

To appear in, *Lecture Notes in Computer Science: Radical Innovations for Software and Systems Engineering*. Wirsing, M. and Knapp, A., eds., Berlin: Springer-Verlag, 2003.

A New Paradigm for Requirements Specification and Analysis of System-of-Systems

Dale S. Caffall and James B. Michael

Missile Defense Agency, 7100 Defense Pentagon, Washington, D.C. 20301-7100
Naval Postgraduate School, 833 Dyer Rd., Monterey, California 93943-5118
butch.caffall@mda.osd.mil, bmichael@nps.navy.mil

Abstract. In a system-of-systems, the number of possible combinations of interactions among the systems is theoretically infinite. System “unravelings” have an intelligence of their own as they expose hidden connections, neutralize redundancies, and exploit chance circumstances for which no system engineer might plan. In this paper, we propose a new paradigm for system-of-systems design. Rather than decompose each system within the system-of-systems in a functional fashion, we treat the system-of-systems as a single entity that is comprised of abstract classes. We demonstrate how our paradigm can be used to both avoid the introduction of accidental complexity and control essential complexity by applying object-oriented concepts of decentralized control flow, minimal messaging between classes, implicit case analysis, and information-hiding mechanisms. We argue that our paradigm can aid in the creation of sound designs for the system-of-systems in contrast to creating a federation of systems through a highly coupled communication medium.

1 Introduction¹

During the past decade, systems-of-systems have exploded into the battlespace of the joint and coalition warfighters. The acquisition community’s response in the U.S. Department of Defense to the rabid craving for more accurate information and more lethal functionality has been a less than stellar hobbling of various legacy systems and ongoing system developments through tightly coupled and lowly cohesive communication shackles.

While there are many issues with system-of-systems acquisitions, the first issue that we must address is the requirements definition and allocation issue. Just as the requirements issue continues to plague single-system acquisitions, the requirements issue is much more complicated in the acquisition of system-of-systems. For example, many of the systems that comprise a system-of-systems may be legacy systems

¹ This research is supported by the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

that operate as stand-alone capabilities in the operational world, having been developed with specific sets of requirements and with specific system functionality in mind. Additionally, just as we developed the legacy systems, we are developing new systems that will become members of a system-of-systems under similar conditions. That is, we are developing these systems as stand-alone capabilities with specific sets of requirements and with specific system functionality in mind.

Now comes the desire to slam these various systems together and connect these systems through some communication medium in the hope of achieving greater functionality, although this tack does not necessarily result in the intended synergistic effect. One can identify the systems that will form the system-of-systems, and then set out to bend, fold, spindle, and mutilate these systems in the fevered hope of producing a functional composition: it is difficult to think about the system-of-systems as a single entity, which may explain why system developers sometimes mistakenly focus on modifying individual systems with little deliberation and consideration for the system as a whole.

1.1 Current Approach to Developing System-of-Systems

Our tools for reasoning about a system-of-systems typically consist of little more than a “sticks-and-circles” diagram. The “circles” represent the various systems that comprise the system-of-systems while the “sticks” are means of information transfer, a messaging protocol, and, perhaps, a translator box to translate the messaging format from one system to another. Armed with this sophomoric view of the system-of-systems, we attempt to analyze and describe the system-of-systems through a trivial picture of the various systems as connected by a convoluted labyrinth of lines. Unfortunately, sticks-and-circles diagrams lack both a formal semantics and the richness needed to express the many dimensions of system behavior. Are the circles meant to represent systems, subsystems, modules, classes, objects, functions, hardware, or some other entity? Are the sticks meant to represent data flow, triggers, synchronization, calls, inheritance, or something else? [1]

Much too often, we initiate detailed design and coding from reasoning about the sticks-and-circles diagrams. During the development, we add new layers of features and functional enhancements to the system software without clear insight into the organization of the system software. Inevitably, the basic organization of the software that seemed so reasonable at the beginning of the development process begins to break apart under the weight of the revisions made to the system software. [2] Sadly, the software development becomes another casualty to report in future studies as to why software developments are not successful.

Traditionally, this methodology failed to achieve an interoperable and integrated system-of-systems. With each new failure, the system engineers attempted to “tighten up” the protocol standard; however, the system-of-systems did not achieve the desired degree of interoperability and integration. The end-state is a collection of systems that are tightly coupled with a realized protocol standard that only serves to greatly increase the system-of-systems software complexity.

As we have witnessed time and again, system software critical interactions increase as the complexity of highly integrated systems increases. In the complex system-of-

systems, these possible combinations are practically limitless. System “unravelings” have an intelligence of their own as they expose hidden connections, neutralize redundancies, and exploit chance circumstances for which no system engineer might plan. [4] A software fault in one module of the system software may coincide with the software fault of an entirely different module of the system software. This unforeseeable combination can cause cascading failures within the system.

Software complexity and size are dramatically increasing in our delivered products. Customers are demanding more features in their systems in less time than ever before. Under the demands of management, software developers scurry to coding with only a minimal of planning and reasoning about software architectures and system requirements. As a result of this mad rush to the goal line, software developers are stumbling and fumbling the ball—rarely scoring a touchdown. Unfortunately, software developers are building software products that have about a 26% chance of completing on time and on budget. For Government software projects, developers have about an 18% chance of completing their projects on time and on budget. Moreover, the delivered products will have fewer features and functions than originally desired by the customer. [8] Of great alarm to the Department of Defense, only 2% of the software was usable as delivered. [5]

1.2 A New Paradigm for Developing System-of-Systems

How do we reason about such a structure so that we have at least a modicum of chance to realize a functional system-of-systems? Can we extend the existing set of tools that we use in reasoning about a single system development to the more complex system-of-systems development? If true, can we use these tools to identify potential sources of accidental system software complexity?

A maxim espoused in all engineering disciplines is to “keep it simple.” The best we can do in software engineering is to minimize “accidental complexity” and control “essential complexity.” Accidental complexity occurs due to a mismatch of paradigms, methodologies, and application tools. Essential complexity is a fact of software engineering in that system software is inherently complex because software applications are the most complex entities that humans build. As system software is used in ways never envisioned by the developers, operators tend to demand that extensions be made to their system software. [7]

While we cannot address all of the issues that negatively impact the development (and follow-on maintenance) of system software, we will examine the issue of requirements specification for system-of-systems. Typically, detailed system specifications address merely the leaves of the system-decomposition tree. [9] Software engineers cannot develop a sound and complete software package from only the very detailed system specifications. Indeed, software engineers require several layers of abstraction beginning at the top layer of abstraction down to the very detailed system specifications. It is at the upper layers of abstraction in which software engineers reason about the system and make architectural and design decisions.

We argue that a new paradigm needs to be adopted by the acquisition community in which the system-of-systems is treated as a single functional entity during requirements specification and analysis. Our initial research to develop the new paradigm

has centered on the application of object-oriented design (OOD) techniques in conjunction with the Unified Modeling Language (UML), which we report on in the case study presented in the next section.

2 A Case Study from Missile Defense

We begin the discussion of our proposed paradigm by first introducing a hypothetical missile defense system: it is a system-of-systems made of legacy systems and systems to be constructed for providing new functionality for the composite system. Figure 1 is a sticks-and-circles diagram of our hypothetical system.

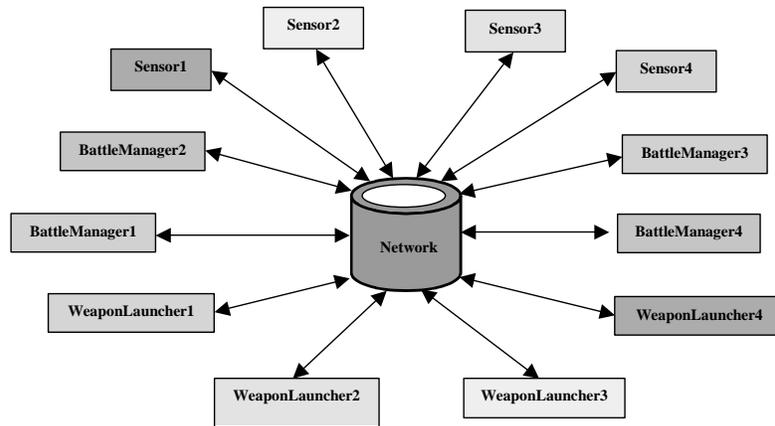


Fig. 1. Components of our hypothetical missile defense system

The greatest source of system software faults will occur in the integration of the various systems. With respect to our case study, the missile defense system will be a complex product that will contain many discrete software packages within each system. As a rule, these software packages will be developed independent of each other and programmed in many different languages. Additionally, the system will include legacy systems that are currently in operation. The means of integrating these elements and legacy systems are intricate tactical data links that support the message transfer within the system-of-systems.

It is difficult to reason about requirements and analyze the system-of-systems by relying on the sticks-and-circles view. Although presented as a single entity, it is challenging to understand the affects of requirements changes and component limitations in this view. As previously mentioned, our reasoning tendency is to focus on the individual systems of the system-of-systems in the hope that the desired functionality wondrously appears.

Unfortunately, magic and marvel are not tools that are abundantly available to system developers. Their fervent yet futile hopes for integrated systems and desired

functionality too often fall shattered on the road of broken acquisition dreams. Frustration and antipathy are the frequent products of system-of-systems development.

Let us propose another view of the missile defense system in which we apply UML and OOD techniques. We will develop a class diagram with abstract classes for the major components of the system-of-systems. We will reason about the class diagram in our attempt to develop subclasses to which we can begin to allocate requirements and analyze system capabilities and limitations. Additionally, we will identify message requirements and message flow in our attempt to reduce coupling in the system-of-systems by developing requirements for simplified interfaces between the components. Finally, we will propose a reassignment of methods to increase the cohesion of the components.

The object-oriented paradigm offers a new system-of-systems requirements and design methodology that provides for both minimizing accidental complexity and controlling essential complexity through the use of decentralized control flow, minimal messaging between classes, implicit case analysis, and information-hiding mechanisms.

While the hypothetical missile defense system will not be a pure object-oriented design, we can incorporate many of the principles of object-oriented design to decrease the complexity of the artifacts produced during the development of the system-of-systems. We believe that software engineers of system-of-systems can use this object-oriented paradigm to produce a sound design for the system-of-systems rather than the traditional federation of systems through a highly coupled communication medium.

2.1 Definition of Classes

The first step in modeling a system-of-systems using our paradigm is to develop a class diagram of abstract classes. For the hypothetical missile defense system-of-systems, we will use the following five classes, with the corresponding class diagram shown in Figure 2:

1. **Threat Missile:** The Threat Missile class is the enemy missile that contains warhead of mass destruction: nuclear, chemical, or high explosive munitions. The adversary will launch the threat missile within the confines of his state. The missile will climb into the exo-atmospheric region that constitutes up to 80% of the missile flight. The missile will re-enter the atmosphere over our forces or defended assets at which time it will impact at its aim point.
2. **Sensor:** The Sensor class is the object that detects the threat missile. Sensor is an abstraction of two subclasses: Infrared class and Radar class.
3. **BM/C2:** The Battle Manager/Command and Control (BM/C2) class processes track data from the sensor. The BM/C2 monitors the threat missile, develops firing solutions to negate the threat missile, and directs a weapon to launch its interceptor with the BM/C2-provided firing solution. The BM/C2 class is an abstraction for all system echelons of battle management.
4. **Weapon:** The Weapon class develops firing solutions, calculates the probability of kill, and implements the BM/C2 authorization to engage the threat missile.

5. **Interceptor:** The Interceptor class is the engagement mechanism that negates the threat missile. The Interceptor class is the abstraction for both directed and kinetic energy intercepts of the threat missile.

The message requirements in the above class diagram are very specific as compared to the single, large network interface in Figure 2. One can readily determine the messaging requirements of each class, such as the Sensor class determines the attributes of the Threat Missile class, the BM/C2 class needs formed track data from the sensor class, Weapon class waits for control data from the BM/C2 class, and Interceptor class waits for the interceptor release command from the Weapon class.

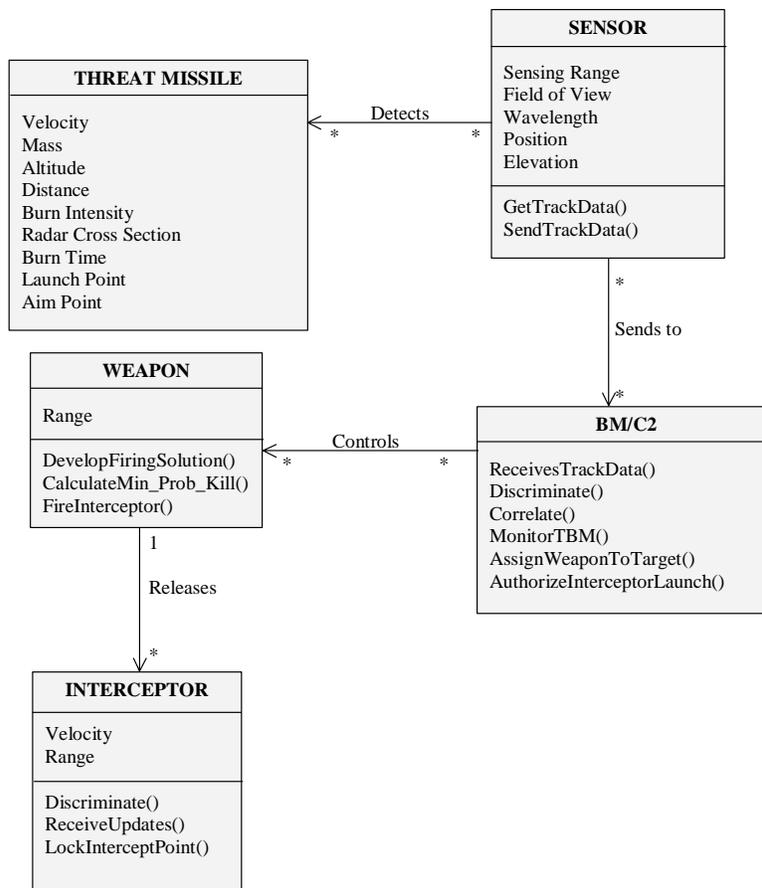


Fig. 2. Class diagram of our hypothetical missile defense system-of-systems

2.2 Definition of Abstract Interfaces and Subclasses

From the class diagram in Figure 2, we can begin to define abstract interfaces between the classes. Rather than the largely unmanageable and complex network interface of the sticks-and-circles diagram, we can begin to develop specific interface requirements from the class-diagram approach.

Let us add detail to the Threat Missile class as this is the point of reference for our missile defense system. We can develop subclasses (i.e., short-, medium, and long-range threat missiles) of the threat missile class as depicted in Figure 3.

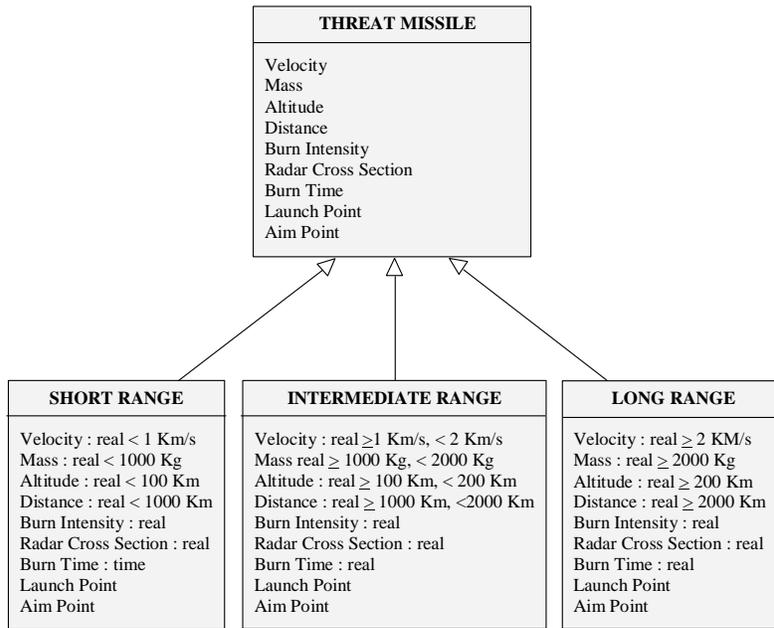


Fig. 3. Subclasses of Threat Missile class²

In our definition of the subclasses, we have assigned attribute values. In our example, we have assigned fictitious data so that our example remains out of the classified regime. These subclasses with the assigned attributes will form the basis for our reasoning about the missile defense system.

The sensor class is responsible for detecting the Threat Missile class, so let us develop subclasses that can detect the Threat Missile subclasses that we have defined. The subclasses for the sensor class are depicted in Figure 4.

By considering the subclasses of the threat missile class, we can design a sensor framework for which we can attain overlapping coverage of our sensor subclasses to greatly increase our opportunities for the detection of the threat missiles. We can also

² All attribute values listed in subclasses are fictitious and do not represent real threat missile data.

develop additional requirements to bolster our detection capability. For example, after considering the Threat Missile subclasses for a potential adversary, we may desire to increase the sensing range of the Sea-Based Sensor to extend our coverage into an adversary's territory into which a Ground Sensor solution is not feasible. We can now levy this requirement change on the Sea-Based Sensor subclass.

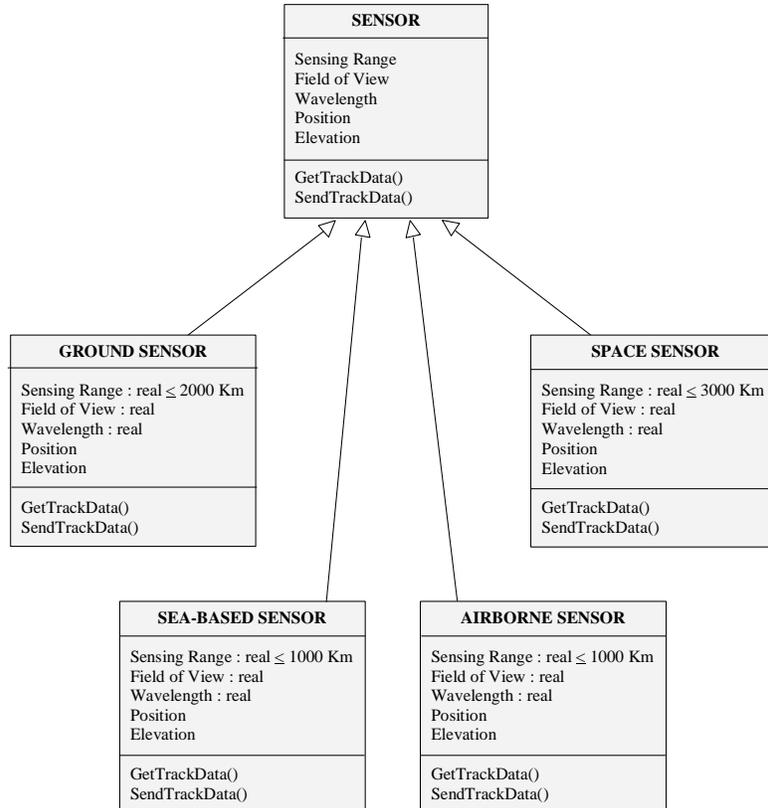


Fig. 4. Subclasses of Sensor class³

After we have detected a Threat Missile object, then we must develop a firing solution and engage the threat missile. As depicted in Figure 2, the BM/C2 class handles these functions and several other important functions. While these functions are related, the incorporation of these methods in a single class lessens the cohesion of the class. Rather than a single BM/C2 class, we might develop the BM/C2 class as an aggregate of several classes as shown in Figure 5.

As depicted in Figure 2, we separated the methods for developing and realizing a firing solution from the BM/C2 class and assigned these methods to the Weapon

³ The attribute values, as in Figure 3, are fictitious and do not represent real threat missile data.

class. These methods are similar in function so the cohesion of this class is high. This separation is important as the realizations of the BM/C2 class and the Weapon class may physically reside on different hardware platforms. So, in addition to increasing the cohesion, we reduce the coupling by substituting more interfaces that are small and better defined for the larger interface required for data flow and messaging of the sticks-and-circles architecture depicted in Figure 2. The Weapon class and its associated subclasses are shown in Figure 6.

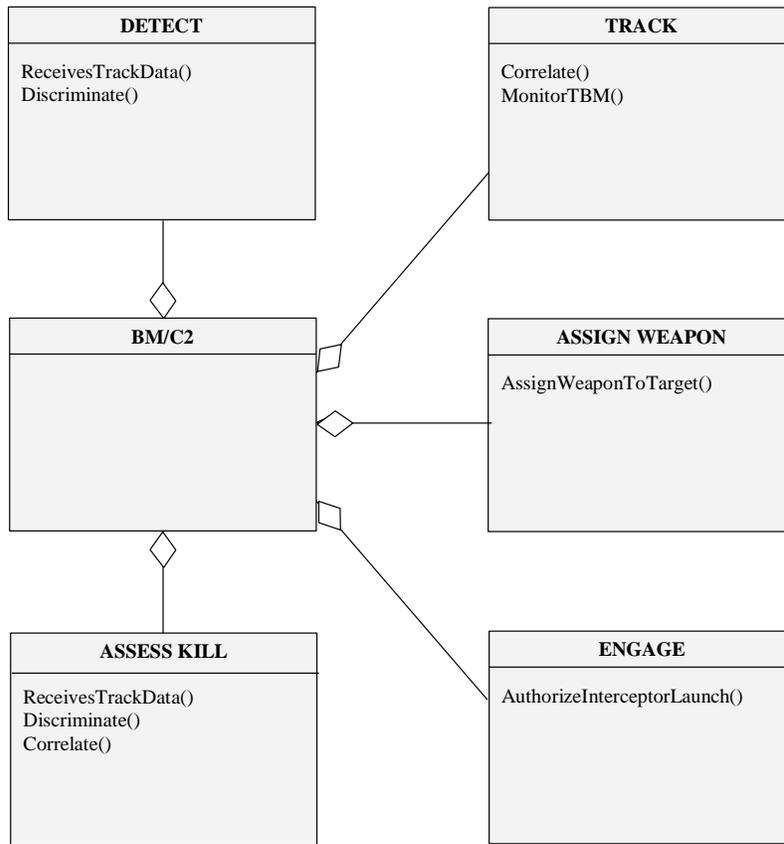


Fig. 5. BM/C2 class as an aggregate

Lastly, we consider the Interceptor class. Given the attributes of the Threat Missile class as well as potential deployment of our hypothetical missile defense system, we can develop the attributes and associated requirements for the Interceptor class. For example, the velocity of the Intermediate Range subclass of the Threat Missile class ranges between 1 km/sec and 2 km/sec and the distance of this same subclass ranges from 1000 km to 2000 km. As we consider the minimum altitude in which we must negate the threat missile to ensure minimal ground effects of the resulting debris, we

can determine minimum velocities for our three subclasses of the Interceptor class. These subclasses are depicted in Figure 7.

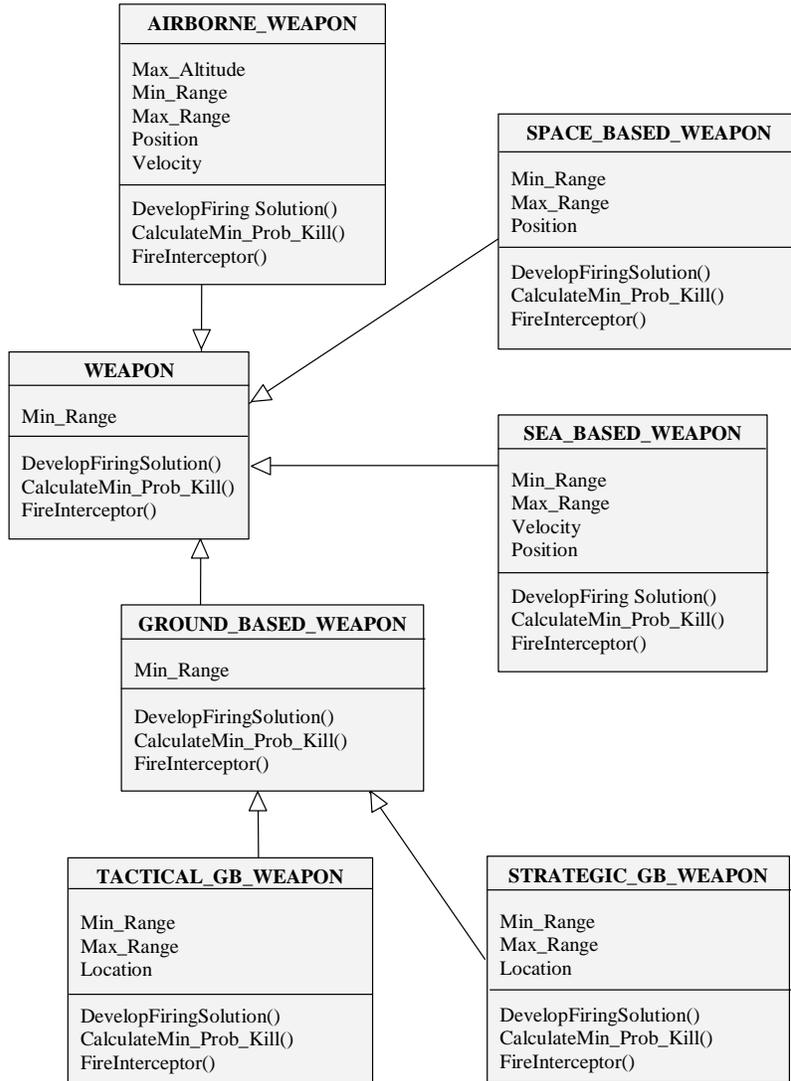


Fig. 6. Subclasses of Weapon class

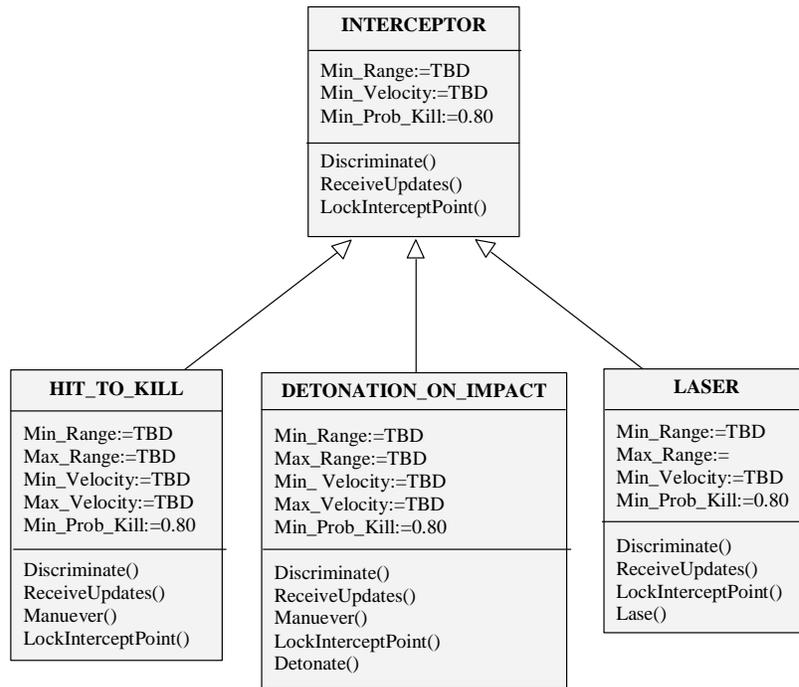


Fig. 7. Subclasses of Interceptor class

2.3 Minimal Messaging between Classes

As we reason about the classes and subclasses of our missile defense system, we can see that we will develop many interfaces in the realization that replaces the single, large network interface of the sticks-and-circles diagram of Figure 2. This is important to us in that we can manage a larger number of small, well-defined interfaces; however, the single, large network interface is much too unwieldy and complicated to manage effectively. We can reduce the messaging requirements of the large network interface to only that which is necessary for realizing the subclasses of our system-of-systems. Because the interface requirements are now manageable and known to all of the system developers, we have enhanced our ability to effectively integrate these systems into a system-of-systems.

Additionally, by treating the missile defense components as classes and developing concise interfaces that implement the minimum level of information sharing among the classes, we can define a data structure that implements data hiding. That is, by

reducing the message traffic among classes to only that which is necessary to complete the missile defense missions and functions, we can prevent external programs from inadvertently modifying the state of a given class or injecting superfluous message traffic that may cause undesired system-of-systems behavior. [6]

By defining a data-only interface strategy, we can greatly reduce the coupling of the missile defense components. A data-only interface design will result in a data-only integration realization. That is, each system within the missile defense system-of-systems will provide data that is suitable for transport and use by another system. Thus, the missile defense system-of-systems will exhibit the following properties [3]:

- More likely to work with legacy software code
- No build-time coupling in any system
- Missile defense systems are not required to share a common platform
- Missile defense systems can share a database to store exchanged data

A final benefit of realizing many small, well-defined interfaces rather than a single large interface will be the flexibility for incorporating future changes in a given class without negatively affecting the other classes. By data hiding and minimal message traffic, the software within a missile defense class is effectively independent in structure and realization than the other classes. As such, an internal software change to any single missile defense class should not affect any other class given that the interfaces among the classes remain unchanged. [6]

2.4 Inheritance and Decentralized Control Flow

As we define the class and subclass attributes, the concept of inheritance becomes important in that the allocation of requirements through attributes and methods ensures consistency in the realization of the subclasses in our developments. Each system developer will know the minimum set of requirements that must be implemented and each developer knows what requirements the other developers will realize.

By careful assignment of methods to each class, we can avoid the creation of the so-called “god class” that performs the bulk of the work within the system-of-systems. [7] Typically, we overload the battle manager function with the vast majority of the work. More often than not, the battle manager software contains many dissimilar tasks and requires a complex messaging network. Rather than primarily exchanging control or triggering messages among several classes, the typical battle manager requires the continual transport of great amounts of data that results in more complex rules of messaging and bandwidth requirements. By employing the aforementioned UML and OOD techniques, we can reassign methods to other classes in which these methods are better suited.

For example, consider the discriminate method listed in the BM/C2 class in Figure 2. This requires that the Sensor class send a great deal of data to the BM/C2 class. Perhaps we might reason that the Sensor class should contain the discriminate method and send a much smaller, refined track file to the BM/C2 class for prosecution. This would greatly reduce the messaging requirements and greatly simplify the interface between the Sensor class and the BM/C2 class.

2.5 Encapsulation

As we reason about the classes and subclasses of the hypothetical system, we find that we can modify the methods to maximize the benefits of data hiding within the appropriate class. In the large sticks-and-circles network of Figure 1, nearly all data is public by definition of the single, large interface to each system. By developing appropriate methods for each class, we can begin to hide data within its class.

For example, consider the development of a firing solution for a given threat missile. In the large sticks-and-circles network, the firing solution uses public data that is visible to all other systems. Because the data is public and the network connects each system to all other systems, it is difficult for software designers to understand the impact on system behavior as it is not readily apparent what system functionality is dependent on the public data.

On the other hand, we can determine the data requirements for the development of the firing solution in the Weapon class in Figure 6, and understand that the software developers should hide that data within the Weapon class. While this data hiding may be more difficult in procedural software, the public data issue is more readily apparent in the class views of the system-of-systems than in the large sticks-and-circles network diagram.

3 Conclusion

By applying UML and OOD techniques to the system-of-systems development, we can glean a great deal more insight into the system-of-systems requirements definition and allocation issues than with the conventional sticks-and-circles diagrams so often used to model these large, complex systems. By developing a class diagram with abstract classes for the major components of the system-of-systems, we can reason about the class diagram in our attempt to develop subclasses to which we can begin to allocate requirements and analyze system capabilities and limitations. Additionally, we can identify message requirements and message flow in our attempt to reduce coupling in the system-of-systems by developing requirements for simplified interfaces between the components. Finally, we can reassign methods to increase the cohesion of the components and we can hide data within a class to minimize the negative impacts of future modifications to either the system functionality or the data.

These aforementioned benefits of applying these UML and OOD techniques cannot be derived from the traditional views of system-of-systems designs. While software designers encounter other problems in system-of-systems designs, we believe that software developers can more easily reason about the system-of-systems requirements and associated allocation, thereby improving the system-of-systems architectures and designs by employing the techniques previously outlined in this discussion.

References

1. Bachman, F., Bass, L., Carriere, J., Clements, P., Garlan, D., Ivers, J., Nord, R., and Little, R. Software Architecture Documentation in Practice: Documenting Architectural Layers. Special Report CMU/SEI-2000-SR-004, Software Engineering Institute, Pittsburgh, Penn., Mar. 2000.
2. Constantine, L. L., *The Peopleware Papers: Notes on the Human Side of Software*. Upper Saddle River, N.J.: Prentice-Hall, 2001.
3. Garland, J. and Anthony, R. *Large-Scale Software Architecture: A Practical Guide to Using UML*. New York: John Wiley & Sons, Ltd., 2002.
4. Greenfield, M. A. Mission Success Starts With Safety. Presentation given at the Nineteenth Int. System Safety Conf., Huntsville, Ala., Sept. 11, 2001.
5. Lesishman, T. R. and Cook, D. A. Requirements risks can drown software projects. *CrossTalk* 15, 4 (April 2002).
6. Parnas, D. L. *Software Fundamentals: Collected Papers by David L. Parnas*. Reading, Mass.: Addison-Wesley, 2001.
7. Riel, A. J., *Object-Oriented Design Heuristics*. Reading, Mass.: Addison-Wesley, 1996.
8. CHAOS: A Recipe for Success. The Standish Group International, 1999.
9. Weber, M. and Weisbrod, J. Requirements engineering in automotive development: Experiences and challenges, *IEEE Software* 20, 1 (Jan./Feb. 2003), 16-24.