Polymorphic System Architecture

Jeff Bryson
Lockheed Martin Simulation, Training, & Support Orlando FL.
jeffery.e.bryson@lmco.com
jeffery.bryson@incose.org

Abstract. The complexity of tomorrows problems will be far greater then problems we face today. This complexity will only increase the importance of systems engineering and systems architecture activities. As the value of the activities rise, the cost of the activities must fall. Neither a Multi-year analysis and development schedule nor a short term schedule with a high risk of failure will be acceptable solutions. The solution for solving these complex problems will be to move systems engineering and architecture artifacts from the Non-Recurring Engineering cost over head to reusable company assets. A Polymorphic System Architecture (PSA) is one way of achieving this goal.

Run-Time polymorphism (RTP) has been used in the software community for over 20 years to satisfy dynamic reconfiguration, plug-n-play, extensibility, and system redundancy requirements. RTP is also used to construct software Systems of Systems. Systems engineers now have the same requirements applied to system architecture. A PSA utilizes the same software technology but applies it to the system architecture (Bryson 2009). The goals of using a PSA are:

- Reduce the complexity of the system architecture
- Satisfy functional requirements within the system architecture
- Define an architecture that has the potential to mature over time
- Move the system architecture from cost overhead artifacts to company assets
    - Create an architecture library of reusable artifacts that can be reused to solve new problems

Introduction

Polymorphic technology is a key component of advanced Object-Oriented software. The purpose of this paper is to provide guidance on applying this technology to both software and non-software systems. A PSA does not require software to be implemented. All that is really required are formally defined abstract interfaces, objects (or components) that are developed based on the abstract interface, and a good understanding of how to apply this technology to produce a reusable and adaptable architecture.

Polymorphic from ancient Greek means ‘many forms’ (AtomicObject 2010). The phrase **Polymorphism** is used in both Computer and Biological science. In the context of a PSA polymorphism means ‘many types’. Within the PSA the system components (software, hardware or other) can appear to be a different ‘**Type**’ of component at any point in time. It is this ability of components to change types that allows the architect to create a system that can dynamically change behavior and satisfy specific types of requirements.

Within a system design, functional requirements are allocated to decomposed elements of the design. Within a PSA, specific types of requirements allocated directly to the architecture. The design will often dictate an implementation, where a PSA will at most only imply an implementation. This definition of architecture extends the definition provided within the INCOSE SE Handbook v3.2 (INCOSE 2010). The ability to dynamically change components

also allows systems to plug-n-play new services. The key activity of the architecture team is to identify and define these 'Types' and link them together with the relationships and patterns presented in this paper. The results of this activity should allow the components created or instantiated from the architecture to be assembled (dynamically) in multiple ways to solve multiple, dynamic, and complex problems. In UML and SysML these 'Types' are defined as '**Classes**', 'Objects' are instantiated from these 'Classes'. This ability of the system to reassemble itself and dynamically change behavior during execution creates 'Run-Time Polymorphism'.

The relationships between the design attributes (or architectural component types) are the primary artifacts of the architecture (Bryson 2009). The relationships of '**Shared Aggregation**', '**Inheritance**', and '**Abstraction**' are specifically used to create a polymorphic architecture. The system architect must understand these technologies, not only to create the system architecture, but to explain how the system architect can directly satisfy specific functional requirements. "When specific types of relationships between classes are identified, the 'whole' of the 'design' (or architecture) becomes much more powerful than the sum of the parts" (Bryson, 2008). Reference Appendix A for a detailed definition of these and other concepts the System Architect must understand.

Abstract interfaces are used to create RTP in computer science. This same technology is now being applied to software and non-software components within the system of Service Oriented Architectures (SOA) and System of System solutions. The application of this technology at the system level enables the creation of a 'less' complex system architecture that can solve one or more complex problems.

Shared Aggregation allows the system to dynamically find and use (or reuse) a service instance (or component) when required. Once the service instance is no longer needed it should be released for use by other parts of the system. Composition defines one component as permanently dependent on another component. **Abstraction** means that the system is going to reuse the 'control' behavior within a system. **Inheritance** allows one defined type/class/architecture to reuse behavior, functional requirements, interface requirements, analysis, and testing associated with a previously defined type/class/architecture. This reuse of systems engineering artifacts requires a formal linkage between the architecture elements and the requirements, analysis, and testing (Bryson, 2009). These polymorphic concepts are illustrated in the design pattern known as the 'Strategy' pattern (Gamma 1995). The Strategy pattern is a common pattern for implementing SOA systems and is illustrated in the UML diagram in Figure 1.

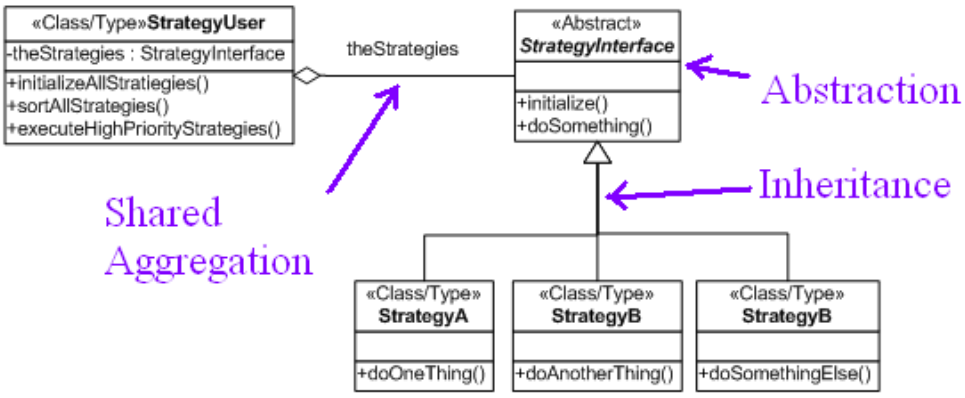

Figure 1 Gang of Four (GoF) Strategy Design Pattern

Figure 2 is a UML diagram that shows both the classes/types and objects/components of a

specific implementation of the strategy design pattern. By presenting both the Class/Object or Type/Component relationships in the diagram the value of this pattern can be seen. Not only are the Object/Components in the diagram reusable but the architecture (classes/types + relationships) is reusable. The 'controller' does not care that 'Component2' is instantiated from the 'StrategyA' class or that 'Component6' is instantiated from the 'StrategyC' class. The 'controller' can invoke the 'initialize' or 'doSomething' operation on any strategy implementation. The shared aggregation, abstraction, and inheritance relationship in the pattern creates common control of components behavior that may be implemented in different ways.

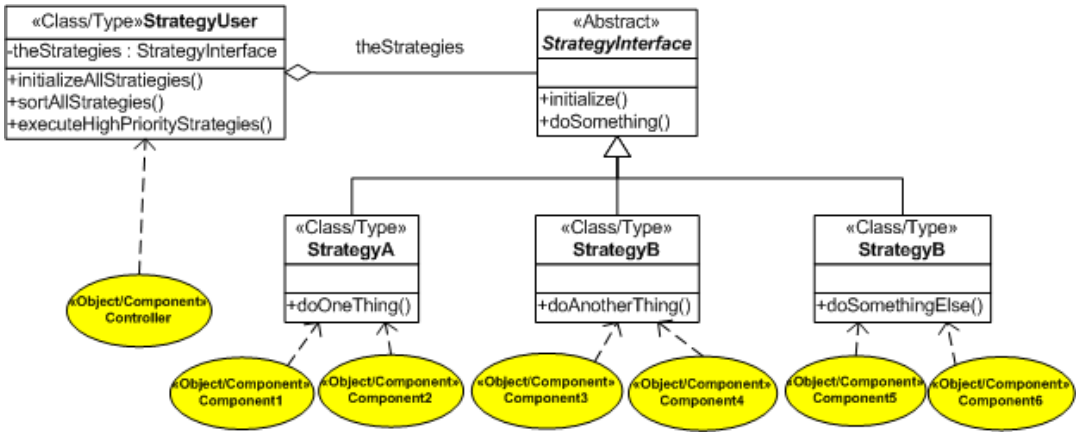

Figure 2 GoF Strategy Design Pattern Implementation

The architect, designer, and developers must understand that the "Abstract" interface does not exist to make the job easier for the developers of abstract components (Strategies A, B, and C's). The development of the strategy classes (StrategyA, StrategyB, and StrategyC) becomes more constrained. The 'Abstraction' exists to allow the "StrageyUser" (or the strategy controller) the ability to dynamically reconfigure system behavior and allow for the dynamic plug and play of new implementations of the abstract interface. The components created based on the strategy classes ("StrategyA", "StrategyB", and "StrategyC") become dynamic building blocks of the system that become reusable assets. This ability of the system to dynamically reconfigure how the components of the system work together allows the architecture to change (and be reused) to solve different problems. The strategy pattern also provides extensibility by allowing new instances of the existing behavior classes to be dynamically created. This 'maturing' ability is described in more detail later.

The strategy pattern does have a drawback. It defines a single point of control. This can create a single point of failure in the system. An even more powerful polymorphic architecture that removes this drawback is based on the 'Composite' design pattern (Gamma 1995) illustrated in Figure 3.

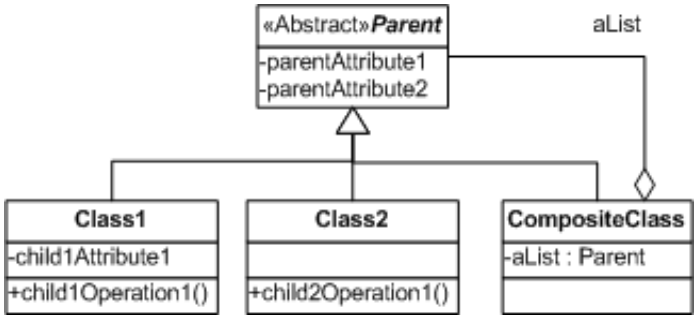

Figure 3 GoF Composite Design Pattern

The Strategy pattern can be extended to become the Composite pattern by defining the "StrategyUser" (Controller) as one of the strategy behaviors (see the 'CompositeClass' above). This is an excellent pattern for developing a 'System of System' This dual relationship (shared aggregation and inheritance in the controller) allows the architect to build a system of abstract controllers that can control other controllers. The controllers become building blocks of the system just like the other components of the system. The components of these patterns or architectures are not required to be software components. There is a requirement for a formal definition of the abstract interface. The connections between the components of this type of system are reconfigurable and provide the ability to dynamically alter system behavior.

The ability to dynamically reconfigure these building blocks allows the PSA to satisfy functional requirements. The encapsulation of the complete set of elements associated with the architecture provides reusability of the architecture. (Bryson 2008)

Polymorphic Functional Requirements

The PSA allows system developers to satisfy the following types of requirements at a higher level of abstraction.

- Dynamic reconfiguration of system functionality
- Plug-N-Play functionality
- Extendibility
- System Redundancy
- System Reuse

A PSA allows the architect to define a system of components that can be reconfigured to solve new types of problems. These components are building blocks that can be used and reused. If one component becomes unavailable another one can be dynamically connected to replace it. New types of components can be added to the system easily as long as they adhere to the defined abstract interfaces. A PSA can reduce the complexity, development and maintenance effort of the system and provides a framework for the components to be used and reused as building blocks of the system.

Polymorphic Mediators

A polymorphic architecture requires a '**Polymorphic Mediator**' within the system. The mediator allows the controller components to connect dynamically with the service components regardless of location.

A standardized middleware application is generally used in software. In the Common Object Request Broker Architecture (CORBA) an Object Request Broker (ORB) application executes on each processor. The ORB is the polymorphic mediator which allows a polymorphic server to register its service. The ORB also allows a polymorphic client to request and connect to a specific server. In Web Services, the Universal Description Discovery and Integration (UDDI) service provides the same functionality.

In system architecture, the software components should use a standardized middleware product. However, non-software portions of the system require different types of polymorphic mediators. This mediator could be as simple as a call center that links the client with an available server or as complicated as an airborne, space borne, or sea based communications network. A well defined and managed abstract interface is required to provide a reliable linkage between clients and servers in the system architecture. The abstractions within the architecture should be independent of any specific middleware implementation.

A Maturing Architecture

Even if there is little or no software associated to the architecture, a PSA is an Object-Oriented product. "OO" technology and (good) refactoring allows the product developer to create a product (or product line) that can mature over time. A PSA by itself does not guarantee a system that matures. Refactoring (reinvestment) activities must occur over the entire life cycle of the product line. This activity implies that the architecture and related engineering artifacts are evolving and maturing over the system life cycle. The architecture(s) become an evolving and maturing asset for multiple products.

How does this maturing behavior occur? Some examples show us this process. Figure 4 and Figure 5 shows how the Strategy pattern is matured to create the composite pattern.

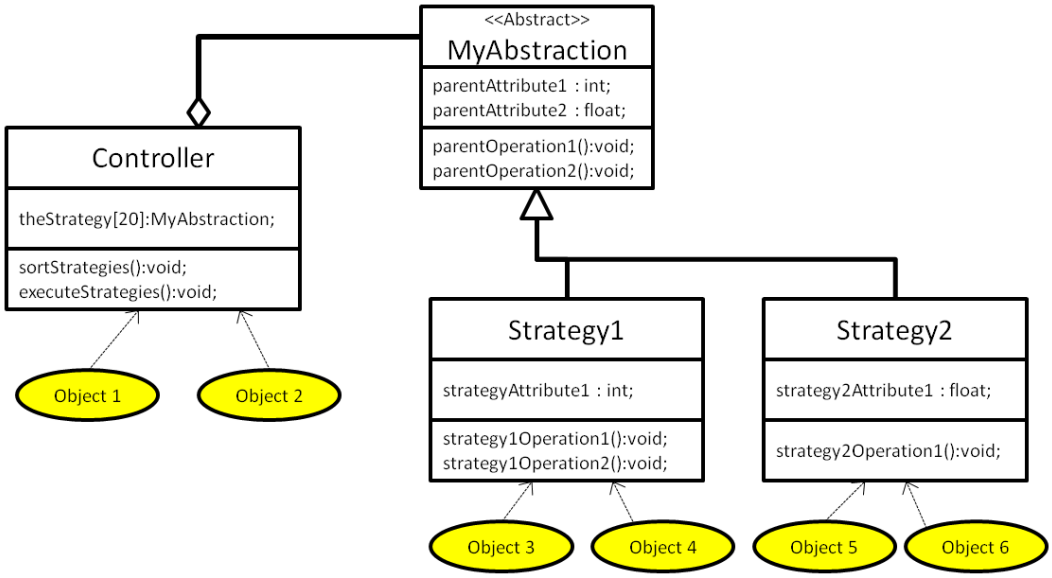

Figure 4 Strategy Design Patterns

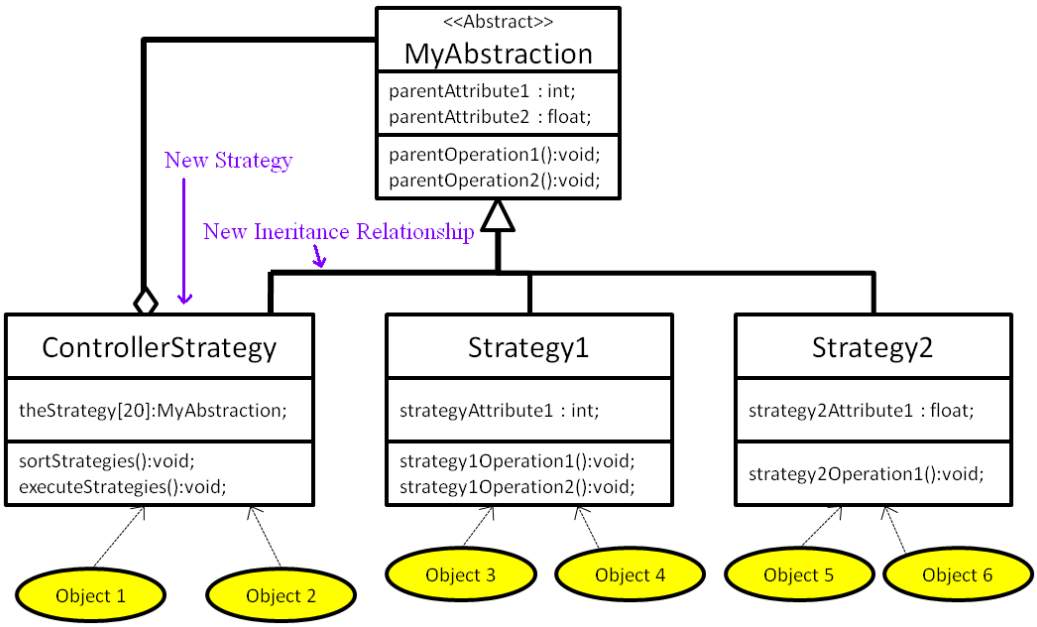

Figure 5 Maturing Strategy pattern to Composite pattern

The ‘Strategy’ pattern matures into the ‘Composite’ pattern by adding an inheritance relationship between the ‘ControllerStrategy’ and the abstract interface. All controllers in the new architecture become reconfigurable building blocks. Because controllers can manage any kind of strategy, and controllers are a strategy, controllers can now mange controllers. With the polymorphic relationships the Composite design pattern gains the ability to create architectures of architectures or systems of systems. If existing ‘Strategy’ based architecture systems were updated with this new architecture, only objects 1 & 2 above would need to be rebuilt. The change should only occur because there is a need (requirement) for controllers to control controller. In this case, any changes to the system will only be associated to instances of the controller class.

Figure 6 shows us how to mature the system with extensions or by adding new instances of objects.

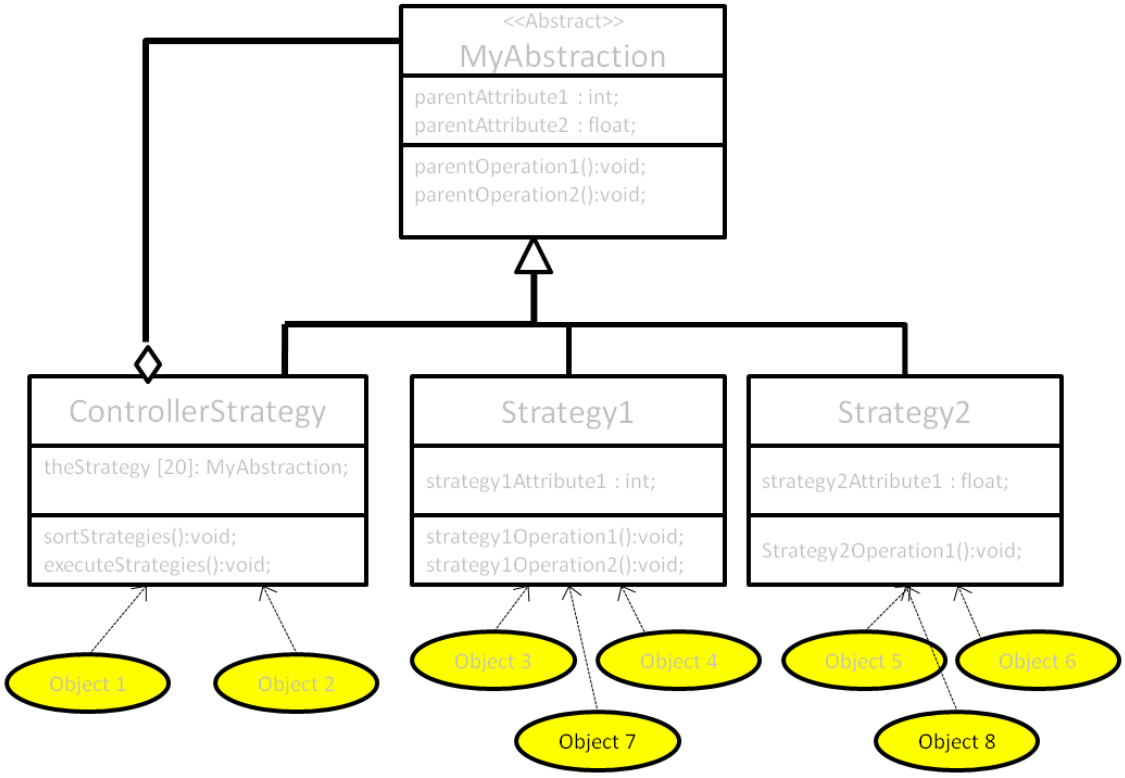

Figure 6 Extensions with New Objects

In this example Objects 7 and 8 are new implementations of Strategies 1 & 2. There is no need to update any part of the existing system as long as there are adequate resources within the system to manage the addition of the new objects.

A third method of maturing the architecture is to add a new strategy. Figure 7 illustrates how this is done.

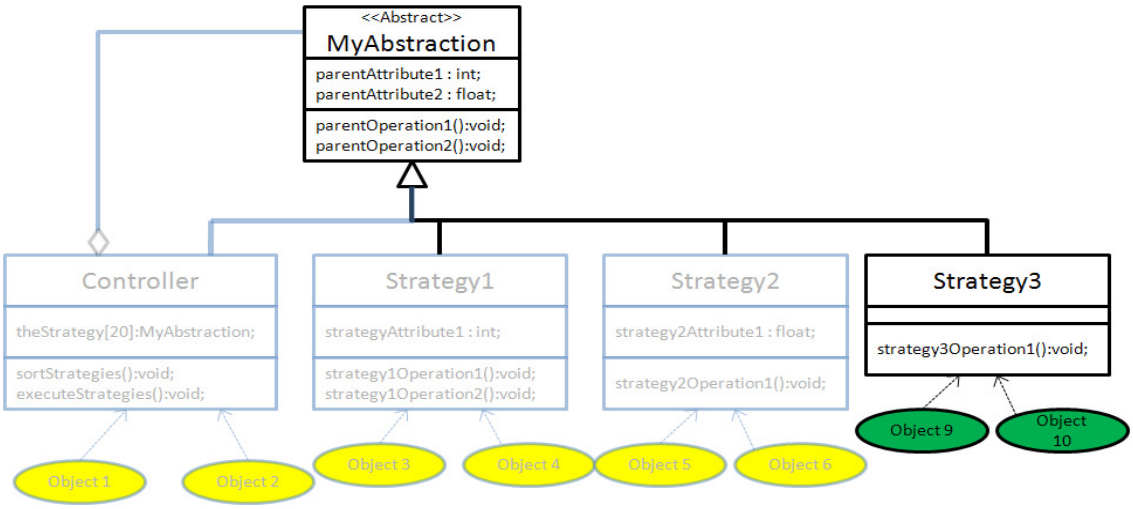

Figure 7 Extensions with new Strategy

The final example is more complicated and illustrates refactoring of the system architecture artifacts. The effects of change are more costly, but the architecture is truly maturing and improving over time. If operations 'strategy1operationXYZ()' and 'stratgy2operationXYZ()' are identified as common to only strategies 1 & 2 (see Figure 8), then the architecture should mature and combine the operations as a single common service.

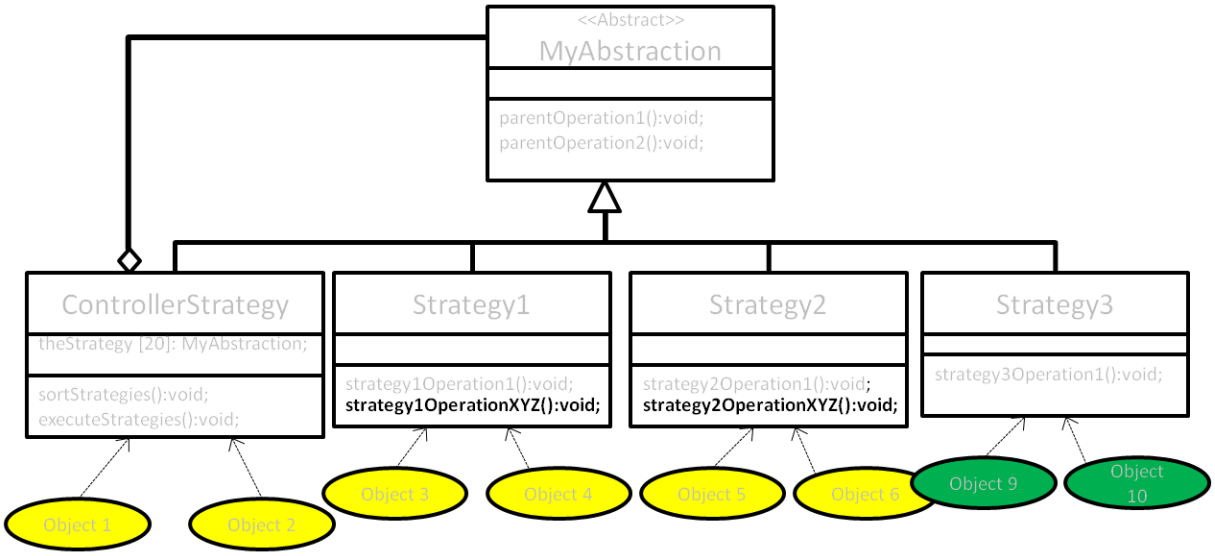

Figure 8 Discovery of Common Behavior

In order to reuse the behavior of the operation and create a single definition to document, maintain and test, an additional layer of abstraction between Strategies 1 and 2 and the original abstract interface is created. This behavior is illustrated in Figure 9.

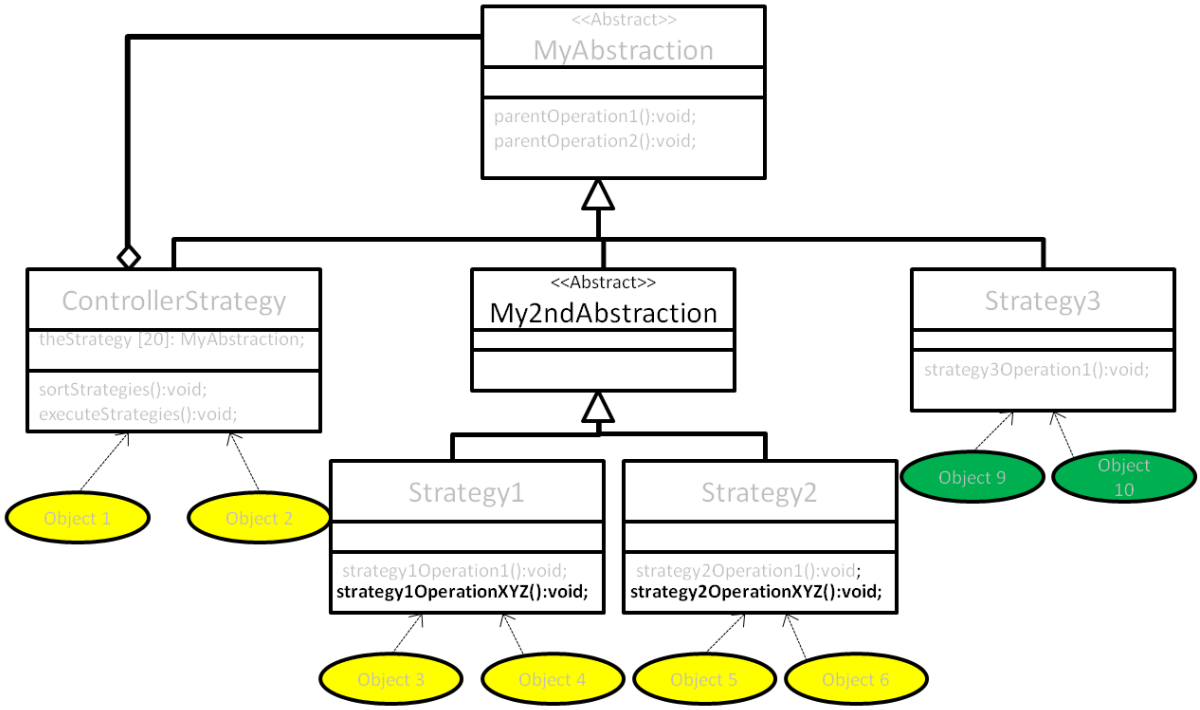

Figure 9 New Abstraction Layer

The common behavior is promoted to the new layer of inheritance. The ability to promote and demote behaviors within the architecture is required to allow the architecture to mature. This ability is illustrated in Figure 10.

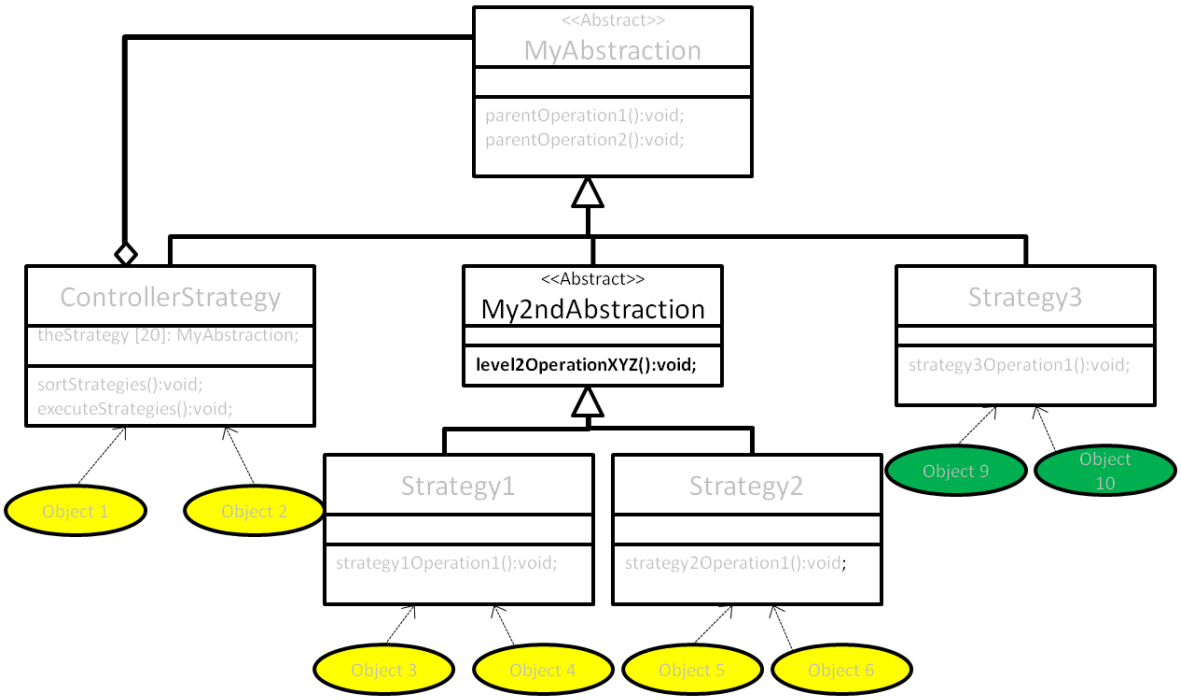

Figure 10 Promoted Behavior

This type of change does not come with zero cost. While the functionality of Objects 3-6 do not change, all instances of the Strategy1 and Strategy 2 classes will need to be rebuilt. There may

be a need for an updated the logic within the controller with each level of abstraction added to the architecture. These types of changes should not be made lightly as there will be impacts to portions of the system as each change occurs. Many existing products become obsolete over time while these types of updates within a PSA improve the architecture and create reusable artifacts that mature over time and last indefinitely.

A Polymorphic System Architecture encapsulates all of the systems engineering artifacts and can mature over time. This ability implies that the System Architecture (or Architecture library) is a living entity. *Both encapsulation and refactoring allow system engineering and system architecture activities to move from Non-Recurring Engineering (NRE) activity and cost to the corporate reusable asset list.*

Applying Polymorphic System Architecture

It is not difficult for the non software Object-Oriented experienced engineer to understand how and where to apply this style of engineering. If the logic in the system under development requires nested "IF" conditions or "Case" logic (regardless if the logic is software driven or driven by other means) it is easy to translate this functional logic to a (run-time) polymorphic logic. Figure 11 illustrates how nested "IF" conditions can be translated to 'Case" logic and then how the case logic can be translated to polymorphic logic. The specific rules are:

- The 'value' translates to the abstract interface reference
- Each enumerated value (red, blue, and green) becomes a child class derived from the abstract interface.
- There is a single (common) behavior/operation added to the abstract interface

The decision on which behavior to invoke is determined dynamically by which object the abstract reference addresses.

```
If (value = red)
  doSomething();
elseif(value=blue)
  doSomethingElse()
elseif(value=green)
  doAnotherThing();
endif;
```

```
Case value is
 when red =>
    doSomething();
when blue =>
    doSomethingElse();
when green =>
    doAnotherthing();
end case;
```

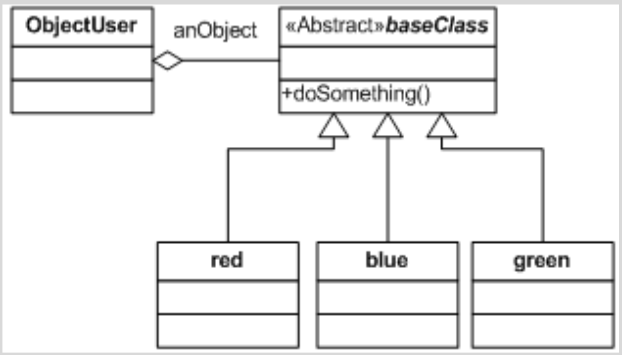

Figure 11 PSA Translation

There are two key activities for the system architect to focus on. First, identify system requirements that match the requirements identified within this paper. These types of requirements should be allocated to the. Second, the complexity of the solution should be reduced by identifying common service and control behavior within the system and using inheritance and abstraction to create single entities within the architecture to provide solutions for multiple problems. Common Service behavior is defined using 'inheritance'. Common control behavior is defined using 'abstraction' and 'shared aggregation'.

Although Model Based Systems Engineering (MBSE) is not required to create a PSA, proper semantics within modeling tools provide the needed assistance in encapsulating and linking the architecture artifacts. The potential also exists with a MBSE environment to create an

executable (logical) architecture that can be used to verify the architecture prior to component development. Executable architectures can be used to "perform dynamic analysis on the architecture" (Friendenthal 2007) to reduce risk during the maturing activities of a PSA.

Functional Design Solutions

All the above capabilities can be implemented using standard structure design methods. However, linear growth of these functional designs is very expensive and can quickly create an uncontrollable environment. Satisfying these types of requirements in a functional architecture has the ability to easily migrate to the 'Highly Coupled' anti-pattern. Patterns capture expert knowledge about "best practices", antipatterns because their use (or misuse) produces negative consequences (Smith 2000). This anti-pattern can create a 'Stove Pipe' architecture. A Stove Pipe is "A system procured and developed to solve a specific problem characterized by a limited focus and functionality; a system that contains data that cannot be easily shared with other systems"(DOE 1999). A stovepipe system has problems with extensibility because each component is statically bound directly to other components require to complete a task.

Defining interfaces should be part of the requirements analysis and system architecture effort. Each interface requirement should be linked to the functional behavior (requirement) that triggers the interface and the functional behavior (requirement) that is triggered by the interface. (Bryson 2009)

Where can the System Architect use PSA's?

A PSA satisfies the requirements of a Service Oriented Architecture (SOA) but not all SOA's will satisfy the definition of a PSA. A PSA requires the use of abstraction and inheritance where a SOA does not. The services or Service Level Agreements (SLA) within the SOA are the abstract interfaces within the PSA environment. Service providers publish their servers which are dynamically accessed by service consumers as needed. New services can be added to the system to allow the system to be extended. "Services provide the incremental building blocks (within an SOA) around which business flexibility revolves, but services need a supporting architecture for their deployment, delivery, and management".(IBM 2007)

In a System of System (SoS) design, a PSA allows a polymorphic client to access a polymorphic server (regardless of the servers location) and creates a system that exists in a virtual multi-core environment. Depending on the performance requirements, a polymorphic SoS could execute on a single processor or 100,000 processors. As long as subsystems are designed to execute in parallel, the SoS can expand and shrink as performance requirements change. Using architectures that are no longer dependent on where logic behavior is executing the migration to new technologies (multi-core processors [100+], shared cache) becomes much less costly and the benefits of a PSA can be fully realized.

The engineering process should have one or more levels of abstraction defined for each activity (Management, Analysis, Architecture/Design, Development, Implementation, Test, and Maintenance). There may be different types of these for H/W, S/W, Product Support, and Manufacturing that can be reused depending on the requirements of a specific contract.

The Military command language is a process that conforms to the PSA idea. The command to Advance, Engage, and Withdraw have the same basic meaning for a fighter pilot, ground soldier, Navy Task force, and Marine Platoon. Each one of these entities could carry out each command differently.

Our Education system could benefit from a PSA. The goal of every class should be to teach each student how to solve problems. The basic process for solving any kind of problem is the

same.

- Define what the problem is
- Define what you know
- Define what you need but don't know
- Define a solution

The implementation of these activities to Mathematics, Science, Social Science, Grammar, Sports, and Art will all be different. How the process is applied to problems of varying complexity may also be different and may require students and teachers to change the specifics of how they solve problems. In a complex problem you may also define what is and what is not in scope. Variables that are within your control and which ones are not man also need to be defined.

Additional Work to be Completed

Measuring the 'Abstractions', 'Inheritance', and 'Shared Aggregation' relationships in the system architecture can identify if the architecture is polymorphic or not. Identifying the specific requirements that are directly allocated to the architecture can also provide a valuable metric. However these measurements do not identify the quality of the architecture. Work still needs to be done to identify the specific relationships and patterns (metrics) within the system architectures to provide guidance as to the quality of the system architecture.

A Case Study that spans the use of a common PSA across at least 3 systems in a single product line needs to be completed. Cost, Schedule, Quality, and customer satisfaction metrics need to be gathered to provide justification for this engineering style.

Conclusion

As projects become more complex, there is a growing need to reuse software, hardware, and system engineering artifacts. Polymorphism exists specifically to create reuse of control and service behavior. A Polymorphic System Architecture adds value by creating extendable and reusable systems engineering artifacts. The abilities of a Polymorphic System Architecture that matures over time and moves systems engineering artifacts from Non Recurring Engineering cost to corporate assets can make the system architecture invaluable. When the system architecture satisfies requirements, the complexity of the system can be reduced. Polymorphic technology applied to software and non-software systems provides a means to achieve these goals. This depends on the ability of the architects, customers, and program managers to identify and understand the specific types of 'Relationships' used to create a Polymorphic System Architecture. The end goal of applying polymorphic technology to systems architecture is to be able to start solving a new problem with an 80% proven solution.

Appendix A PSA Terms and Technology

To define these relationships the system architect needs to understand the following specific Object-Oriented terms and technology.

Containment	Object	Composition
Encapsulation	Polymorphism	Inheritance
Type	Object Reference	Abstraction
Class	Aggregation Composite & Shared	Polymorphic Servers
Polymorphic Clients	Polymorphic Mediator	

Containment – A primitive pattern in which one construct (structure) contains a collection of other constructs.

Encapsulation – A primitive pattern where data and behavior are grouped (or tied) together in a single construct. "*Encapsulation simply says that there should be a way to associate the two* (data and behavior) *closely together and treat them as a single unit of organization*". (Linden 1999) Encapsulation extends the pattern of containment and creates a new construct where specific Data and Behavior related to that Data are grouped together as a single entity. Figure 12 illustrates a UML class that binds together Data (attributes) and Behavior (operations). In Object-Oriented environments encapsulation provides one form of code reuse. The construct that provides this grouping is the class.

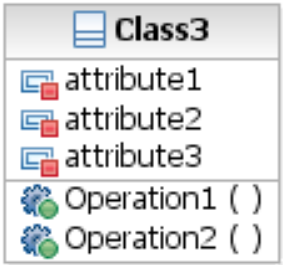

Figure 12 Encapsulation Primitive Pattern

Within the context of a PSA the concept of encapsulation must be extended to a higher level of abstraction. 'Architecture' needs to be the encapsulating entity when using the PSA style of engineering. The parts of the encapsulated architecture will include the 'Classes' and specific relationships between the classes. Also encapsulated within this architecture will be the requirements, requirements analysis, and testing associated with the requirements satisfied by the architecture.

Type –The Gang of Four defines a 'Type' as "identifying a specific interface to data". (Gamma 1995) In a PSA, 'Types' are used to define formal (externally visible) specification used to interact with components. The components can be as simple as an integer 'type' allocated in software or a complete system architecture. By 'typing' the object or component, the architect places specific constraints on how the object or component is used.

Class – "An object class describes a group of objects with similar properties, common behavior, common relationships to other objects, and common semantics" (Rumbaugh 1987). The class is the construct in Object-Oriented Programming that encapsulates behavior and data. When reviewing a polymorphic architecture, it is important, to understand that a class defines a 'Type' of construct. In a PSA the classes are the specifications for the building blocks of the system. An 'integer' is a type for representing numbers. A class is a user defined type for encapsulating data and behavior. The class is used to create objects or instances of the class. The objects reuse the behavior and data definition within the class but the value of the data is

unique for each object.

Using ‘Types’ and ‘Classes’ are one way of encapsulating data and behavior.

Object – An instance of a class type. Rumbaugh defines an object as “a concept/abstraction/thing with a well defined boundary and is relative to the problem at hand”. (Rumbaugh 1987) Object are dynamically created instances of the class type. The objects are the building blocks of a PSA. The classes are the specification for those building blocks. Figure 2 illustrates this relationship.

Object Reference – An object reference is a construct that allows an object user to hold onto the object. In C++ the object reference is literally a pointer to where the object is located in memory. In Figure 2 the attribute ‘theStrategies’ is an array of object references of the type ‘’StrategyInterface’.

It should be noted that a ‘GOTO’ statement (memory pointer), ‘Pointer’ to an operation, and a ‘Reference’ to an abstract interface are **NOT** the same things. Using memory pointers and operation pointers are no different than using the ‘GOTO’ statement. An interface reference to a component/object uses the ‘Type’ specification of the class to ensure that a request for service is associated to an appropriate server implementation. This relationship is verified in strongly typed programming languages by the compiler. This relationship can also be ensured in system architecture with non-software components by using a strongly ‘typed’ Model-Based System Engineering (MBSE) tool.

Aggregation Shared and Composite – Identify specific ways that objects or components are related to each other. Aggregation indicates that one construct has is part of to a second construct. Shared and Composite aggregations are specific types of aggregations (OMG 2009). Composite aggregation occurs when one construct is contained within another. When the first construct is created the second construct is also created. When the first construct is deleted then the second construct is also deleted. With the composition relationship the second construct will live and die with the first. Shared aggregation is when one construct references another. In shared aggregation the second construct can be referenced and shared by multiple constructs (OMG 2009). Shared aggregation is shown with the non-shaded diamond and composite aggregation is shown with the shaded diamond in UML/SysML. The diamond is connected to the class that uses the referenced class as illustrated in Figure 13.

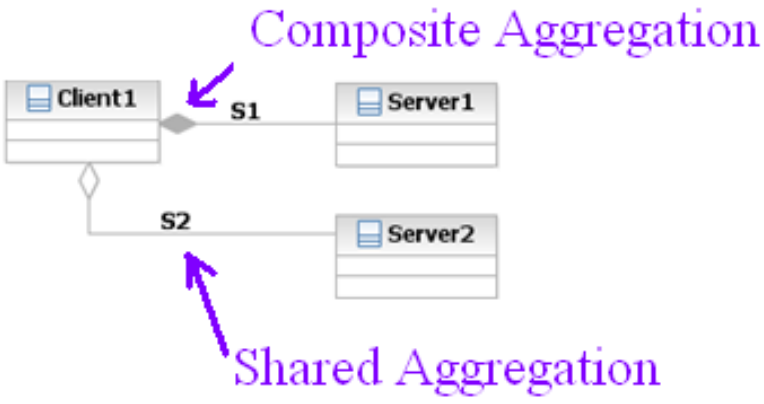

Figure 13 Aggregation and Composition Primitive Pattern UML Diagram

Inheritance – A relationship between two or more classes (also known as generalization in UML/SysML)(OMG 2009). “This relationship allows one class (*or type*) to refine (or extend) another class (*or type*)”. (Linden 1999) One class is the parent class (or superclass) and the

other class is the child class (or subclass). The arrow head points to the parent class in UML/SysML (Figure 14). The child class will contain the data definition and behavior defined in the parent class. Inheritance is not a copy of the data structure and behavior. If the parent class is updated, the child class will receive the same updates. The parent provides a single definition that is shared by all the children.

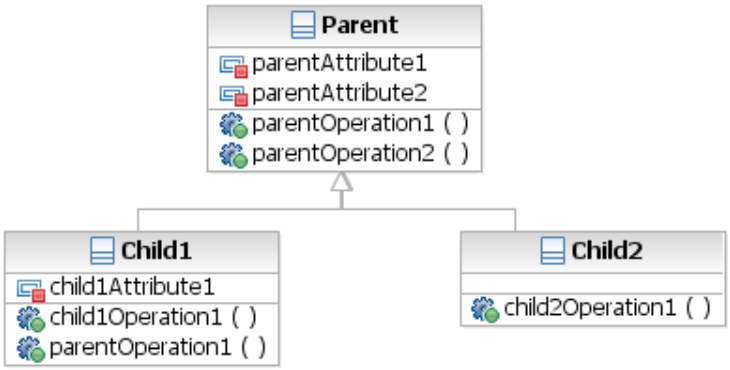

Figure 14 Inheritance Primitive Pattern UML Class Diagram

Polymorphism – The GoF (Gamma 1995) defined polymorphism as the ability to substitute objects with matching interfaces at run-time. Polymorphism allows the system to have multiple implementations of behavior and then select the appropriate behavior depending on the circumstance. Polymorphism allows an architect to define an interface to behavior that is independent to how the behavior is implemented. There are two basic types of polymorphism. Static polymorphism (or overloading) occurs when the selected behavior is defined at software compile time or system instantiation. Overloading is generally created by defining multiple operations with the same name. Each operation has the same name, but the input parameters are of different types. Once the system is instantiated the behavior logic will not change.

Run-Time Polymorphism (RTP) occurs dynamically. The behavior logic changes within the system as it executes. RTP requires the definition of an abstract class or interface. RTP allows architects to define systems that have functional requirements to dynamically change behavior. The strong typing of the abstract interface allows the reliable dynamic changing of behavior.

Abstraction – Dictionary.com's definition states "an abstract or general idea or term" (Dictionary.com 2010). This definition is **NOT** the way abstraction is being used in a PSA process. The GoF defines an abstract class as "a specialized class that defines an interface and defers the implementation to subclasses". (Gamma 1995) Abstraction in this context is defining specific constraints that users of the abstract interface must adhere to. It is the behavior behind the interface that is abstract.

A Software or System architect can create a PSA by defining an abstract interface, having a polymorphic client use (or reference) the interface, and having objects instantiated from subclasses of the abstract interface. RTP is created by combining the relationships of aggregation, and inheritance with abstraction.

In some software languages, abstraction is also referred to as an 'Interface' (i.e. JAVA). The use of the word 'Interface' differs from what is commonly used in 'Structured Design'. In structured design an interface commonly represents a 'Pipe' that connects two hardware or software components. This pipe is viewed as a cable or wire between two connected points. An abstraction defines the connectivity between two points differently. In a structured design if the interface between two components is seen as a wire, then in a polymorphic design, the connector on the component and the connector on the wire are combined to create the (abstract) interface. The polymorphic system has the ability to dynamically plug different wires to a

single connection as long at the connection (abstraction) on the wire matches the connection on the component. It is the ability to dynamically change these connections that allows a polymorphic architecture to alter system behavior and satisfy functional requirements. *The Abstract interface creates the weakest form of coupling possible between two components. This weak coupling allows for an architecture that is much less impacted by changes in the system.* (Unknown 2009)

This Abstraction is an 'Aggregate' relationship. The child construct inherits the interface from the abstract construct. There is never an instance (object) of the abstraction (class). Constructs that inherit from the abstraction are required to implement the defined behavior (operations) in the abstract class. This concept creates a strongly 'typed' and highly reliable 'GOTO' statement. A controlling object will have a reference to the Abstract Construct type (class). This reference can only point to instances of children of the abstract construct. Each child type is required to implement its own version of the behavior while still adhering to the defined abstraction (interface). When the controller changes the reference from one child construct instance to a different child construct instance the behavior will change dynamically (i.e. during Run-Time). Figure 15 illustrates how abstraction is represented in UML/SysML.

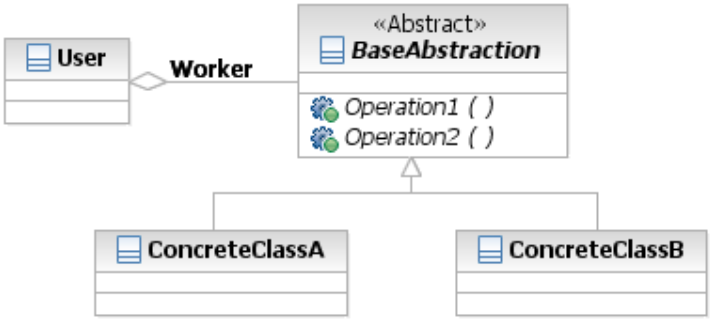

Figure 15 Abstraction Primitive Pattern UML Class Diagram

Architecture – The decomposition of a system where the synthesis of the decomposition satisfies specific requirements that have been allocated to the architecture. The architecture should also reduce the complexity of the solution. Architecture provides the definition and justification of the building blocks of the system. The definition of specific types of relationships allows the system architect to create a PSA and satisfy specific functional requirements with the architecture. In a polymorphic architecture the architect is focused on defining the types of components in the system and the relationships between these types.

Polymorphic Mediators - allows the polymorphic clients to find polymorphic servers that provide the services defined in a specified abstraction. There may be multiple mediators within a system. The mediators must work together to allow clients and servers to connect dynamically.

> **Polymorphic Servers** – Classes/Types that inherit from the defined abstract interface. Components/Objects will be instantiated from these classes.
>
> **Polymorphic Clients** – Classes/types that reference or have an aggregation to the abstract class. Each client will and should have server load limits. PSA load balancing can be accomplished by a client easily handing servers to another client. During critical system activities clients can also balance and share critical behavior of the system.

References

AtomicObject. *Object Oriented Polymorphism.* 2010. http://www.atomicobject.com/pages/Polymorphism.

Bryson. «Prescriptive Requirements Analysis.» INCOSE Region V Conference, 2009.

------. «Plymorphic System Architecture Summary.» OOPSLA 2009.

------. «The UML Design Abstraction.» OMG Technical Meeting, 2008.

Dictionary.com. *dictionary.com.* 2010. http://dictionary.reference.com/.

DOE. «Improving Project Management in the Department of Energy.» ISBN-10: 10-309-06626-3, 1999.

Friendenthal «Object-Oriented Systems Engineering Method (OOSEM) applied to Joint Force Projection (JFP), a Lockheed Martin Intergrating Concept (LMIC).» INCOSE 17th Annual International Symposium, 2007.

Gamma, Helm, Johnson, Vilssides. «Design Patterns Elecment of Reusable Object-Oriented Software.» Addison-Wesley, 1995.

IBM. «Introducing the Value and Governance Model of Service-Oriented Architecture.» IBM Online course VW003. 2007

INCOSE. *INCOSE Systems Engineering Handbook V3.2.* INCOSE-TP-2003-002-03.2, 2010.

Linden, Peter Van Der. *Just Java 2.* Sun Microsystems Press, 1999.

OMG. *OMG Unified Modeling Language Superstructure Version 2.2.* Specification, www.omg.org, 2009.

Rumbaugh, James. *Object-Oriented Modeling and Design.* Prentice-Hall, 1987.

Smith, Willians. Software Performance AntiPatterns 2nd Internation Workshop on Software and Performance. 2000

Unknown. 2009.

BIOGRAPHY

Jeff Bryson has his masters in Computer Science from Florida Institute of Technology. Mr. Bryson has worked in the commercial and defence software industry for over 25 years. He has provided solutions for Sprint Telecom, NASA, Perot Systems and has taught Computer Science at DeVry University. Mr. Bryson is a staff software engineer at Lockheed Martin Simulation, Training & Support in Orlando FL. He has received the OMG UML Certification and is an IBM certified SOA associate. Mr Bryson has presented at INCOSE, OMG, and OOPSLA conferences.