. Object oriented programming combined with design patterns and generic programming is the perhaps the most simplified approach till today, which need to exploited for developing new systems. Our research will give insight into this exciting topic.

Reusable Software Components in Legacy Codes

Chaman Singh Verma and Ling Liu

PROJECT DESCRIPTION

Most of the large software system evolve over the years and become increasing complex. The legacy part of the system, constitutes a considerable portion of the overall new system. As of now, these legacy codes were written in Fortran, C, Cobol etc which were functional programming languages which difficult to maintain and extend in their functionality. Object-Oriented programming(OOP) enable us to organize the system into objects by information hiding, encapsulation, polymorphism and other techniques which results in increased flexibility compared to non-OO systems.

Object-Oriented programming, per se, does not solve all the problems in software development, in fact, it could be even harder. In order to use OOP, one must search for pertinent objects, factor them into appropriate classes of the right granularity, define class interfaces and inheritance hierarchies, and compose them into single system. The added flexibility also allow us to create spaghetti system, create monolithic classes with lots of functions, This was not the essence of object orientation.

Software reuse is identified as one of the best strategy to handle complexities associated with development and maintenance of complex software. Design patterns and Generic programming are two most powerful emerging techniques which has changed our perspectives about software reuse. Generic programming allows developers to reuse the software by parameterizing the datatypes and Design Patterns stress upon decoupling the system for increasing flexibility. Many application domain (example GUI builders, network communications libraries) have been greatly benefited by design patterns, but their usefulness in other areas needs further studies with real applications.

We experiment restructuring with design patterns into *3D Mesh Generation Software* which is a good representative of other applications and fairly complex to address many issues with the legacy system. It involves components which were written in FORTRAN, c, python and C++ and uses MPI and Pthreads. This application have potential for reusing both Generic programming and Design patterns in a single system,

This report is organized as follows. First section describes the scientific objective of our work, second section descrive the impact of the work for future, third section describes the background information of the techniques which we will be using in our new system, fourth section outlines our research plan, fifth section briefly explains our experimental testbed and sixth section briefly summarizes our work.

1 Scientific Objectives

Although, design patterns and generic programming have been very successful in new software development, we are not aware of any work from other research groups, which have studied the effectiveness of these techniques in restructuring the legacy codes.

Any scientific method needs to experiments with some objective metrics. Unfortunately, there are no well established or broadly accepted metrics for the software till today. Still, we should be able to characterize the new system with the following properties and answer the following questions.

- **Flexibility** A new system should be flexible enough to adapt to new functionality and unanticipated changes in the requirements and the environments. More flexible systems are generally loosely coupled. **Question:** Is the new system with the design pattern more flexible than the old system ?

- **Maintainability** A good maintainable software is characterized by the easiness of *Removal of Defects* and *Enhancement by adding and removal of the component*. Up to what extend, design patterns helps

end users in this regard ?
Question: Is the new system easy to modify and more adaptable then the old system ?

- **Efficiency** Efficiency is measured in both machine which uses the resources and the person who develop and use the system. Any new systems should not degrade the existing system efficiency.
  - System degradations could be caused by increased indirections, duplications, runtime identifications, excessive functional call overheads etc.
  - Human efficiency involves learning and adapting to the new system and it should be a natural process. Any shock or drastic changes only bring reluctance and opposition for use.

  Questions 1 : How easy it will be to restructure the legacy code with the design patterns ?
  Questions 2 : Is the design incremental ?

- **Completeness** Are the existing design patters sufficient enough to provide to capture the commonly used software designs.
  Question Can we say, Yes, this is the pattern I was looking for ?

- **Preciseness** A good design pattern should describe the intention and functionality of the pattern in an unambiguous way.
  Question: Why did you use this pattern and why not this ?

- **Quality Improvement** Using the Fenton criteria, can we categorically say that overall system has improved with the use of reusable components.

- **Better Organization** The new systems should have better *Separation of Concerns* and easy to understand. A better organized systems allows learning at different hierarchies. In these systems, the complexities are hidden behind very few and simple interface functions through which system interact with other components. **Question:** Is the new system better organized for future adaptation ?

- **New Design Patterns** New applications and environments will require us to explore new systems. We may have to explore domain specific patterns which will be most efficient then the general purpose solutions. **Question:** Do we need more ?

2 Impact of Research

It is hard to predict, specially future. But there are certain trends which are unlikely to reverse or decrease. Open source initiatives by organizations is based on the assumption that when software developers can read, redistribute and modify the source code for better functionality and design, the software evolves as any other biological entities which are far superior than their ancestors. This concept reinforces the requirements for simple reusable components.

- **Grid or P2P Computing** Both Grid and P2P (Peer to Peer Systems) are emerging technologies and have changed our views the way we will be tackling complex problems in future. These technologies will enable large-scale aggregation and sharing of computational, data and other resources using Internet Infrastructure. Grid technology is based on the idea that few centers will have large resources to cater the needs of user, and P2P a decentralized system will rely on the resources and power of millions of computers individual possess and available for free use using Internet for collaboration.

- **Collaborative Computing** As our expectations from software solutions is increasing, there is no way one individual, one organization, or one university can boast of competitive in all aspects of science or engineering solutions. In the past, softwares more or less were individual assets, and the programmers were Creator and Destroyers for their own creations. This model is no more valid and more and more universities and organizations will be working in large group to create multidisciplinary applications, which will have profound impact on our lives.
- **Directions for Wrapping legacy codes** The amount of legacy codes accumulated over the years is so large and so much investment already done, that it is practically difficult to rewrite the entire systems with modern styles. Our research may help (success or failure) in evolving a middle path where legacy codes are wrapped with new systems if they look new and flexible.

3 Reusable Software Components

3.1 Characteristic of Legacy Codes

Whether we like or not, legacy codes will always constitute a large part of the software, and this hard reality is unlikely to change. From our experience, we find the following characteristic of legacy codes.

3.1.1 Advantages of Legacy System

- They evolve over a large period of time sometimes more than 10-15 years of more (Example CAD, Unix operating system)
- They are trustworthy in their limited functionality.
- Large number of software uses them in their system and therefore, their users base is high.
- Since most of software were written in low-level programming languages relative to modern high level languages and therefore they are devoid of templates, virtual functions, polymorphism, run-time identifications etc, legacy codes can be considered emanating from assembly language programming and therefore have better performance.

3.1.2 Disadvantages of Legacy System

- These are difficult to maintain and extend.
- These are in general huge and require large investment of money and human efforts.
- Older system have lots of duplications of the code for the same functionality with different datatypes.
- Lack of *Separation of Concern* paradigm in these codes, remind us of spaghetti.
- They do not take advantage of modern processors design. Most of the codes were written when thread programming were in its infant stage and distributed computing are non-existent.
- They have little or no concern about security of the system.

Software component as defined by Hongji Yang in his textbook *Successful Evolution of Software Systems* is

A coherent and configurable software package, independent of the applications in which it has been used, with well defined interfaces in the different contexts to interact and communicate with other components in order to compose a large system.

A component as per definition could be simple macros, small functions, classes, or even entire software package. It is independent of programming language as long as there is some way to invoke the functions from other programming languages.

Software reuse have been very successful in many areas. People use compilers, system libraries, numeric libraries and GUI libraries for long time. These libraries suffer from one big disadvantage because they have fixed interfaces and data structures they use. There is very tight coupling between their algorithm and data, therefore they are not extend-able for user-defined datatypes.

Most of the users, when encounter the word *Reusable Software Components*, they immediately draw an analogy with automobile industry and start thinking or looking for **Black Box Software Components** . The word reuse in software has intrinsic character of being tailored according to one's need. Reuse doesn't prohibitive customization, but emphasizes on increasing productivity by learning from experiences of others and apply them judiciously.

The reusable software component should increase the quality of the software. There are no universally accepted metrics which can be used to measure the quality in an objective ways. Fenton (1991) described the software quality as

quality = Reliability + Availability + Maintainability + Usability
Maintainability = Understandability + Modifiability + Extendability + Testability

3.2 Design Patterns

The Oxford English Dictionary defines Pattern as
Anything fashioned, shaped, or designed to serve as a model from which something is to be made; a model, design, plan, or outline
Chrisopher Alexendra says "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing is the same way twice"
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides are considered to be strong proponents of design patters. In their famous book *Design Patterns: Elements of Reusable Object-Oriented Software*, they were first to catalog design patterns systematically. Their design pattern has been nicknames GOF (Gang of Four). GOF notes that most of the designers do not solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. Depending on the functionality of the patterns, GOF categorize patterns into three distinct classes.

3.3 Creational Patterns

Examples Singleton, Factory Method, Prototype Method

3.4 Structural Pattern

Examples Adapter, Bridge, Decorator, Flyweight

3.5 Behavioral Pattern

Chain of Responsibility, Memento, Observer, State, Template Method, Visitor

One of the reason why software developers do not exchange their software design because there is no common vocabulary to describe their software design. Design patterns helps developers understanding each other ideas and put across ideas at much higher-level abstractions.

3.6 Generic Programming

Generic programming recognizes the fact that dramatic productivity improvements could be achieved if algorithms and data on which they operate could be isolated and some abstract mechanism (iterators) should allow to traverse over the data.

Formally David Musser defined GP concisely

A Generic programming is *programming with concepts* where a concept is defined as a family of abstractions that are all related by a common set of requirements. A large part of activity of generic programming, particularly in the design of generic software components, consists of concept development-identifying sets of requirements that are general enough to be met by a large family of abstractions.

Generic components could be implemented either by class or templates. Constructing Generic Components using templates has advantage that their instantiation is done at compiler time and therefore avoid run time overheads associated with the late-binding mechanisms in different programming languages.

STL is perhaps the finest example of Generic programming. Inspired by STL, Boost Graph Library (BGL), MTL (Matrix template library), CGAL (Computational Geometry Algorithm Library), ITL (Iterative Template library) have been developed and they have confirmed that generic programming is indeed an extremely powerful mechanism for software reuse. All these libraries are template based and optimized, Results have been shown that their performance is comparable to the specialized codes.

3.7 Reluctance for Reusing Software Components

Despite the enormous advantages of reusable software components in short and long terms, incorporating them into new systems or in restructuring the existing applications have not been up to the expectations. From the economic perspective, Graham(1991) reported that reuse strategy could save more than 20 % of the development cost. In my view, reluctance to use could be attributed to the following reasons

- There are no automatic or semi-automatic tools to identify the structure or patterns in the software.
- Incremental approach is difficult therefore most organization avoid taking risk.
- There are no evidences from real applications which has demonstrated order of improvement in quality of the new system using the new techniques.
- The learning graph could be steep.
- There is temptation to rewrite codes among highly motivated software developers.
- There are little or no documentations or their support is limited.
- If the components comes from commercial companies, their are copyrights and royalty payments issues.

3.8 Mixed language Programming

In a large and evolving systems, it is not very unusual to find pieces of codes written in different languages. Most of the legacy codes are written in Fortran, C, Cobol or Pascal. Mixed programming languages has far more advantages than disadvantages, therefore, in almost good probability likely to stay there for long time to come in any large system.

- Some programming languages are better suited to some specific tasks.
- People have different expertise and sometimes biasness.
- Some higher level languages may be less efficient than low level languages.
- It is too expensive to rewrite the new system.
- The user base for the existing system may be large and better supported.

Compiled language (Fortran, C/C++) are order of magnitude faster than interpretive languages (Perl, Python etc), they are extensively used for fast prototyping in the initial phase of the software development, or when the execution time is not the important consideration. It is not unusual to find that a large system is entirely written in some interpreted languages and only small pieces of code, which require some performance improvement, are coded in higher performance compiled languages. Any software component development, should accept this reality and prepare himself for gluing his software with other components seamlessly.

3.8.1 Interfacing two different languages

Almost all programming languages provide some form of mechanisms to integrate with other programming languages. Many software make use of Language Interface Description Languages(IDL) to allow two different programming languages to talk to each other.
SWIG is an interface compiler (http://www.swig.org) that connects programs written in C and C++ with other scripting languages. It works by taking the declarations found in C/C++ header files and using them to generate the wrapper codes that scripting languages need to access the underlying C/C++ code.
SWIG is also a powerful too for rapid prototyping and system integration.

4 Research Plan

Our research plan is as follows

- We will study the design patterns and generic programming as applied in other application domains.
- We have identified one application and possible design patterns which could be applied in it.
- We will implement design patterns in C++ and integrate them into our application.
- We will study the flexibiity and quality of the new system and report our findings.

5 Experimental Testbed : Computational Geometry

5.1 Introduction

Computational Geometry is a sub-branch of algorithm design mainly concerns with design and analysis of algorithms for geometric problems involving objects such as points, segments, polygons, polyhedrons etc. Computational Geometry (CG) is an important component in various industrial applications such as mesh generation, computer vision, computer graphics, GIS etc. In most of the cases, robustness, rather than efficiency is of primer concern in adopting to a CG component. CG pose various difficulties in all phases of software development i.e. algorithm design, zero-tolerant numerical precision implementation, testing and debugging. The complexity in CG arises from numerical imprecision and whole set of datatype such as integer, real, rational number, exact-arithmetic, or adaptive floating point calculations come into picture. CG is one of the excellent example, where Generic Programming has been found to be extremely useful.

5.2 Brief description of application

Most of the scientific simulation need to discretize the geometric domain into small-2 geometric cells such as triangle (2D), and tetrahedra in 3D. The quality of cells has profound impact on the quality of the simulation. The process of generating high quality mesh is called Mesh Generation.

5.3 Components in Mesh Generation System

- **Geometric Modeling** Construction of Geometric Model involves design the model with predefined geometric primitives such as circle, plane, sphere, box or more sophisticated NURBS modeling. Very interactive and graphical display systems are needed to design these models. Most of the time, these model are constructed using commercial CAD systems.
  Component Language Most Fortran and C.

- **Adaptive or Multi-precision library** Geometric algorithm demands robustness and the use IEEE floating point available on most of the computers may not be desirable. Most researchers either use exact arithmetic of fast adaptive precision libraries for the calculations.

- **Geometric Kernel Library** Geometric Library has large collection of algorithms and data structures for geometric calculations. Most of the data structures are Spatial data structure such as KD-Trees, Quadtree, Octree, BSP etc. Algorithm involves finding line-line intersection, convex-hull calculations, polygon tessellations etc.
  Component Language C++ using templates.

- **Mesh Generation Algorithms** Mesh Generation is a process of discretization the domain into small-2 simplexes (called mesh) of 2 or 3 dimensions. Two most popular algorithms are Delaunay Triangulation and Advance Front Technique which are capable of generating high quality mesh in the domain.
  Component Language C or C++

- **Data Containers** In general, practical applications requires millions of simplices of 2-3 dimensions which are input for Finite Element analysis or for other applications. Storing and retrieving the data requires using databases such as SQL, Oracle etc.
  Component Language C/C++ interface with databases.

- **Parallel Decomposition and Object Migration Components** Mesh Generation is a time resource intensive application. It is not unusual to get a good mesh generation in more than 10-12 hours of dedicated computational time with huge requirements 20-30 GB or disk space. In recent times, distributed memory machines are being used to speed up the computations which require lots of sophisticated software infrastructure to decompose, distribute, and control and pieces of job assigned to each participating processes.
  Component Language C/C++, MPI, Threads.

- Interactive Visualization Either at the development time or at the last stage analysis, Mesh Generation require highly interactive graphical systems so much that they are integral part of software development.

5.4 Reuse in Computational Geometry

Reusing Software components in CG is not a question of personal choice, it is a must and require enforcement. In general, CG

5.4.1 Killer Primitives

- **Calculating area and angles of a triangle** Given the side-lengths a, b, c of a triangle, classical trigonometric gives

$$\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = (a+b+c)/2 \tag{1}$$

$$C = cos^{-1}\left(\frac{a^2+b^2-c^2}{2ab}\right) \tag{2}$$

  Well, these equations are so simple, that looking at the first glance, I would not have minded implementing them myself in 4-5 lines of code, but before you implement, please consult the 22 page paper by Prof. W. Kahan *Miscalculating Area and Angles of a Needle-like Triangle*. After reading the paper, you might do what I did, search for software on Internet.

- **Orientation Test** Orientation test are heart of most of Geometric computations. In simple words, these tests asks programs to determine the side (left, right or on) of one geometric object (point, line, plane etc) with respect to other other geometric object (line, circle, sphere etc). Does C lie on, to the left of, or to the right of vector ab.

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}$$

  The answer depends on the sign determination of the either of the two matrices.

  Does d lies on, inside, or outside of abs

$$\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix} = \begin{vmatrix} a_x - d_x & a_y - d_y & (a_x - d_x^2) + (a_y - d_y^2) \\ b_x - d_x & b_y - d_y & (b_x - d_x^2) + (b_y - d_y^2) \\ c_x - d_x & c_y - d_y & (c_x - d_x^2) + (c_y - d_y^2) \end{vmatrix}$$

  Again the answer depends on the sign determination of the either of the two matrices. To a beginner, evaluating a $2x2$, $3x3$ or $5x5$ matrix should be a trivial task. To a serious programmer or scientist, the quest for correct computer evaluation has opened the Pandora box for new research areas. Reader is encouraged to read Ph.D. thesis of Jonathan Shewchuk (http://www.cs.berkeley.edu/ jrs).

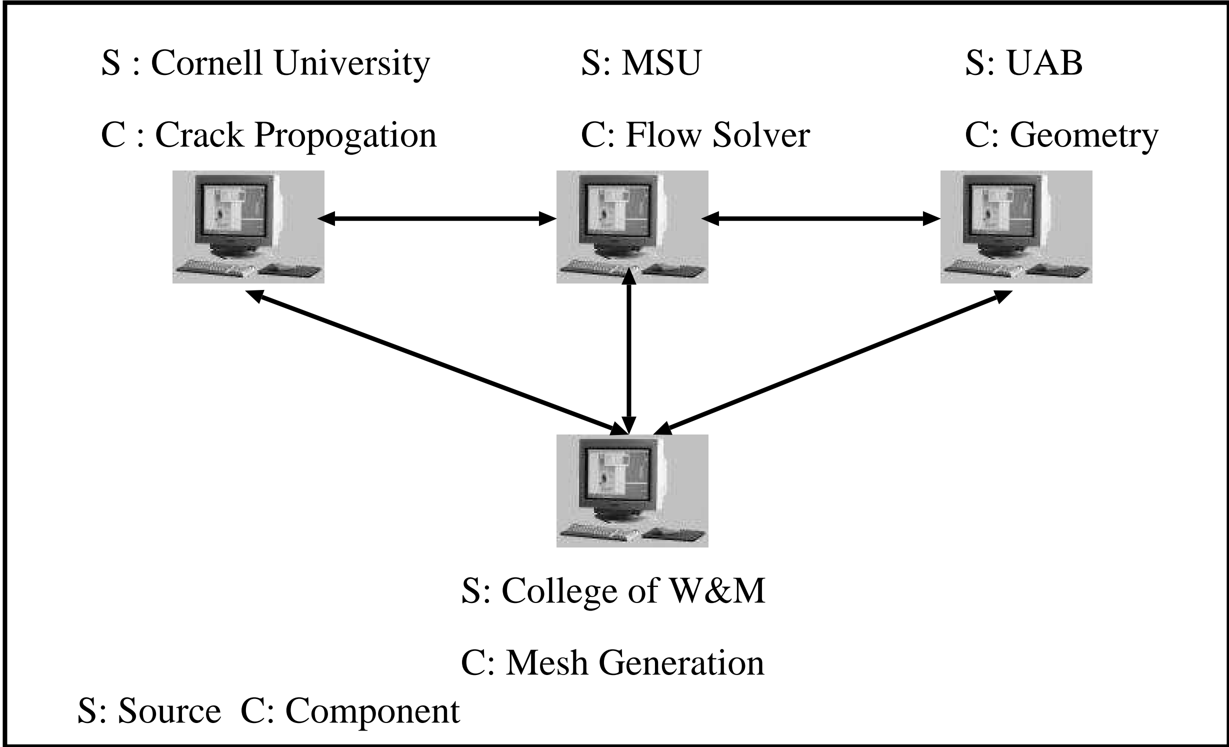
  If the user is still more convinced in writing his own trivial code, he should read more about them in Jonathan Shewchuk's lectures notes *Lecture Notes on Geometric Robustness*

5.4.2 Optimized Codes

Most of the software are generally not optimized to take hardware capabilities and often general purpose pieces of codes do not give more than 10-15 percent of the peak performance specified by the computer vendors. Modern processor design is very complex and employ sophisticated techniques such as Cache, prefatcing, pipelining etc to extract maximum performance out of CPU. Most of the general purpose software are oblivious to these parameters and therefore perform poorly. In many of the applications, overall performance is dictated by small number of functions and in performance improvement in there execution time dramatically improve the performance of the applications.
Geometric calculation is an application area which depends on few geometric functions and therefore it is important that these functions are highly optimized. Writing them is non-trivial task and this is one of the area, where reusing can be of great help.

6 Summary

We believe that reusing software is very important for software development. We have to accept the fact that software takes years to mature and therefore it is important that we do not discard the useful software which were written in traditional programming languages such as FORTRAN, C or interpreted languages such as Python, Perl etc. We need some middle-ware technique where quality of old software could be incorporate into the modern programming languages, and in our view Design patterns and generic programming could provide us some respectable solution.


S : Cornell University
C : Crack Propogation
S: MSU
C: Flow Solver
S: UAB
C: Geometry
S: College of W&M
C: Mesh Generation
S: Source C: Component

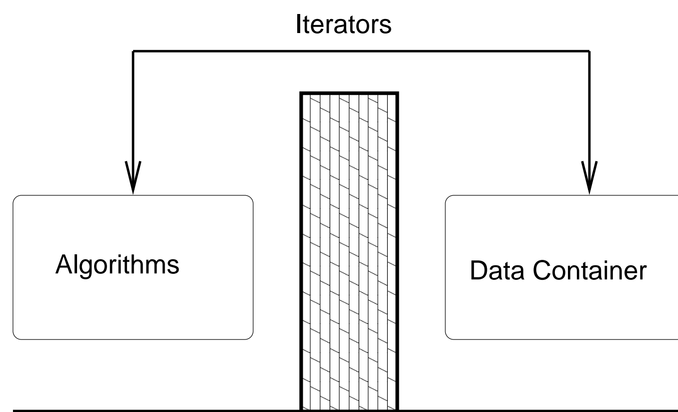


Figure 2: Successful Reuse require separation of concerns

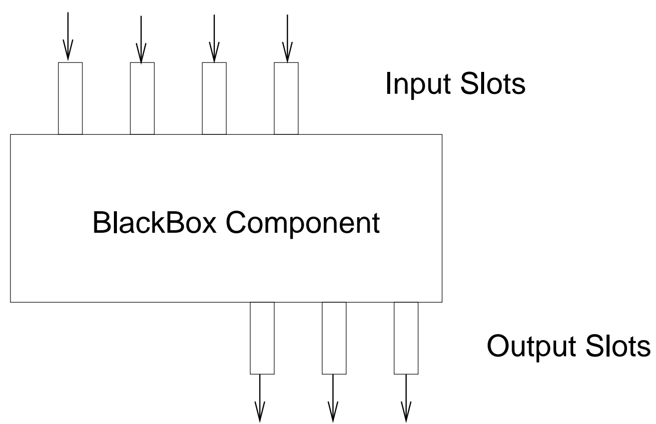


Figure 3: Common perception of Software Component