

# THE USE OF PRODUCT LINE BASED CHECKOUT SYSTEMS FOR PAYLOAD PROCESSING

*Phillip T. Meade, NASA, Kennedy Space Center, Florida*

*Curtis Myhand, Command and Control Technologies, Titusville, Florida*

## Abstract

Launching payloads into space continues to be a weekly occurrence around the world. Although it is becoming commonplace, it has not become a low-cost or low-risk, quick and simple process. Due to these conditions, concepts have been developed and efforts are underway to create a generic set of processes and tools that will advance payload processing into the 21<sup>st</sup> Century, making it more efficient and less complicated. These new approaches include developing more cost effective processes with shorter cycle times, and require fewer resources than in the past. The recently developed concepts and projects address the life cycle of payload processing through product line strategies, integrating agency and customer needs and requirements, and support geographically distributed processing.

Of these, the concept with the greatest impact to the launch site is the use of product line strategy. By converting payload processing systems to a product line based on a core system of common checkout technologies, large reductions in Operations and Maintenance (O&M), Sustaining, and Logistics costs can be realized. Furthermore, the increased compatibility between systems will enable the seamless interchange of data products and applications. Finally, long-term savings will be realized by enabling the creation of additional products in the line for the support of new programs while avoiding the reengineering of common assets.

For this goal to become a reality, however, care must be taken in the design of the architecture

upon which the checkout systems are based. This architecture must be constructed with the entire domain of checkout system applications within its scope. By combining this comprehensive architecture with generic software components, the various capabilities – including the front-end interfaces – can be abstracted from the design of the overall system, allowing for a configurable multi-use core system that is extensible to the various situation specific systems required throughout the lifecycle of a payload.

## Introduction

### *Background*

The John F. Kennedy Space Center (KSC) is the National Aeronautics and Space Administration (NASA) center of excellence for payload processing and launch activities. As such, it is the responsibility of the center to adequately test all payloads and integrate them into the shuttle for launch. The task of processing a payload for launch is very involved and requires the cooperation of many different organizations, companies and countries.

KSC is where the hardware is first integrated on to the Shuttle payload carrier and tested against certified test equipment. Many problems usually arise from database inaccuracies, as well as software and electrical interface incompatibilities within these highly complex systems. As a result, the payload is subjected to a series of verification tests. This “test and checkout function” is one of

the most manpower intensive portions of the pre-launch payload lifecycle. This function occurs two and sometimes four times at the launch site, per payload.

To accommodate these various tests, multiple checkout systems have evolved, each providing a different level of fidelity, flexibility, and capability. Additionally, new checkout systems have been developed to meet new programmatic needs, such as construction of a space station. In this case, the payloads are space station components possessing electrical interfaces and data formats incompatible with those of the shuttle. Obviously, this has led to a sizable infrastructure at KSC designed to adequately process shuttle and expendable launch vehicle payloads, International Space Station (ISS) payloads, ISS elements, and ISS Reuse and Reutilization (R&R) missions.

To further add to the proliferation of checkout systems, payload developers typically construct their own checkout systems for command and control of their payload during development, factory test, and on-orbit operations. Although these systems are built to provide similar functionality, each is a unique implementation whose construction and maintenance is a significant duplication of effort.

### ***Current Challenges***

The existing KSC and customer developed checkout systems were provided by separate programs with very specific goals in mind. As a result, these systems have disparate architectures, system software, applications, and user interfaces.

Furthermore, these systems with radically different architectures each require a separate and unique supporting infrastructure. Consequently, there is an O&M and sustaining group responsible for the support of each system. Logistics costs are also driven up by the need to provide unique spares for the various hardware platforms, as well as providing support for varying software packages. The uniqueness of multiple different systems makes them expensive to support collectively.

This situation is further complicated by the fact that the current KSC checkout systems are aging.

Based on 15-year-old architectures, the systems are stovepiped with intertwining system code, making them expensive to sustain and modify. This architecture approach is incompatible with the long-term goals of payload processing at KSC since it is inflexible and resists incorporation of new requirements and upgrades.

Modifying these existing systems often introduces significant risk and expense. These factors are exacerbated by the long development times, complexity of implementing changes, and extensive verification and validation (V&V) costs. These V&V costs are driven by two factors:

- The inability to minimize defects caused by software changes, and thereby reduce the amount of testing required
- The high level of confidence that is required to assure the proper functionality and pedigree of a checkout system used to process flight hardware; any new checkout system must undergo exhaustive testing to validate its credibility and ensure that it poses no threat to the flight hardware.

Finally, the process used to test a payload is affected by the design of the checkout system. One example is the inability to export data in support of geographically distributed teams. Testing processes are unnecessarily constrained due to this outdated architecture and corresponding limited capabilities. This results in inefficiency and reengineering throughout the payload lifecycle, further increasing both risk and cost

The most common example is the lack of continuity of data products. Due to the differences in the systems used to process a payload, the databases, displays, and applications cannot be shared between systems. This drives system-specific reengineering of these necessary products.

This duplication of effort is a problem closely related with the need to provide payload developers with a common, KSC certified, checkout system for developing payloads, and prevents the proliferation

of unique, customer-created, systems for each payload. In doing so:

- The cost and the risk to the payload developer could be avoided.
- KSC would be assured that customer-developed data products would be compatible with the launch site checkout systems.
- The higher fidelity of the common checkout systems would reduce the number of problems detected at the launch site.

This, however, is the topic of other papers [1, 2]. The above-mentioned challenges culminate to increase not only the cost to NASA but also the resulting cost to the customer.

## Product Line Solution

More and more software engineering organizations and communities have come to the realization that in developing single systems they are, in essence, repeatedly developing the same capabilities (i.e. “reinventing the wheel”).

KSC is guilty of committing this mistake as well. For instance, all checkout systems process commands and data, provide a user interface, and archive or record data. Every time a new system is developed, because the existing systems are inadequate for the new purpose (e.g. not flexible, scalable, portable, sustainable, etc.), the solutions for these capabilities are reinvented.

The problem is that the existing systems were not engineered with large-scale reuse in mind. It is universally acknowledged that there are significant benefits of achieving software reuse. Consequently, KSC must develop the infrastructure for the strategic reuse of software. How to accomplish this objective, however, has always been elusive.

Software engineering is an immature discipline. Indeed this has garnered the attention of professionals in government, industry and academic communities worldwide for many years. Yet for all of the attention, innovative technologies and tools,

none has proven to be the silver bullet. In 1987 Frederick Brooks postulated, “the very nature of software makes it unlikely that there will ever be an invention that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware. We cannot expect ever to see two fold gains every two years” [3].

In light of these challenges, software engineers at KSC (like many others worldwide) are embracing the over 20 year old concept of developing for system families [4] and mature methodologies that hold hope for answering many of the problems currently faced in owning multiple checkout systems. The use of a software product line for checkout systems would allow for the large-scale strategic reuse of software assets that is desired.

This reuse, however, is only part of the story. The true benefits result from the creation of a common checkout system platform. This platform would provide for the consolidation of infrastructure, and allow interoperability between systems, as well as the reengineering of key business processes.

A checkout systems platform would in effect be a collection of the common elements, especially the underlying core technology, implemented across all of our checkout systems. This would manifest itself in system architecture capable of supporting the common components of each checkout system within a common framework.

Once developed, these common components would form the basis of each checkout system. The application specific capabilities for each checkout system would then be added to complete the full set of functionality. In this way, the pitfall of trying to design a system that is all things to all people would be avoided. The acknowledged trade is that the user does not get to exactly specify a whole new system, but quickly gets most of the capability with high quality and reliability. Only those attributes that are common to all checkout systems in the domain would be incorporated into the common assets. The resulting systems would then be aligned with the goal of meeting the challenges currently faced.

## ***Benefits***

Software product line development allows the reuse of basic family requirements, models and analysis, architecture, design, code, test cases and procedures, and documentation. The process of producing a new system, called Application Engineering, becomes more of an integration effort than a development effort with only system unique requirements being newly implemented. The domain model, reference (generic) architecture and documentation are used as the basis for adapting the product family assets to produce the new system.

The benefits include the following:

- New systems can be fielded much more quickly than the typical 3-to-5 year development life cycles experienced with current systems.
  - New systems can be developed at a much lower cost and with fewer resources.
  - New system developments will entail much less risk as the needed cost, schedule, and resources will be considerably more predictable.
  - New system developers will be able to focus on the system specific problems at hand instead of re-implementing the basic functionality previously implemented for other systems.
  - Personnel can move seamlessly from project to project because the architecture, components, and development processes will be familiar to them.
  - O&M personnel will be familiar with all systems in the product line family allowing them to seamlessly support any.
  - New systems will be much more reliable as defects will have been identified and fixed in their ancestors.
  - New systems will have come with an established pedigree allowing for the elimination or minimizing of design certification activities.
- Processes, procedures and data can be shared across all of the systems of the product family.
  - Better training material and documentation, which can be shared.
  - Sustaining engineering costs can be shared across all projects.
  - Logistics costs can be shared because of the common architecture.
  - Upgrades, enhancements and improvements to the common assets can be realized across all systems of the product line.
  - System specific applications and products with applicability to the product family can be added to the common set of assets for other systems to reuse.
  - Configuration management, requirements management and CASE tools can be shared across projects.
  - Money and resources spent on the development of checkout systems can now be diverted to more strategic goals
  - The development of an applications framework to greatly reduce applications development costs now becomes worthwhile.

## **Implementation Plan**

The expectations regarding the development of KSC checkout systems have always been high. Mastering complexity to quickly provide high quality systems that facilitate effective maintenance, evolution and reuse has always been a key expectation. Unfortunately, like the vast majority of large software projects, our development efforts have not met this expectation. Software product lines are a promising solution because:

- Through our own lessons learned, we reached the conclusion that we should develop for families of systems prior to ever becoming aware others were developing software product lines.
- There have been a number of successes through the ad hoc reuse of one of our systems, the Partial Payload Checkout

Unit (PPCU). The PPCU was originally developed to verify partial payload interfaces for the Space Shuttle program. It has since been adapted for reuse 3 other times, including the launch system for the Delta Clipper (the Real-Time Data System) and the primary checkout system for International Space Station elements at KSC, the Test, Control and Monitor System.

- They have been proven to be successful, and there are a number of highly touted implementations [5].

The challenge is how to assure a successful implementation. We address this by first concentrating on the development of strategic plans for attaining the desired quality characteristics for the product line, adopting new technologies and processes to maximize the effectiveness of the product line, and managing of the effects of technological change. KSC has a great deal of experience performing these activities and generally does them very well.

### Strategic Planning

It is important for an organization to have clearly defined objectives and detailed low-level plans for meeting those objectives in order to be successful. Personnel at all levels of the organization must also understand the significance and priority of each objective. To reduce the overall life cycle cost of payloads, NASA must reengineer its current payloads business processes to:

- Increase synergy and leverage resources across activities
- Reduce operations, maintenance and sustaining costs of checkout systems
- Reduce costs of future system modifications or development

These goals must in turn be refined to a level (see Figure 1) where a direct relationship to the key requirements for the quality attributes of the product line can be established. Quality attributes are the characteristics that comprise the qualities of service

(e.g. performance, usability, and dependability) and the development qualities (e.g. reusability, modifiability, and portability) of the product line. Quality attributes exist in all systems to some degree, either deliberately or, if not considered, by accident. It is by these attributes that the product line and subsequent products (i.e. specific checkout systems) will be judged by all stakeholders.

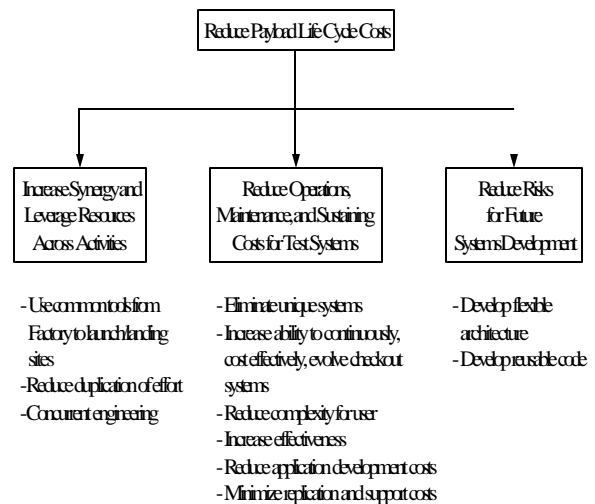


Figure 1. Strategic Goals

The key quality attributes of the product line are summarized as follows:

- Reusability - The product line must provide the foundation for systematic, large-scale reuse of system assets including requirements, architecture, components, test cases and documentation.
- Scalability - Systems can be scaled to execute on a single laptop computer supporting a single Input/Output (I/O) interface to a large, distributed, multiprocessor system capable of supporting many interfaces. Scaling the system can require configuration changes but does not require system software modifications.
- Portability - The use of open system standards is strategically implemented to

the maximum extent feasible. The system provides insulation from the operating system and Commercial-Off-The-Shelf (COTS) solutions for portability.

Portability is measured by the amount of effort required to re-host the system on different platforms. Generally, increased portability should not be acceptable at the expense of increased recurring costs (maintenance, training, etc.).

- **Modifiability** - The system is designed to maximize flexibility (accommodate change). Modifiability includes or is related to extensibility, maintainability and modularity, and is greatly affected by software coupling and cohesion. Changes can occur as the result of new COTS versions or products, new or improved standards, new requirements, or defect fixes. The measure for modifiability will mostly be how easily changes can be made and whether changes can be localized with little ripple effect to the system at large.
- **Performance** - Performance is the measure of latency and throughput.
- **Security** - Access to the system including its constituent components, capabilities, and data products is strictly controlled and the systems are protected under all circumstances.
- **Usability** - The system is user-friendly providing an intuitive environment through conceptually consistent interfaces without unnecessary complexity. Usability includes the ability to readily configure the system.
- **Dependability** - This is the confidence that reliance can justifiably be placed in the services provided by the system. The measures of product line dependability emphasize availability, reliability and the system's continuity of service.

The quality attributes are often very interrelated and highly affected by architecture and implementation decisions. It is important to

understand their interrelationships, prioritize and develop a strategic plan for realizing each attribute, and document the necessary requirements to assure the plans are met. Without the proper focus on the quality attributes, they are often easily and inadvertently compromised during day-to-day activities or discarded as problems jeopardize significant deadlines. Therefore, the implementation strategy for each attribute must be well understood by the development team, especially the decision-makers, as they must influence design, policy, and process decisions for the product line to meet requirements and exist as advertised.

### ***Adoption of New Technologies and Processes***

How to develop software that is reusable on a large scale has always been elusive. "In the early days of Object-Oriented (OO) development, there was the belief that objects are reusable by their very nature and that reusable OO software simply "falls out" as a by-product of the application development. Today, the OO community widely recognizes that nothing could be further from the truth. Reusable software has to be carefully engineered and engineering for reuse requires a substantial investment." [6]. The software architecture provides the foundation for this investment. The architecture provides a reference model that can be reused in the development of new systems. Also, it is in the development of the software architecture that the satisfaction of the quality attributes must first be addressed. Architecture development is where the trades will be made concerning the degree to which the quality attributes are implemented. However, engineering for a family of systems requires a new approach to architecture development: the two life cycle model. In developing our product we propose to use two life cycle phases, Domain Engineering and Application Engineering. Domain Engineering defines the scope of the product line and develops the analysis and generic design artifacts that are common to all family members of the product line. These assets provide the basis from which specific checkout systems will be built by performing Application Engineering. Application Engineering follows a more normal system development life cycle except

for the reuse of assets developed during Domain Engineering.

The Domain Engineering life cycle, which is unique to product line or system family development, is comprised of three phases: Domain Analysis, Domain Design and Domain Implementation. During Domain Analysis, the scope of the checkout systems domain is identified and the domain model for the checkout systems product line is developed. The domain model results from an analysis of legacy systems and the vision of what the next generation checkout systems should be. It is through Domain Analysis that we are able to determine the common and variable features of the family members and the dependencies between these variable features. This is a mandatory activity to engineer software for reuse.

Reusable software must account for greater variability than would normally be considered. Feature modeling is the technique for doing so. A feature is a user-visible characteristic of the system. Domain Analysis includes feature modeling in order to identify only those relevant features and variation points required for the product line. Higher maintenance costs are the result of omitting relevant features and variation points. However, the penalty for including many unused features or variation points is also higher maintenance and development costs. Consequently, it is very important that the Domain Analysis identify the proper features for inclusion.

To perform Domain Analysis, we tailor and augment the Feature-Oriented Domain Analysis (FODA) method developed at the Software Engineering Institute (SEI) at Carnegie-Mellon University. FODA emphasizes the identification of prominent or distinctive user-visible features within the systems of the product line. These features are the requirements implemented for each of the systems and provide the basis for the elicitation of requirements. The FODA method has a well-defined process, establishes specific products for later use, and is supported by case studies that exemplify its use.

Four sub-models, the context, features, information, and operational models are defined by the FODA method. These models provide distinct views of the basic product line, its function and its behavior. These models are represented in an implementation-independent manner that allows decisions concerning the mechanisms for implementing variability to be kept out of the analysis model.

The operational model will be developed using the Coats-Mellon Operational Specification (CMOS) method to define user scenarios that represent a complete and accurate model of the system behavior. System experts are used to provide information regarding the legacy systems in the product line domain and verify the initially developed models.

The FODA and the CMOS permit the representation of models that can be easily understood by the system experts so they may concentrate on the content of the models. Additionally, the CMOS model allows for the derivation of verification and validation test cases prior to detailed design.

The purpose of Domain Design and Domain Implementation is to develop the common architecture with implementation components for the product family. The development of a product line architecture is ideal for KSC checkout systems as they are normally designed with the flexibility to support systems with long life spans.

The SEI has developed a method for developing reference (conceptual) architectures that is dependent upon the identification of architectural drivers for the product family. Architectural drivers are the functional, quality, and business requirements of the product family. The method, called the Architecture Based Development (ABD), can be initiated once the architectural drivers have been identified (prior to even the completion of requirements elicitation and analysis), and emphasizes the use of architectural styles (patterns) to realize quality and business requirements.

The most frequent reason for the development of new KSC checkout systems has primarily been

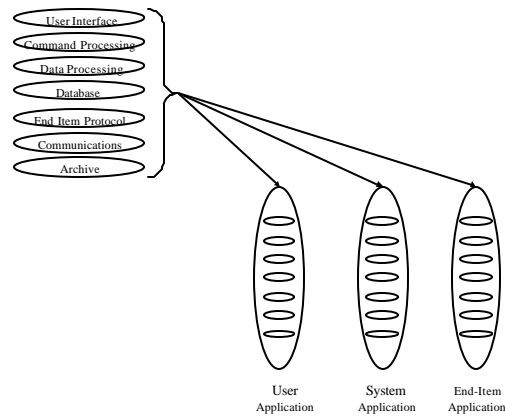
the inflexibility of existing systems that cannot be changed to meet new requirements in a cost-effective way. Therefore, it is very important to assure the architecture permits the quality attributes affecting the evolution of the product line to be met. These quality attributes are reusability, scalability, portability, modifiability and performance.

### Reusability

The purpose of product line development is to attain reuse, and having a common architecture provides a structure for doing so. Generic C++ design techniques permit the development of highly general and abstract components that can be integrated in many ways to produce very efficient code [7]. Additionally, OO analysis and design techniques are used to apply design patterns to solve recurring design problems [8]. Highly orthogonal implementation components, capable of being combined in as many ways as possible, are then designed to maximize reuse [9].

### Scalability

The architecture provides scalability through the use of a hybrid approach in which applications inherit their processing from functional components as shown in Figure 2. This approach, proposed by Lawrence Finch in 1998, increases scalability and modifiability [10]. Through Domain Analysis, horizontal components are identified and organized around functional parts of the system such as the user interface, database interface, command processing, data processing, and archive components. These components are then included in self-contained entities implementing business requirements called vertical components. For example, a vertical component could be an IEEE-488 end-item component that handles everything from the device driver interface to processing commands and measurements to updating the user interface. This approach provides great flexibility and also allows for reuse.



**Figure 2. Approach for Increased Scalability**

### Portability

The portability of software systems is affected by so many factors that a road map must be developed. The use of a layered architecture is currently one popular technique for attaining good portability and modifiability. Another is the use of open system standards as defined by the POSIX 1003.0 Institute for Electrical and Electronic Engineers, which is devoted to POSIX, the Portable Operating System Interface for UNIX.

New, to us, is the idea of uncoupling or loosely coupling applications, databases, displays, and Commercial-Off-The-Shelf items in general. These items are normally developed independently of the checkout systems software and tend to evolve in relatively independent cycles of their own. Innovations like application frameworks make this possible without the loss of performance, and technologies like the Extensible Markup Language, XML, provide flexibility in formatting and distributing data.

Finally, often over looked development tools, such as language compilers and libraries, must also be considered. These tools, like the GCC or Qt for graphical user interfaces, support the portability of the products they generate.

### Modifiability

The architecture must permit the maintenance and extension of the product family capabilities with relative ease. A layered architecture is used to partition the software into units that are only permitted to use the services of lower level layers.

Layering provides cohesion and helps to limit possible changes from affecting multiple layers. OO design techniques support this layering approach by making layer interfaces public. Generic design techniques, popularized by the C++ Standard Template Library implementation, are also an extremely important innovation as they permit the development of generic containers and algorithms that can significantly enhance extensibility and reuse.

### **Performance**

Given the long life spans of KSC checkout systems, optimizing for performance can, within reason, increase the usefulness of the product line. Performance modeling and profiling can assure the system is as usable and scalable as desired. The architecture permits concurrency through the use of lightweight processes implemented as POSIX threads (Pthreads). Los Alamos National Laboratory has proven, through the implementation of an application framework, that C++ and Pthreads are viable for the execution of high performance applications once limited to supercomputers [11].

### ***Development Strategy***

Currently, there is a need for a physically portable checkout system capability that cannot acceptably be satisfied by adapting existing KSC systems. This necessitates the development of a new system called the Payload Test and Verification System (PTVS). The PTVS is the catalyst for developing a product line and represents the first in a new generation of checkout systems.

As the foundation for future checkout systems, PTVS development has been very deliberate. Initial concepts were developed over 2 years ago, highlighted by analysis activities geared towards the identification of the important quality attributes, lessons learned from owning the legacy systems, and candidate system technologies. Last year, a pathfinder effort was commissioned to:

- Document the concept of operations or vision for the “ideal” system based on experience with the legacy systems. This is extremely important to improve the

effectiveness of next generation checkout systems and the efficiency of their operations.

- Further refine the design concepts.
- Identify risk mitigation strategies.
- Produce a project plan, including a strategy for migrating legacy systems to the product line or extracting legacy system assets for reuse in the product line.

Using the two life cycle model, the development includes a Domain Engineering life cycle and an Application Engineering life cycle. The Domain Engineering life cycle is further subdivided into systems engineering and analysis, and production phases.

During Domain Engineering, an analysis of the key legacy systems is performed in conjunction with the Domain Analysis effort to identify strategies for the evolution of those systems to the product line. In-depth analysis of these systems can be very costly. Therefore, the initial goal of the legacy systems migration analysis is to identify the technical and economic feasibility of supporting the reuse of various legacy applications, databases and displays with the product line. The objective is not to adopt technologies from the legacy systems for long-term use in the product line. It is to allow the product line to accommodate the systematic migration of the legacy systems to the product line or to support the evolution of legacy systems until a time when they are no longer needed and they can be decommissioned.

Following Domain Analysis, the development of the architecture and common product line software proceeds in a more traditional manner, with the exception that architecture and detailed design must accommodate the required feature variations. To achieve the highly desired flexibility and reuse, generic design techniques are used to develop general or abstract components that allow variation points through generic programming and OO techniques such as type parameterization and polymorphism.

The Application Engineering phase, following the Domain Engineering phase, is performed in a traditional manner following the spiral model and develops the PTVS-specific capabilities. Once the new technologies for the product line are used by PTVS and thoroughly tested and proven, the capabilities can be used for methodically migrating legacy systems to the product line or incorporated for use in the legacy systems. This is deemed the best way to manage risks to the heavily used and safety critical systems.

Admittedly, this plan results in additional cost to the PTVS, and adds an additional checkout system platform. However, it is believed that these additional project costs will be recovered by the payback from savings once the migration of existing systems is complete. The addition of another platform is only a temporary step in the planned migration or elimination of existing platforms.

## **Agency Implications**

The business case for proceeding on with a project such as has been described here will have to be made. Undoubtedly, this case will be made to encompass the entire space center, justifying the cost based on expected return on investment seen at KSC. However, the real potential of such a project can only be truly appreciated at the agency level.

NASA has realized that this is true of many of its programs, and has organized the agency around “Centers of Excellence” intended to focus and lead the effort for a particular agency initiative. This strategy helps to reduce duplication of effort and multiplication of costs.

The concept of a checkout systems product line integrates very well with this concept. There is a need at multiple NASA centers for a checkout system capability. Obviously, the redundant development of common capabilities at multiple centers drives agency costs. Furthermore, the basic platform requirements for a checkout system can be further abstracted to a command and control platform. This increases the potential synergy that could be achieved by an agency wide initiative to institute product line processes.

By establishing a generic or reference command and control platform at the agency level, and then creating derivative platforms for more specific applications like checkout systems, the agency could maximize its reuse of assets while decreasing duplication of efforts, and ultimately costs.

In addition to the financial benefits offered by such an approach, there is an inherent strategic benefit that can be derived. By abstracting specific checkout system and even command and control technological implementations to an overriding product-platform, the agency will have the ability to control the direction of these core technologies at a strategic level. Since the core technologies of the platform ultimately drive the direction of the resulting products, this approach would allow NASA to link its strategic objectives in this area to the product-platform.

Consequently, an achievable method of controlling the direction of multiple products produced by multiple programs at multiple centers would be achieved. The result would then be a more effective implementation of NASA strategic vision.

## **Summary**

It is no longer economically feasible for NASA to develop and maintain unique one-of-a-kind checkout systems. The infrastructure required to support such systems requires too much time, money and resources. By using the product line approach, leverage can be gained through the reuse of previously developed assets to accrue the benefits in virtually all phases of the project life cycle.

The use of software product lines holds the promise of greatly expanded reuse of software, allowing for reduced risk, cost and development time, while creating a product-platform for checkout systems. This platform would enable the consolidation of system-unique infrastructure such as O&M, sustaining, and logistics while providing large scale interoperability between systems. This interoperability, combined with a portable checkout system capability, would allow for the restructuring

of business processes, resulting in greater operational efficiencies.

A successful implementation will ensure these benefits are truly realized. However, the development of the product line need not be a high-risk endeavor. Experienced software system designers have a variety of mature, well-documented, technologies and processes available for the implementation of software product lines. Placing emphasis on identifying, planning, and attaining the desired quality attributes is as critical to the development of the product line as is the strategic use of the mature and emerging technologies.

If care is taken in planning and executing the development of a checkout system platform, the rewards will be great. This project has the ability to reduce costs while strengthening KSC's alignment with NASA's strategic goals. It directly supports the strategic goal:

*"Enhance core capabilities (people, facilities, equipment, and systems) to meet NASA objectives and customer needs for safer, better, faster, cheaper development and operations of space systems."*

If we are to surpass our current limitations and go beyond the boundaries of earth, we must make access to space easy and efficient. A significant part of the cost of access to space results from ground processing. It is imperative that we leverage our experience, and the experiences of others to improve our systems and processes in support of this higher order goal.

## References

[1] Jacobson, Craig, Phillip T. Meade, February, 2001, "Streamlined Payload Processing In The 21<sup>st</sup> Century," Space Technology and Applications International Forum, Albuquerque, NM

[2] Meade, Phillip T., Jacob Heuther, David Headley, Kevin Zari, 2000 "Reinventing Payload Processing at Kennedy Space Center," KSC whitepaper

[3] Brooks, Fredrick P. Jr., April, 1987, "No Silver Bullet" Essence and Accidents of Software Engineering," Computer Magazine

[4] Parnas, D.L., March, 1979, "Designing Software for Ease of Extension and Contraction," IEEE Transactions of Software Engineering

[5] Brownsword, Lisa; Paul Clements, October, 1996, "A Case Study in Successful Product Line Development," Software Engineering Institute Technical Report, Pittsburgh, PA, <http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.016.html>

[6] Czarnecki, Krzysztof; Ulrich W. Eisenecker, 2000, *Generative Programming: Methods, Tools, and Applications*, First Edition, Addison-Wesley

[7] Alexandrescu, Andrei, 2001, *Modern C++ Design: Generic Programming and Design Patterns Applied*, First Edition, Addison-Wesley

[8] Gamma, Erich; Richard Helm; Ralph Johnson; John Vlissides, 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, First Edition, Addison-Wesley

[9] Batory, Don, October, 1998, "Product-Line Architectures," Invited Presentation, Smalltalk and Java in Industry and Practical Training, Erfurt, Germany, <ftp://ftp.cs.utexas.edu/pub/predator/stja.pdf>

[10] Finch, Lawrence, May, 1998, "So Much OO, So Little Reuse," Dr. Dobb's Journal,

[11] Crotinger, James A.; Julian Cummings; Scott Haney; William Humphrey; Steve Karmesin; John Reyenders; Stephen Smith; Timothy J. Williams, 1998, "Generic Programming in POOMA and PETE," Seminar on Generic Programming, Dagstuhl Castle, Germany