Program of Reuse at Manufacturing Productivity Operation

Alvina Nishimoto

Hewlett-Packard Company
Manufacturing Productivity Operation
5301 Stevens Creek Blvd. M/S 51U/92
Santa Clara, CA 95051
408-553-3682, alvina_nishimoto@hpc700.desk.hp.com

Abstract

This paper describes a program of reuse that began in January 1984 with a new MPO product and has evolved to include three other existing products and all new enhancements to these products over the last seven years. The topics covered include:

- How the program was initiated and how our definition of reuse has changed over time
- How the program evolved to include other already existing products
- Development of the fix process for reuse components
- Coordination issues between products and the formation of the Shared Components Council
- Results of reuse metrics
- What lessons we learned and suggested areas for improvement

Major emphasis is on what we learned along the way and how our reuse program has evolved to include other products.

Keywords: re-engineering for reuse, reuse maintenance process, reuse councils, reuse metrics

Getting Started

In January 1984, Manufacturing Productivity Operation (MPO) launched a fast track new product for just-in-time manufacturing, HPJIT, to be completed in nine months. From the start, the R & D team drew up several objectives for the project:

- To invent a high-quality software product where there was none before, either internally or externally.
- To meet all scheduled milestones, the most important being the manufacturing release date, which was stated at the beginning of the project.
- To increase the productivity of engineers in designing this type of software product.

Several members of the project team had worked on the division's flagship product, Materials Management (MM/3000). Written several years ago, these engineers had encountered the difficulty in supporting this product. We had "cloned" so much of the product, meaning that a similar routine could exist in as many as 100 different transactions or batch programs.

We had many service requests that we classified as "generic", indicating that this problem existed in many transactions or batch programs and would involve a large effort to repair. Our dilemma was that although the code was almost the same, we still had to touch all the pieces of code that had been cloned to fix a defect. Therefore, at the beginning of the project, we decided to spend some of the project's development time creating some reusable code.

We placed many of the common calls to our tools layer inside of utility subroutines that could be called from any of the product modules (see Figure 1). We wrote this reused code for data base searches, screen handling, and transaction logging, as well as other more specific HPJIT functions. By analyzing cloned code, we produced reuse code and essentially did domain analysis at an ad hoc level.

The initial impetus behind the development of this reuse code was to avoid duplication of Pascal source code, but we also designed each routine for maximum flexibility so as to be usable by transactions and modules yet to be developed. Through the use of flexible common utilities, we newrly eliminated the duplication of Pascal source code throughout the HP JIT software.

Expansion to Other Products

After the manufacturing release of the product, a couple of the team's members moved on to work on the rewrite of MM for a new operating system. This product was called MM-JIT/XL. Much of the reused code written for the HPJIT product became the basis of the rewrite. We felt that this reuse code could do a lot to increase the productivity and quality of the project.

Interestingly enough, we found out that using reused code for another product involved a certain amount of initial investment to begin with. Some of the areas involved:

- Training to help people understand when and how to use the reused code. Unless people knew what reuse code was available and how to use it, it was impossible to expect engineers to write reusable code.
- Code templates to provide engineers with a framework for which to write a transaction. This also turned out to be a form of reuse code showing how the reuse code should be put together

Figure 1

to create a standard transaction. We found code templates essential in the beginning, but once we had a working transaction, engineers used these transactions as their code templates.

- Reuse code changes to support the new and larger scope of functionality and user interface. The rewritten product was a much larger product than HPJIT and had a different user interface. With such a large product, it became very difficult to anticipate all changes to reuse code ahead of time. We found the utilities changing a lot during the initial few months of the project or anytime a new type of transaction was developed.

- An initial investment of six engineering months with ongoing additional effort of six engineering months over a year and a half. In contrast, initial investment of HPJIT was one engineering month with ongoing additional investment of six engineering months over a five month period.

- A new set of responsibilities that required engineers to make ongoing modifications to reuse code as well as their code deliverables. In addition, senior engineers served as what was called module coordinators. Their responsibilities included facilitating communication, promoting design consistency across the product, and optimizing engineering productivity. Module coordinators raised engineers' awareness of reuse opportunities through the inspection process and could prevent an engineer from proceeding to the next stage of development.

- Making reuse a part of the development process. We set up metrics to help measure our progress. Our design and code inspection defect categories involved the following:

  - Improper utility (reuse) usage
  - Redundant code
  - Creation of new utilities (reuse code)
  - Not using an already established utility (reuse code)

We also measured the NCSS (non-commented source statements) for the reuse code versus unique code. Figure 2 shows the breakdown of the code for each of the mini-releases (MRs, releases to our development partners). Architecture and application utilities categories represented the reused code and generated code was code created through a code generator.

Despite the fact that MPO cancelled this project to rewrite MM/3000, we took this reused code and some of the transactions from the rewrite as the basis for a new enhancement to MM/3000. This new enhancement added functionality needed by process industry manufacturers and MPO released this product in October of 1988.

Figure 3 shows distribution of the code for this enhancement to Materials Management called Process Repetitive Option (PRO). The architecture category represents the reused code; the shared code represents reuse code that are not callable routines but is shared via include files; and generated code is code created through a code generator. With shared and utilities this enhancement had a reuse percentage of 41.4%.

Figure 2

Figure 3

Fix Process for Reuse

The fix process for supporting a reuse product could be characterized by the following:

- Although reuse code tends to create smaller code units allowing for smaller deliverables, there are more opportunities for coordination and dependency issues. This situation requires a greater amount of teamwork and egoless contribution since engineers become more dependent on each other. It also requires a commitment to reuse and flexibility to take schedule slips and unanticipated changes.

- With more code components, it was sometimes difficult to identify a component that caused the defect. Luckily, we had a trace facility built in our tools layer that gave us the ability to trace our reuse code without code changes. With very minimal code changes, this trace facility also gave us the ability to name the reuse code in the trace. We gradually added these code changes to the reuse code when we repaired the code.

- Initially, when new products added the reuse code, there were many changes in the reuse code required to accommodate various differences in the transaction processing in these new products.

  Because of the testing effort involved with these utilities, we often got to a point where we did a "reset" and fixed all backlogged change requests and tested all at once. Since the volume of code changes was large, a formal check out and check in of the code was not done.

  In some cases where the rework effort was too large, we replaced a piece of reuse code with a new routine, essentially obsoleting the old code. Whenever we scheduled changes in code that used the old routine, we would evaluate whether the rework to call the new routine could be done at the same time.

- As more products began to use the reuse code, we found that it became necessary to isolate the product dependent parts of the code to separate include files so that the same utility could be used by a new product without much disruption to the products already using the utility. These include files usually contained product specific information in a table.

  An example of this was a utility that displayed information on whether a dataset record had been retrieved or not. We made the utility generic across all products, but the details as to the user interface for the error handling of each product sometimes varied, and we placed this in a product-specific table.

- As the number of transactions and programs using reuse code grew, the testing effort became an issue. We often could change the code rather easily but had difficulty assessing the impact to other products, and as a consequence, we spent a lot of time on the testing effort.

  We, in the beginning, felt that we needed to test each affected transaction or program. As it became more and more obvious on the impracticality of this approach, we found we needed to better assess the impact to other products and only test the high risk components and/or those components that gave us the most variation in pathflow in the reuse code.

Reuse Inspection and Metrics Analysis

Encouraging the use of reuse code was included as an important part of our inspection process. Our design and test plan inspection included the following cause codes related to reuse:

- Design Change
  - Work-in-process utilities that should be considered
- Enhancement
  - Logic that should be made into a NEW application or architecture utility
- Utility Usage
  - Defects in the parameter list
  - Defects in initialization of the parameter list
  - Incorrect usage of a utility
  - Not using an established utility

Figures 4 and 5 show examples of the utility usage category for mini-releases releases to our development partners. The design change and enhancement categories also show up, but include areas other than reuse and unfortunately was not broken out. As we became more comfortable with our utility usage that category decreased, so much that we do not even track the category today.

We also track service requests from our customers that occur in reuse code and have found that the majority of the known problems tend to occur in the unique code and not in the reused code. For the process industry enhancement to Materials Management, customers reported 27 defects in 69,300 NCSS of reused and generated code versus the 56 defects reported against the 13,500 NCSS of unique code. This results in approximately .4 defects/KNCSS in reuse code versus 4 defects/KNCSS in unique or a ten times improvement.

Current Situation

MPO today has the following number of shared components:

-	Shared (include files rather than callable routines)	9
-	Utilities (callable routines)	200
-	Internal documentation for shared code and utilities	209
-	Data dictionary elements (approximation only)	40
-	Shared messages	210
-	Testing tools (created by MPO)	16

Figure 4

Figure 5

We have shared and reuse code broken down as follows:

-	Shared (include files rather than callable routines)	16,100 NCSS
-	Utilities (callable routines)	24,900 NCSS
-	Generated	28,300 NCSS

In addition to the above code, we have a tools layer of reuse code (not included in the above numbers) that provides the interface between our data customization and the application programs.

As more and more products began to use the reuse code, coordination issues began to cause unanticipated repair due to poor impact assessment of a change. It was fairly easy for an engineer to assess the impact of their change in the product they were working on, but very difficult to assess the impact to other products. As a result, we found problems late in the process when the other product picked up the change and schedule delays resulted.

As this situation became more critical, we organized a group called the "Shared Components Council". We expanded the definition of reuse to be called shared components, since we found that reuse did not involve just code. We set up the following objectives for this group:

- Continue to evolve a process that allows for different groups to work on shared components for different application release cycles with a minimum of impact and rework to each application.
- Communicate changes for shared components and proactively assess the impact to each application.
- Foster a process that allows us to continually add to our library of shared components and leverage from new utility development done by each application with a minimum of duplicate code.

Some of the strategies to address these objectives include:

- Definition of the shared component process that include:
  - Update process: Process for informing people impacted by the change and how to go about the impact assessment, and what steps should be followed to make the change.
  - Checkout/checkin procedures: How to checkout and checkin components, what tags should be used, and how the tags will be managed.
  - Synchronization process: Process for the creation of multiple versions of a shared component and how these multiple versions will be merged back to one version.
  - Release account management: How the latest version of shared components are distributed to various products.
  - Testing strategy: What type of testing should be done and what should be considered when testing a modification to a shared component.

- Regular update to discuss future additions and changes to shared components to assess their impact to other projects in the lab.
- Regular communication of changes to shared components during critical periods, especially during different application releases to determine priorities and schedule for incorporation.
- Regular communication of distribution processes to ensure that each application is getting the version desired.
- Regular communication of new components to ensure that there is a minimum of duplication or overlap.

Benefits of the Reuse Program

- Quality

  Reused code has much better quality than unique code. We have found that reused code is much more thoroughly tested since it is used in so many situations. Although it is often difficult to make a good impact assessment of changes, problems in code changes are revealed earlier. The large path flow coverage usually reveals problems before the release of the product, due to the heavy usage of our reused code. For this reason, we place a high priority on stabilizing the utilities early in the implementation phase.

  Relying on many shared utility procedures eliminates the risk of errors caused by oversight and accident when replicating source code changes throughout individual modules.

- Consistency and Standardization

  Perhaps the best benefit results from the consistency and standardization that reuse code enforces. We can now count on, for example, the lineprinter to be opened the same way in the transactions and programs that use the utility. Before, that same lineprinter routine existed in dozens of places, each individually perhaps performing slightly, or in some cases, drastically differently from the other.

  Eliminating code duplication not only reduces overall code length, but also reduces source code clutter in individual modules, thereby helping to preserve transaction clarity.

- Productivity

  Code maintenance and product enhancement are made simpler since a correction or enhancement can often be made in a reuse module and effected immediately throughout the entire application by replacement of the reuse module alone.

  Some reuse code can be used by similar software systems, saving time and effort when developing and testing other products.

Conclusions

- Initial Investment

  There is some initial investment that must be done with any reuse program. At MPO we started with a new product and then expanded it to other existing products. Both the initial startup and the expansion does require some effort in the development of the reuse code to begin with and the enhancement and maintenance of the reuse code as new products use the code.

- Coordination Effort

  As more and more products use the reuse code, the coordination effort between the products must be carefully managed. Impact assessment must be accurately done to ensure that we do not introduce defects to other products and rework is kept to a minimum.

- Long term versus short term productivity

  A reuse program requires both an initial investment and an ongoing investment as new products and enhancement use the reuse code. In addition, the coordination effort must be carefully managed. In the beginning the investment may be high, but it is paid back in the long term by better quality, more code consistency, and better supportability.

The following summarizes the key benefits and the prerequisites to success:

Benefits	Prerequisites to Success
High quality	High initial investment
Consistency and standardization	Detailed planning
Productivity	High coordination between engineering teams
Reduced support costs	Source code management system
	Testing strategy
	Documentation of reuse routines
	Defined maintenance process
	Measurement system that supports reuse

Acknowledgments

Thanks to Nancy Federman and Raj Bhargava for their support of the initial reuse program at MPO. Thanks also to Neal Clapper, Barb Williams, and Mike Levy for their continued support on this paper.

References

[Bhar 86] Bhargava, Raj K., Lombardi, Teri L., Nishimoto, Alvina Y., and Passell, Robert A., " New Methods for Software Development: System for Just-in-Time Manufacturing." *HP Journal* 37 (June 1986): 11-18.