Design reuse framework: a perspective for lean development

Naveen Gautam*
DaimlerChrysler Corporation
800 Chrysler Drive
Auburn Hills, MI 48326, USA
E-mail: ng33@daimlerchrysler.com
*Corresponding author

Ratna Babu Chinnam and Nanua Singh
Department of Industrial and Manufacturing Engineering
Wayne State University
Detroit, MI 48202, USA
E-mail: r_chinnam@wayne.edu
E-mail: nsingh@mie.eng.wayne.edu

Abstract: While the notion of reuse is around for a long time, its real potential is not exposed and exploited in product development. When reuse is applied through a knowledge centric process at low granularity level, it can help in reducing the development process wastes. In this article, a systematic and comprehensive reuse process framework is proposed by recycling the design knowledge and development artefacts in a seamless development process. In the proposed process framework, knowledge assets are created by decomposition, generalisation and parameterisation. These decomposed and atomised assets are used as reusable building blocks for the next level of product design. Requirements are modelled and testing is performed in virtual environment by auto generating the test cases and test vectors. Model-based development is demonstrated as an enabler of effective reuse. In this research, it is emphasised that model-based seamless development is instrumental for removing process waste and making the product development process lean.

Keywords: design reuse; knowledge management; design reuse process; lean product development; product development; model-based development.


Biographical notes: Dr. Naveen Gautam is Lead at DaimlerChrysler Corporation (Chrysler Group) for implementing model-based development for body and power electronics systems. He received BE degree in Mechanical Engineering from Rajasthan University (India) in 1986 and MTech degree in Mechanical Engineering from Indian Institute of Technology Kanpur (India) in 1993. He completed his PhD degree in Industrial Engineering from Wayne State University (USA) in 2005. He has also worked at Mercedes Development System and DaimlerChrysler Research in Germany. He is the author of several technical publications in the areas of non-traditional machining, system dynamics and design reuse based lean product development.

Copyright © 2007 Inderscience Enterprises Ltd.
N. Gautam, R.B. Chinnam and N. Singh

Dr. Ratna Babu Chinnam is an Associate Professor in the Department of Industrial and Manufacturing Engineering at Wayne State University (USA). He has authored over 50 technical publications in the areas of knowledge-based engineering, intelligent engineering systems, and advanced manufacturing engineering. Most of his research is funded by the National Science Foundation and other state/federal agencies. He carried out collaborative research with Ford Motor Company and DaimlerChrysler and consulted for such companies as Energy Conversion Devices and Tecton. He is a member of Alpha Pi Mu, INFORMS, NAMRI, and INNS.

Dr. Nanua Singh is a Professor in the Department of Industrial and Manufacturing Engineering at Wayne State University (USA) since January 1993. He was a Professor and Head of the Department at the University of Windsor, Canada and taught there for six years until December 1992. Before joining the University of Windsor, he taught at IIT Delhi (India) for seven years. He has written three books, published over 80 papers in various journals including IIE Transactions and over 50 conference proceeding papers. Currently, he is working in the areas of robust engineering, concurrent engineering and smart product modelling in a knowledge-based engineering environment.

1 Introduction

Shrinking product development times, increasing international competition, rapidly changing technologies and customers demanding high variety options are some of the forces that drive new development processes (Wheelwright and Clark, 1992; Pine, 1999; Ulrich, 1995; Helman et al., 2003). The complexity of offering greater product variety is being attempted by firms in many industries by implementing a variety of reuse strategies. Reusability of design is applied commonly in many industries, however, to different degrees. A close look into any product, in particular a mature product, will reveal that several components are either common with other products of the same manufacturer (commonalities across product families) or reused from the last generation of the product. While the potential for design reuse has been extensively discussed in the literature (Richards, 2000; Shahin, 1999; Duffy and Ferns, 1999; Mangun, 2000; Hillier, 2000; Carlson, 1993; Fisher et al., 1999) and it is widely practised in industry (Dillon, 1998; Culley, 1999; Catthoor, 2000; Roy, 2003; Fox, 2003), the scope for reuse was very much limited to the ‘physical’ world (i.e., reuse of components, sub-systems, and associated designs). However, there is enormous potential for ‘low level’ reuse, something not well exposed by academia and not exploited in industry. Reuse is not only the process of using physical components or associated designs but also the process of implementing or updating product information using existing ‘assets’. Assets can be specifications, designs, user documentation, test plans, test results, etc. However, we observed that most of the studies and reports in the literature mainly focus on reuse of the physical components or associated designs. In this study, we expose other forms of reuse that can significantly improve product development process efficiency and eliminate process waste. We also demonstrate how these concepts can be implemented with real world examples.
1.1 Levels of reuse

Reuse occurs in many ‘layers’ and each layer brings significant benefits based on its level of granularity. As shown in Figure 1, with the reuse layers pyramid, only the tip of the iceberg is exposed and considered by and large for reuse, whereas immense potential of reuse is yet to be exploited at a low level of granularity beyond the physical world in the form of requirements, behavioural models, and testing. In order to give a complete picture of reuse, we have described different forms of reuse. In the first section, high level of reuse (conventional and more prevalent) is described, and in the subsequent sections, low level of reuse is described. However, we see an immense potential remains unexploited in low level(s) of reuse. Our main contribution through this research is exposing low level of reuse and proposing ways to exploit it.

Figure 1  Reusability pyramid

1.2 High level(s) of reuse: physical layers

1.2.1 Platform-based reuse

Ulrich and Eppinger (2004) defined platform as “the collection of assets, including component designs shared by similar but distinctive products is called a product platform”. Planning the product platform involves managing a basic trade-off between distinctiveness and commonality. On the one hand, there are market benefits from offering several distinctive versions of a product. On the other hand, there are design and manufacturing benefits to maximise the extent to which these different products share common components.
By using a platform approach, a company can develop a set of differentiated products (Wheelwright and Clark, 1992). In general terms, the potential benefits of the platform approach are: reduced development and manufacturing costs, reduced development time, reduced systemic complexity, better learning across projects, and improved ability to update products.

1.2.2 Architecture-based reuse

Product architecture is the arrangement of the functional elements of a product into physical blocks. The purpose of product architecture is to define the basic physical building blocks of the product in terms of both what they do and what their interfaces are with the rest of the devices. Ulrich (1995) defines product architecture as “the scheme by which the function of a product is allocated to physical components”. Broadly, product architecture can be classified as completely modular and completely integral. The architecture specifies which modules will be part of the system and what their functions will be. Interfaces describe in detail how the modules will interact, including how they fit together and communicate. Standards test a module’s conformity to design rules and measure the module’s performance relative to other modules (Sosa et al., 2003).

Modularity in architecture

A module can be described as a large group of components that are physically coherent as a sub-assembly, which often has standardised interface designs. Modules may be common across different products, but they may also be specific to any one product. Modularity allows the designer to control the degree to which changes in processes or requirements affect the product, and by promoting interchangability, modularity gives designers more flexibility to meet these changing processes. This flexibility allows for delaying design decisions until more information is available without delaying the product development process. Another benefit is the ability of modularity to reduce life cycle costs by reducing the number of processes and reducing repetitive processes. Ulrich and Tung’s (1991) work details the costs and benefits of modular products.

1.2.3 Component reuse

Component reuse has been extensively discussed in the literature (Fisher et al., 1999). Component reuse is increasingly viewed by companies as a way to offer high variety in the market while retaining low variety in operations. For example, Dell Computers assembles several different end products using a relatively small set of core components. However, there is a trade-off on degree of sharing/reuse in terms of advantages and limitations. Since each component must be designed and tested, component sharing can reduce cost and improve quality. But there are several factors that can influence the cost of production and product development. Due to higher volume, economies of scale result in low production cost. Most often, a high level of performance standard (stringent requirements) of one component results in an over-engineered component for the milder or low level of performance standards. Common quality assurance, procurement, distribution, and spare part inventory reduce the overall system cost. Quality of the reused/shared component is often higher than the quality of uniquely designed
components. This may arise due to learning and quality associated with the increased production volume that may justify higher level of investment in product development and refinement.

Components reuse or sharing result in trade-off and negotiation of requirements and functional boundaries of more than one application. This type of compromise results in two major issues. The first one is fear of brand dilution, when the reusable component comes from the high-end product line. The second fear is the possibility of an oversized or over-designed component, which may perhaps reduce the variable and fixed cost due to volume and minimal investment in development and tooling but at the same time some indirect overheads like weight or electrical load may increase. This increased weight overburdens on the final product and has to be addressed appropriately. This translates back into high variable and fixed costs as some other components are now to be designed towards lower weight and stronger material criteria.

In order to pursue a sustained reuse path, we need to take into account certain considerations for reuse. Some of the enablers of reuse are: follow standard naming convention, use level of abstractions, design sub-systems before detailed design, and design interface before detailed design.

1.3 Low level(s) of reuse: knowledge resource layers

This is the underwater part of the iceberg which has not yet received due attention and remained unexploited even though it has tremendous potential for product development efficiencies improvement.

1.3.1 Requirements reuse

Requirements reuse is commonly defined as the process of analysing, eliciting and managing requirements at a suitable abstraction level so that they can be reused in new systems (Shehata et al., 2002). Several benefits have been discussed in literature on requirements reuse (Nuseibeh and Easterbrook, 2000; Diaz-Redondo et al., 2003; Laguna et al., 2004; Gotzhein et al., 1998), however much less work has been done in the form of solid methodology and step-by-step implementation guidelines with any detailed case study to support the proposed methods. In the following sections, we have provided a methodology along with an example to prove the concepts. Moreover, most of the research in the area of reuse neglects requirements engineering, except for the kind of commonality and variability analysis which goes on to develop a framework after the requirements for the product line have been established (Shehata et al., 2002).

1.3.2 Model reuse

Behaviour and implementation models are next step in the development process to capture user requirements and implementation details. The reuse of these models offers great value if modular models are designed based on singularised requirements and parameterised to absorb product specific variability. In the next sections, we will describe how model can be used as an enabler for reuse-based lean development as well as an oracle for testing.
1.3.3 Testing (cases, vectors and results) reuse

“Even if thousands of functions are carried out flawlessly, just one mistake can be catastrophic to a mission” (28 March 2000, NASA press release). In order to carry out thorough testing for increasingly complex systems, we need almost infinite resources to test each system’s iterations during development. However, in any project we have neither infinite resources nor time to carry out that large number of tests. Therefore, we need to identify alternate ways of dealing with this by testing automation and testing reuse.

In our study, reuse is considered primarily in three dimensions. One is along the process chain as we go deep into design; modelled functional requirements can be reused to auto-generate the test cases. In this way much effort made in modelling can be translated as payback for generating extensive test scenarios, test cases, and test vectors. The second dimension of reuse is along the time, between different versions to revalidate the next versions with playback of pre-recorded test cases and comparison of results against expected test results. The third dimension of reuse is in the form of test cases, test vectors and test results, across different vehicle lines. Based on the level of common functionality among vehicle lines, a large number of test cases, test scripts and accepted output can be reused.

In the following sections, we will uncover systematic reuse of knowledge assets such as requirements, models, test cases, test vectors and test results for lean product creation, which are essential and valuable artefacts of the product development process.

2 Model-driven development process: core enabler of lean reuse

The model-driven approach to product development (Figure 2) is actually a perfect complement to realise a requirements-driven development process and provides a solid foundation for lower level reuse. This process has been discussed in the literature (Thiry et al., 2004) and several advantages of this approach have been identified. El-Far (2001) studied the wealth of published work portraying case studies in both academic and industrial settings as a sign of growing interest in model-based testing (Gronau et al., 2000; Petrenko, 2001; Robinson, 2003; Froehlich and Link, 2000). However, no study shows it as a major tool for implementing systematic reuse in lean product development. The idea is simple: a picture is worth a thousand words. Consequently, model-driven development is based on a visual language instead of human language or programming language. The target profile of a system as described in the agreed set of requirements is transformed into a visual language that allows description of behaviour, how to use a feature, how a component interacts with its environment and whatever else needs to be considered. Every requirement is visualised in one or more model elements, and all model elements together are the model of the target system.

Having such a model offered the following advantages:

- It has been shown to stakeholders to explain how the system will appear and to confirm their acceptance, which reduced the chances of misinterpretation and eliminates rework.
- It removed ambiguities often associated with natural, spoken languages.
Design reuse framework: a perspective for lean development

- It made easier for system developers to understand what they are expected to deliver.
- It was the perfect documentation of the source code produced to realise the model.
- It provided an oracle for generating test-cases and test vectors.

Even more, extremely compelling advantages were added when the visual modelling language is a formal notation, such as Unified Modelling Language (UML), Simulink/Stateflow® or Statemate®.

- The model of the system has been executed: stakeholders were able to see the dynamic behaviour and decide whether the requirements are valid.
- By executing the model, the fulfilment of the requirements were verified.
- The source code to implement the system was produced automatically from the model.
- Test cases and test scripts were auto-generated for final hardware and software testing.

As shown in Figure 2, model-based software development provides a seamless development environment. All of the ‘brain work’ was done at the model level. The entire specification and implementation work was done by using the visual notation of the formal modelling language. By executing the model, the system was simulated and any issues or errors were detected very early in the development process. In case of any modifications, enhancements, or error corrections are needed, these are also performed in the behavioural model. A behavioural model is logical representation of system behaviour created based on user requirements. This model can be simulated or executed against the system requirements to verify the requirements. Once the user requirements are correctly captured, the faulted behaviour and other implementation details not exposed to the user but necessarily required to implement the system (such as low power mode operation, architecture specific input/output mapping, and interface definition) are carried out in implementation model. All the vehicle line specific variations were handled at this level. Therefore, feature behaviour models are completely reusable across all vehicle lines. Once the implementation model is developed and validated, the source code for a target executable is produced automatically at the push of a button. The operating system and micro-defined work, such as low-level hardware drivers, are reduced to a minimum and are easily combined with the model-generated code. Other standard pieces of the puzzle such as communication driver (CAN, LIN, J1850), network management, communication database, etc. are being incorporated as add-ins and these are completely reusable across multiple Electronic Control Units (ECUs) in various car lines. This approach also covered ongoing development and enhancements of the product, which is an essential prerequisite for a suitable life cycle solution. All development artefacts resulting from the process are being reused for future projects. Obviously, including an existing sub-system represented and well-documented by a set of requirements and corresponding model elements is a much easier task than taking a piece of source code with an outdated, manually generated documentation and trying to guess what this source code is actually intended to deliver.
With advanced technology offered by the Automatic Test Generator, a whole range of test vectors can be auto-generated from the graphical specification that can be included in Test Plans. The generated test vectors can be run on the embedded system with a hardware in-the-loop tester to ensure that build meets the specification.

It is evident from the given description that model-based software development is a seamless development process and facilitates a significant amount of systematic reuse as an inherent part of the process. However, we took the next step and proposed a framework to establish it as a knowledge-based lean process.

Competitive research (Morgan, 2002) identifies the generic process waste in the product development. However, in this study, we would like to address how model-based development process is instrumental in eliminating these process wastes:

- **Hands off**
  Whenever responsibilities for the product are handed from one group/person to another group/person, there is hand-off. Each time there is a hand-off, significant information and time is lost. High numbers of hand-off also make it difficult to built accountability into the system. Since in the model-based development, code is auto-generated based on the model there is no or very little hand-off. Similarly, test vectors are also auto-generated using Auto Test Generator; thus, there is no hand-off for test cases generation. Therefore, the natural and smooth handover of upstream value aid to the downstream activities makes the process lean.

- **External quality reinforcement**
  Whenever a separate group is responsible for quality, it takes ownership away from the primary group/person and adds significant waiting time for inspection events. During model-based development, quality is checked by the modelling group while simulating the models and executables. Once the auto-generated code is delivered to the other group (who is ultimately responsible for integrating all together), they
check it via compilation, input/output levels of binding tests and skeleton functionality checks. All syntax, mapping and binding errors are caught during integration. Since this group is also responsible for integration and system validation testing, all the upstream modelling group work is thoroughly tested by the consumer and not by any third party or separate group.

- **Waiting**
  
The waste of waiting is extensive in product development: waiting for data, a decision of resource, or for other parts to be completed. This is a major source of queues and subsequent long lead times. As we established that the process is seamless and there is continuous flow of information and artefacts within the tool chain, therefore, there is no scope for waiting time unless requirements are not clear and decisions on requirements are delayed.

- **Transaction waste**
  
  This waste occurs when non-value added steps are required to accomplish the primary task. Since process is streamlined and made seamless, there is no non-value added task in the process. In the traditional manual code-based development process, requirements are handed over to write the test cases for ECU verification and validation. In model-based process, all the test cases and test vectors are auto-generated using models; the non-value added step can be eliminated.

- **Re-invention waste**
  
  This occurs when the same problem is solved repeatedly or when existing solutions/designs are not utilised due to lack of learning. This is one of the most dominant advantages of the model-based process. Since, all the feature models are common for all the vehicle lines and constantly updated, any new feature required by a lead implementation is automatically made available for reuse in next vehicle line. Moreover, any bug/issue identified at any stage on one vehicle line at any stage of development is fixed for all. Therefore, quality of the reusable models is bound to improve over a period of time and all the knowledge captured during design and development of product stays in the models.

- **Lack of system discipline**
  
  This is also extensive in some product development systems. The lack of scheduling discipline for instance causes arrival variation, which is responsible for long queues. Since every feature is maintained as we move on, there is only little left to tune it for specific implementation. Therefore, model-based process is robust enough to withstand against scheduling discipline.

- **High process and arrival variations**
  
  The product development process by nature is variable. Task and arrival variation are two primary contributors to long queues and lead times. Since flow of information streamlined with tool and inherent process, arrival variations are minimal. Similarly, process is controlled via a set of predefined steps; it eliminates process variations and quality level of deliverables stay consistent.
• **Long batch sizes**
  According to Factory Physics, cycle time increases with batch sizes. Batch data release is the primary culprit of waste in product development. Because of the ongoing development process and ‘fix as you find’ approach, long batch sizes can be avoided in model-based development.

• **Redundant tasks**
  There are tasks repeated across functions. Multiple inspection points and isolated development are examples of this category. Since all feature models are put together in an implementation model, development is well coordinated and all changes are self-synchronised before application layer being released to other group or supplier for final integration.

• **Stop and go task**
  Each time an engineer needs to reorient for a new project or restart the same project, it is like a set-up. Multiple set-ups are a prime source of waste and poor efficiencies.

• **Unsynchronised concurrent task**
  It is one of the most insidious wastes. It seems like the right thing to do, but unsynchronised concurrency is root a cause of waste. Whenever concurrent development is not communicated or coordinated in a systematic way, it can become counter-productive in the development process. However, with model-based development, the process is synchronised and all development artefacts flow smoothly downstream and leave no scope for non-synchronicity.

• **Inefficient communication**
  Product development by nature is solely based on communication. However, it is often informal and mostly inefficient. This leads to rework, over budget, and run over schedule. Since model-based development is designed to minimise the communication gap by graphical representation of behaviour, it eliminates the scope of communication inefficiencies.

3 **A process framework for knowledge-based artefacts reuse**

In this section, we propose a process framework through which knowledge-based development artefacts such as requirements, models and tests can be reused. In order to carry out thorough testing for increasingly complex systems, we need almost infinite resources to test each system’s iterations during development. However, in any project, we have neither infinite resources nor time to carry out that large number of tests. Therefore, we need to identify alternate ways of dealing with this by testing automation and testing reuse. Testing of application software and telecommunication prototypes based on auto-generated test cases is explored in industry (Addy, 1998; Bagnasco et al., 2002). However, systematic reuse of test cases, test vectors and test results is not reported specifically. Through our review of the current research on reuse at the requirements level (Nuseibeh and Easterbrook, 2000; Diaz-Redondo et al., 2003; Laguna et al., 2004; Gotzhein et al., 1998; Lam, 1997; Mannion et al., 1999; Lewis et al., 2000), we were able
to find a conceptual process framework that was more or less common to all approaches, but these were at high level of abstraction and details were left out for the interpretation of the users.

Part of this framework is founded on the previous work carried out by Shehata and Eberlein (2002; 2004), Lam (1997), Mannion et al. (1999), and Lewis et al. (2000). The development of this framework helped us in gaining a better understanding of how requirements and other knowledge assets can be reused and how requirements interactions play an important role in reuse. For example, a requirement can be very specific to a certain system in a certain domain which thus gives it a very low level of abstraction and a low reuse potential while another requirement can be very general, with a very high level of abstraction and thus a higher reuse potential. We identify the following abstraction: domain specific requirements, generic requirements and requirements implementation rules. Domain-specific requirements are requirements that are derived from certain domain applications and are concerned only with this domain. Therefore, they cannot be reused in any other domains and applications. Generic requirements are requirements that are with some variations common to different applications. One can replace the differentiation part in the requirement with a variable making requirement generic such as programmable parameters for timings, enable/disable across various car lines in different features (e.g., the system shall provide X wipes after a wash cycle).

Generic requirements can be reused in specific domain, using a domain requirements framework. A domain requirements framework is a framework that provides guidance on how to generate a requirements specification with hook-ups to facilitate development.

The process for knowledge-asset based reuse is composed of three phases. The first phase starts with the analysis of a group of systems that are part of a product line in a certain domain that have numerous commonalities between them (i.e., wiper washer system across several car lines, different implementations such as run to park or non-run to park). In the second phase, we customise the reusable requirements, models, and tests and also add some new requirements that are needed to meet the specific needs of the new project. The third phase to this process is requirement interaction management.

3.1 Phase 1: Decomposition, elicitation and generalisation

The aim of phase one is to create a database of reusable knowledge-assets that can be used later as a new system.

- **Domain analysis**
  Phase one starts with domain analysis during which a team of developers starts to analyse already developed systems that have numerous commonalities among them. The output of this exercise is a set of reusable requirements (core requirements and/or parameterised requirements and/or optional requirements) along with a set of rules of usage that the developer has to follow these requirements (e.g., requirements A and B are mutually exclusive).

- **Establish discriminants**
  A discriminant is an entity requirement which differentiates the two systems (two different implementations) (Lam, 1997). A discriminant can take three forms:
Mutually exclusive: when requirements A and B are mutually exclusive, B shall not be selected if A is already selected during reuse (e.g., wiper system type: run to park wiper system or non-run to park wiper system).

List of alternatives: A or B or both A and B can exist, but at least one of these shall exist (e.g., activation of wiper via manual switch and/or rain sensor based).

Optional: feature A exists or does not (e.g., vehicle wiper system is equipped with rain sensor or not).

• Generalisation

The next step in this phase is to generalise the requirements. Generalisation can be accomplished by extracting the implementation/domain specific values from a requirement and replace these with variables and store the parametric values for these variables independently as implementation requirement. When the function/package is optional in the final application/product, it can be turned on or off by enabling or disabling the parameter values. These generic requirements are generic enough to reuse for the next application without any change. However, for each application, parameters need to be tuned for domain specific application. Compare the requirements form each existing system and the reusable requirements; if both requirements have the same constraints, then the constraint itself can be reused. In other cases, it may be justified to relax the constraint in order to make the requirement reusable. For example, during manual or rain sensitive wipe mode, the same communication loss constraint can be reused. If communication is lost between wiper control module and wiper switch or rain sensor, same default action can be taken. Switching the wipers to low speed if they were not in off before communication was lost, can be considered a fail-safe design. However, in the next ignition cycle, wipers can be turned off. This way, the customer can drive the car safely back home, but when the customer starts the car next time, the wiper will not start due to a defective switch or communication fault and has to take the car for repair.

• Atomisation

When requirements are not clear and concise, it becomes hard to manage, change, and test. Therefore, it is proposed to singularise (atomise) the requirements at this stage. Each functional requirement, which can be experienced by the end user, shall be singularised to make testable. In Figure 4, each singularised requirement is represented by an individual line item. A relationship among these requirements can be established using links provided in Dynamic Object-Oriented Requirements System (DOORS® tool).¹

• Grouping (package building)

The next step is to gather similar requirements into a group and place them into one package (e.g., all reusable requirements related to interval wipe are gathered into a package called Interval Wipe). Finally, these packages are stored in the database ready to be reused. These packages are reusable building blocks for future products. During this study the DOORS® tool was used to manage the requirements, which supports linking the singularised requirements with a package.
Figure 3 Artefacts reuse framework: Phase one – design for reuse

Figure 4 Requirements atomisation and package formation

3.2 Phase 2: Reuse

The aim of this phase is to develop a complete set of requirements that describes the new system reusing the requirements stored in the requirement database.
• **Gap analysis**

This phase starts with the identification of the gap between what we have in our reusable requirements repository versus what we want in the new system.

• **Customisation**

This phase proceeds by customising the reusable requirements and adding some new requirements necessary to meet the specific needs of this new project. Some requirements may contain numeric values describing coefficients, array lengths, units of time, or simply constants. Often these requirements can be reused if their numeric values are changeable. Therefore, all the parameterised variables used in generic requirements need to be defined and replaced with suitable numeric values applicable for the current application. A separate table shall be maintained to map these parameter values to the variables for a particular application (implementation). An example of parameter mapping is shown in Table 1 for wiper washer sub-system implementation.

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Parameter for wiper washer sub-system</th>
<th>Implementation 1</th>
<th>Implementation 2</th>
<th>Implementation 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of wipes after wash</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Headlamp wash time</td>
<td>2 seconds</td>
<td>4 seconds</td>
<td>2 seconds</td>
</tr>
<tr>
<td>3</td>
<td>Wash time-out</td>
<td>20 seconds</td>
<td>20 seconds</td>
<td>10 seconds</td>
</tr>
<tr>
<td>4</td>
<td>Number of wash cycles for one headlamp wash</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Headlamp on with wiper</td>
<td>Enable</td>
<td>Disable</td>
<td>Enable</td>
</tr>
<tr>
<td>6</td>
<td>Headlamp on with wiper hysteresis on/off time</td>
<td>5 wipes in 180 seconds</td>
<td>10 wipes in 180 seconds</td>
<td>5 wipes in 180 seconds</td>
</tr>
<tr>
<td>7</td>
<td>Number of sensitivity levels</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>Interval wipe delay time for different stages</td>
<td>36, 18, 9, 5, 3, 1</td>
<td>18, 5, 1</td>
<td>36, 18, 9, 5, 3, 1</td>
</tr>
<tr>
<td>9</td>
<td>Speed sensitive wipe limit threshold</td>
<td>20 mph</td>
<td>20 mph</td>
<td>10 mph</td>
</tr>
<tr>
<td>10</td>
<td>Type of implementation</td>
<td>Run to park</td>
<td>Non-run to park</td>
<td>Run to park</td>
</tr>
<tr>
<td>11</td>
<td>Run to park time-out</td>
<td>3 wipe cycle time</td>
<td>10 seconds</td>
<td>2 wipe cycle time</td>
</tr>
</tbody>
</table>

• **Acquiring requirements rationale**

We believe that requirements rationale is a critical yet overlooked aspect of reuse strategy, which deserves greater attention than currently received. Often, we found the rationale for some of the requirements is absent or difficult to comprehend, which impede the domain analysis and gap analysis. For example, why is there a limit on time delay imposed on the wiper parking retrying strategy before the wiper motor is switched off? In such cases, it is vital to elicit the rationale from expert ‘inside’ knowledge in order to be able to reason the requirements in the domain. If explicit rationale is not captured for selecting special cases or specific values for the
variables, it is likely that in subsequent reuse the variable value will be used as is without any consideration for justification. This type of reuse is dangerous and may result in big issues and recalls.

Figure 5  Artefacts reuse framework: Phase two – design by reuse

3.3 Phase 3: Interaction management

It is our conviction that knowledge assets reuse must be done in conjunction with careful management of the relationships and dependencies that arise between existing knowledge assets and new requirement. Requirements Interaction Management (RIM) was first introduced by Robinson et al. (2003) as “the set of activities directed towards management, and disposition of critical relationships among a set of requirements”.

This phase is a very important task to ensure that the reused and the new requirements will not have any negative influence on each other. The outcome of this phase is a set of dependencies that have been analysed for the detection of any interactions between them.

- Identify the dependencies

  This is the first step in the phase. Even though this step shall be finished during domain analysis, it has been observed that due to lack of resources and pressure for speedy development, it is often left incomplete during the domain analysis part of the first phase and the customisation part of the second phase. There exist several possible forms for the dependencies. However, the most common form is the parent-child relationship and we are focused on this relationship in our study.

  Example: Headlamps On With Wiper (HOWW).

  Requirement 1: Headlamp shall be turned on with wipers on if HOWW is enabled.

  Requirement 2: Headlamp shall be turned on if wiper stays on for X amount of time.
In this example, Requirement 1 is the parent and Requirement 2 is the child. If the HOWW option is not enabled, then there is no need to have the child requirement and corresponding test-case for the delay parameter X for turning headlamps on based on activation of wipers.

- **Linking the requirements**

Once the relationship between requirements is identified, the next step is to establish links between these. Linking shall also facilitate navigation between the linked requirements. The outbound link represents parent status, whereas inbound link represents child status in relationship. We used a tool (DOORS®) which supports this form of dependency representation using inbound and outbound links. These links can also be used for navigation between the linked requirements. Whenever any parent requirement is selected for reuse, it is not necessary to select the child requirement; however, if a child requirement is selected, the parent requirement must be selected. All the linked (parent or child) requirements shall be reviewed for customisation during reuse to make sure that integrity of the system requirement is not broken. A complete artefacts reusable framework encompassing all three phases of reuse is depicted in Figure 6.

**Figure 6** Development artefacts reuse framework
4 Application of proposed framework

4.1 Use of requirements for test case generation

When textual requirements are singularised and written concisely with pre- and post-conditions, they are great sources for generating well-defined test cases. Requirements links are also very helpful in establishing dependency for test cases. With the singularised requirement structure, at least one test case is generated per requirement. Requirements and test cases are maintained in a tool repository. Numerous requirement management tools are available which facilitate linking the requirements and capturing their interactions. These links are very helpful in tracking changes. If a requirement is changed, all linked requirements are reviewed for the changes. Moreover, all corresponding test cases need to be reviewed, as they are likely to change. The relationship of requirements and test cases is well established. However, when requirements are maintained in textual form, every time requirements change, test cases need to be modified manually.

4.2 Use of behavioural models for auto-generating test scripts

Model-based software testing has existed since the mid-1970s (Robinson, 2003), and modelling advocates have written many white papers, which show its dramatic improvement over handcrafted test automation. Yet, only a small segment of the mainstream software industry currently uses model-based testing. Model-based testing has historically been used in industries such as telecommunications and avionics, which have a stringent software quality bar. Savage et al. (1997) claimed that case studies in those industries have shown a ten-fold productivity improvement in reaching that quality level. At this level of test generation productivity improvement, one test engineer using a model-based testing tool can be as productive as ten test engineers using manual test generation (Rosaria and Robinson, 2000).

When requirements are captured in model form, these are much more valuable in the process chain and give high payoffs during testing. Simulation of models provides incredible ways to find bugs and various unwanted behaviour such as non-determinism, instability, write/write racing with different values, etc. during the early phase of development. A common paradigm of test automation is ‘capture/playback’ test scripts. A test script defines the set of actions to be carried out in order to conduct a test and the expected outcomes that are used to identify any deviance in the actual behaviour of the application from the logical behaviour in the script (errors during the course of that test).

The capture/playback method is used because it is easier than ‘coding’ scripts. These scripts are then used to test the newer version of model/application software. These auto-generated test scripts are recorded for all accept scenarios by manually providing a set of input per requirement using Graphic-User-Interface (GUI).

Once recording is completed for the requirements set, these recorded test cases in the form of scripts are played back in batch mode and output results are collected in a file for the current iteration. This output file is (text) compared against standard expected results.
Expected sets of outputs are generated based on the standard test results from a mature design. Figure 7 illustrates playback scripts based on the validation process for the newer model. Deviation from the expected output is an indication of error. These playback scripts can also be used for generation of a set of inputs which can be applied to hardware for the corresponding functional testing.

**Figure 7**Playback test scripts-based validation

There are some inherent problems with this paradigm. First, test automation is only applied at the final stage of testing when it is most expensive to go back and correct the problem. The testers do not get a chance to create scripts until the product is finished and turned over. At this point there is a tremendous pull on resources to just test the product and forgo the test automation effort. Using only capture/playback may be temporarily effective, but using capture/playback to create an entire suite will make the scripts hard to maintain as application modifications are made. Test scripts are constructed to test a new version of the system, but the only system that is available is the old buggy version. Again, the script must be validated against the design model, but it may also be possible to execute portions of the script against the old version of the system. The relatively improved approach is to capture the test vectors based on playback scripts and maintain the test vectors in conjunction with singularised requirements. Any change in requirements will flag the change on all related artefacts and force the update on the corresponding set of test vectors. Figure 8 shows the Statemate tool environment for test scripts playback. Results are captured in an output file, which is compared against expected results (Figure 9).
Figure 8  Playback scripts

![Playback scripts](image)

Figure 9  Test results and accepted output

![Test results and accepted output](image)
4.3 Results

This process framework was partially adopted and implemented in body electronic features development at Chrysler division of DaimlerChrysler Corporation, and the results are significantly encouraging in the form of reduced development time and quality improvement. Design artefacts are reused from one stage of development to the next stage and flow of the information is made seamless in integrated development environment. The knowledge artefacts used in the process are textual requirements, behaviour models, implementation model, test scripts, test vectors and expected output. The development and testing time for an important ECU (integrated power module), which controls significant amount of functionalities, have been reduced by 40% compared to previous model years. Likewise, the maturity level and the quality have also improved by 30%. As one specific example, the amount to time required to revalidate the entire exterior light sub-system with each software release (which used to take eight hours) has been reduced to 40 minutes with the help of pre-recorded test vectors and expected results. Due to significant time savings in revalidation, more iterations have been performed and quality of sub-system functionality have been improved.

5 Conclusions

This paper laid out a systematic reusability framework using different layers of reuse based on granularity. Here we identified the broad differences between the high-level reuse (physical layers) and low-level reuse (knowledge resources layers). High level layers are identified as platform, architecture and physical components where all attention is focused by industry and academia. However, we emphasised more on concepts of low level of reuse, where tremendous potential for reuse remain unexploited. High value of reuse comes from lower layers where knowledge resources in the form of design artefacts are managed at low granularity, such as requirements, behavioural models, implementation models, test cases, test scripts, test vectors.

We observed benefits of reuse from lower layers where design artefacts are managed at low granularity not yet exposed in academia and not well exploited in industry. If focus is shifted from the high level of reuse to low level of reuse, many more product variants can be created quickly with ever increasing quality built into the product. We proposed a framework for artefacts reuse, and then demonstrated how model-based development can be used as a backbone for lean development process. In this way, we reused extensively all the upstream design artefacts for recreating mature and high quality downstream artefacts. We demonstrated how model-based development provides systematic elimination of the process waste and make the development process seamless and lean.
References


Note


Acronyms

CAN: Control Area Network
DOORS™: Dynamic Object-Oriented Requirements System
ECU: Electronic Control Unit
EEPROM: Electrically Erasable Programmable Read-Only Memory
LIN: Local Interconnect Network