The Release Matrix for Component-Based Software Architectures

Louis J. Taborda

Macquarie Graduate School of Management

MGSM WP 2004-3
April 2004
Disclaimer
Working papers are produced as a means of disseminating work in progress to the scholarly community, in Australia and abroad. They are not to be considered as the end products of research, but as a step towards publication in scholarly outlets.

© Copyright: Louis J. Taborda

Research Office
Macquarie Graduate School of Management
Macquarie University
Sydney NSW 2109
Australia

Tel  612 9850 9016
Fax  612 9850 9942
Email  gsm-research@mq.edu.au
URL  http://www.gsm.mq.edu.au/research

Director of Research  Professor John A. Mathews
Manager, Research Office  Ms Kelly Callaghan

ISSN  1445-3029  Printed copy
       1445-3037  Online copy

MGSM WP 2004-3

The Release Matrix for Component-Based Software Architectures

Mr Louis J. Taborda
Macquarie Graduate School of Management
Macquarie University
Sydney NSW 2109, Australia

Email  louis@taborda.com
Abstract:

The challenge of managing the development and evolution of complex, component based software is increasingly being recognized as the development of such systems becomes more common. This paper attempts to understand the relevance of current management best practices by utilizing a simple meta-model to illustrate the impact that architectural complexity and reusable components have on management patterns. The model serves as a heuristic device and supports the view that products based on a framework of reusable software components pose new challenges and have to be managed simultaneously at a number of different levels. This provides a rationale for the Release Matrix, a generalization of a software release plan, previously proposed as a technique for managing software product lines. The Release Matrix has practical applications for tracking the evolution of complex component based systems and is shown via the model to be a natural consequence of increased architectural complexity and component reuse. This paper has relevance for developers seeking simple techniques to help them manage challenging component based programs, as well as researchers interested in the conceptual basis and limits of current management practices.
The Release Matrix for Component-Based Software Architectures

1 Introduction

Software intensive systems continue to grow in complexity and while component based architectures provide benefits in terms of productivity and quality; they also represent a growing management challenge. Beyond traditional software development expertise necessary to assure component quality, component based development (CBD) must address new challenges including component selection, reuse, assembly, as well as the integration testing and evolution of a configuration of inter-dependent components [1]. These challenges are markedly different from one-off systems development and new methods, and techniques are needed to tackle the management of such componentized systems [2, 3, 4].

A significant benefit offered by CBD is the potential to reuse components across a number of products (alternatively, applications or systems depending upon the terminology preferred). CBD methods, like their object-oriented predecessors, encourage the design of components for future reuse and means of easily identifying and utilizing these components in the construction of software systems is an area of active research [3, 5]. This body of work increasingly recognizes that the success of CBD and product line architectures (PLA) are intimately connected to the management of a configuration [4, 5, 6], and hence renewed interest in the discipline of Configuration Management (CM) applied at an architectural or coarse-grained, component or sub-systems level [7, 8] rather than at the software source level. The emphasis of this paper is on the management of configurations of such coarse-grained components that may subsequently be decomposed into the lower level entities that can be queried for more detailed management information.

In systems development programs, exemplified by early military and aerospace initiatives, a complex system is typically decomposed into sub-systems in a manner that has a close, but imperfect correspondence with CBD where decomposition is not the only means of identifying components. In that domain, the Systems Engineering discipline [9] ensures that the sub-systems
integrate into the desired system. For all the formalism that supports such systems development programs, managing CBD can be shown to be inherently more complex. This is due to the fact that, as well as the system integration and program coordination challenges of complex systems development, the reuse of components adds a new level of management complexity. One where there is not a single customer to be satisfied, and thus the demands of the separate products reusing the component has to be juggled.

Traditional project management (PM) techniques, such as Work Breakdown Structure (WBS), the Critical Path Method (CPM), the Program Evaluation and Review Technique (PERT) and Gantt charts, were not conceived to address the significant interdependencies that often make planning a product development in isolation irrelevant. The release of a product is therefore no longer a single, isolated event but rather one that involves careful choreography of a series of smaller, “fractal” component releases. Providing accurate and reliable program status in such environments is extremely difficult, often causing management to become intractable and reactive.

While many CBD practitioners routinely have to grapple with these issues in the workplace, there are few management techniques available to guide their activities. This research has been motivated by such real-world conditions where the effectiveness of current management best practices may appear inadequate. In exploring practical new techniques more suited to these conditions, the meta-model presented in this paper was found valuable for describing the conditions that gave rise to “classical” management best practices and illustrating why the adoption of CBD and systematic reuse necessitates new paradigms to address this new level of complexity.

The remainder of this paper is structured as follows. Section 2 presents an overview of the meta-model to study the effects of product complexity on management patterns. Section 3 investigates the relevance of the most complex pattern and its realization in modern CBD and product lines. Section 4 describes the Release Matrix and shows it to be the natural result of the interdependencies between products and components with practical applications. Section 5
concludes by reviewing the future direction of the research, as well as discussing current limitations and the validation of the model in the real world.

2 Modeling Product Complexity

In an effort to arrive at new insights into CBD, it is instructive to go back to fundamentals and review the management patterns that apply to the development of increasingly complex products. A simple meta-model acts as a heuristic device to investigate the effect of growing architectural complexity and to understand the appropriateness of different management techniques. The model itself was motivated by a desire to understand the challenges of complex software development and offers a rationale for the previously proposed Release Matrix [10].

![Diagram of product component relationships](image)

**Fig. 1.** Four stages of increasing development complexity

---

1 Entity-relationship notation is used to illustrate the model as standard data-modeling techniques can be applied to normalize the many-to-many relationship.
2.1. Monolithic Architecture

We start by considering the simplest architecture (Fig. 1a) where the one-to-one correspondence between a product and component represents the development of a monolithic software system. In such a simple scenario there is little need to distinguish between the terms product and component (better called the architecture) since they are one and the same.

This stage represents the earliest development paradigm when the focus of all effort is a tightly coupled architecture that has only to meet the needs of a single product. As such, it offers a textbook example of early development where PM techniques would ably support the planning and tracking of activities. A single development organization is likely to have been responsible for the release of the complete product, requiring little packaging given its monolithic nature, and using CM to provide the baseline control for the delivered product.

2.2. Product Decomposition

As systems grew in complexity, the natural response was to apply the “divide and conquer” maxim to decompose the product into sub-systems or components. This results in the one-to-many relationship between products and components (Fig. 1b). Irrespective of how the decomposition of the system is achieved (whether by top-down structured methods or object-oriented techniques such as domain engineering) they have the effect of “breaking-down” the monolithic architectures of the past and enabling different components to be developed separately.

Under these conditions, the organization of the work would usually follow the architectural boundaries (which might also represent different skill-sets) so that the individual components could be developed and managed as separate (sub) projects.

While these are useful devices to manage complexity, the trade-off is that the separate component development schedules now have to be coordinated so that they can be integrated into the required product. As customer requirements would in general span or be allocated to the
different components, achieving a product release necessitated the coordinate of the separate component releases. This requires significant management coordination that was often possible because the component teams were dedicated (if not contractually bound) to the product’s release schedule.

In particular, aligning the separate component release schedules is essential to integrate, test and deliver the final product, but achieving this and tracking the progress of the program can be problematic. Thus, it is common for the program to be under the control of a system integrator (or prime contractor in military programs) responsible for the product or system integration, while the components that are not commercial off the shelf (COTS) are developed autonomously by teams that can be in-house, remote, sub-contracted or out-sourced.

System Engineering techniques help address these challenges by ensuring that the separate components are specified and designed to be consistent with the overall system concept and that the customer requirements are traceable across these components. CM must provide the baseline controls to ensure that the different component releases are clearly identified and made available for integration and testing at the product level.

This pattern is manifest in a number of different real-world contexts including military development, CBD, PLAs and all manner of information systems (IS) development. While the nature of the components and their granularity may vary, the management pattern remains consistent and relevant.

2.3. Reusable Components

The potential to reuse or separately market software capability is at the heart of CBD principles. The aim is to identify and isolate such capabilities so they can be managed as separate components that may be reused by multiple products or applications (Fig. 1c). Increased complexity only serves to drive specialization, making it more likely that critical components will mature into reusable components or sub-systems. Indeed, domain engineering attempts to
identify such components early so that a shared framework of reusable components and services exists for products.

There is a spectrum of possible reuse, and COTS products could be seen as the extreme case that best represents this pattern. Here potentially numerous customers have to be supported and, depending upon the maturity of the component/product, this could entail maintenance patches and potential development activity. The level of “shrink-wrap” that the product has achieved will determine the degree of direct influence a customer will have on the component’s evolution or direction. COTS vendors would always hope to be responsive to market needs but reusable components developed in-house as part of (say) a domain engineering effort, can expect a different level of feedback and demand for responsiveness from its customers. The close interaction and greater flexibility afforded by in-house reuse efforts represents the other extreme of the reuse spectrum, often characterized by iterative development and short release cycles.

Irrespective of the environment, a decision by a product group to reuse a common component represents a significant change in the balance of power within a program. The fact that there are multiple customers for a reusable component means that it is elevated beyond the control of any single product group and now must act as a separately managed entity responsible for its own destiny. Thus, the reused component (and its development group) must introduce mechanisms for the collection, prioritization and conflict resolution across its different customers. The release planning and scheduling can become fractious as they directly affect the product plans and capabilities. No single product can dictate the reused component’s evolution or release plan as they have to satisfy the demands of several product markets concurrently, or its reusability (or market) is threatened. The products that use a common component must therefore share influence and must negotiate with the component team to achieve an acceptable component release plan.

A product’s loss of control has to be traded-off against the benefits of utilizing a reusable component that can include shared development and maintenance costs, improvements in quality, the shrinking of schedules and the reduction of risk. Each product group must carefully weigh these considerations before making the decision of whether to utilize a reusable
component. In the case of a COTS component, this corresponds to the classic build vs. buy decision; whereas for in-house components this relates to the decision to identify reusable components instead of recreating software capabilities.

2.4. Concurrent and Systematic Reuse

The ultimate challenge in our study of increasing product complexity is the many-to-many relationship between products and components (Fig. 1d). This represents the concurrent demands of multiple products impacting a configuration of reusable components; exemplified by domain engineering or product line architectures. The critical differentiator between this relationship and the “simpler” product decomposition pattern is that every reuse of a common component creates an implicit coupling between two products using it. What were independent products can now have their plans and schedules intertwined as a result of decisions made regarding the reusable components they share. This situation is illustrated in the Release Planning scenario described later in this paper.

Such a highly interdependent environment creates a challenge for all stakeholders in the enterprise. As described before, each product can place competing and potentially conflicting demands upon the reused components. Component development teams have to plan releases as best they can and also balance the priorities across the different products they serve. Product managers, on the other hand, may have requirements that necessitate modification to multiple components and so have to negotiate with each of these groups to ensure that they can meet the product schedule. As new agreements are reached, however, they have to be reviewed and approved by the other product and component stakeholder groups that may be affected.
The interdependencies inherent in this pattern can therefore be seen to be somewhat outside the realm of traditional management disciplines. PM and CM techniques cannot be assumed to be applicable under these conditions which are significantly different to those they were designed to address. Specifically, planning cannot be conducted for any single component or product in isolation. Attempting to arbitrarily favor one stakeholder over another can make reuse unsustainable and subject to the observed tensions [11]. Instead, an acceptable master plan for the entire enterprise has to be negotiated that meets the enterprise’s business objectives and encompasses the individual plans for the component and product groups.

With this level of interdependency, no single product or component can plan its release schedule in isolation. The multiplicity of conflicting demands and priorities can only be resolved at the business, or enterprise level. All stakeholders in the many-to-many relationship have to be represented and involved in the release planning process and must accept any compromises necessary as a result of agreed business priorities. A necessary feature of this planning is that it has to be done holistically, so that a master plan for the enterprise is developed with each individual stakeholder’s plan synchronized and compatible with the whole. Achieving this, however, is no simple task. Too often in such complex environments it can be expected that
releases will address the needs of the most dominant group or individual rather than the business priorities.

3 Implications of Product Complexity

The last stage of the model described is the most general case and encompasses all previous “stages of evolution” of software architectures. Stepping through the different stages of this heuristic device can be viewed as an historical tour of the increasing architectural complexity and its associated management patterns. The research question that this poses is: How does reality fit the model? This topic is one that is worthy of further investigation, however, the fact is that current system complexity can be shown to have reached the most complex level in the model – without the benefit of correspondingly sophisticated management techniques.

3.1. Pattern Recognition in PLAs

The many-to-many relationship between products and components in complex software architecture are illustrated in Fig. 2. Such a relationship is readily recognizable in PLAs [3, 10] and it has been proposed that it be elevated to the status of a management pattern. This has been termed the “Marketplace Pattern” [10] in recognition of the number of potential stakeholders that could contribute to a PLA, both internal and external, to an organization.

While further research is necessary to determine the prevalence of the Marketplace Pattern outside of PLAs (and its wider acceptance within the PLA community), there are a number of scenarios where it can be observed in practice. To recognize these we have to identify situations where the following two, necessary conditions are satisfied:

(i) A configuration of (partially) reusable components or sub-systems exists or will be brought into existence; and,

(ii) There are multiple products (or more generally, customers) that make concurrent demands impacting this configuration.
The above criteria suggest that the pattern may be identified in many complex software development environments. The increasing utilization of COTS products or components [12], such as the middleware necessary in modern CDB technologies like the .NET and Java platforms, makes this management pattern almost unavoidable. Similarly, the systematic reuse inherent in domain engineering and the resulting software frameworks can be expected to be subject to such a pattern, along with the general category of business IS architectures.

3.2. Related Management Techniques

Modern software architectures have been shown to be susceptible to the highest levels of complexity described by the heuristic model introduced in this paper. The consequence of this complexity is that it creates a challenge that, it is argued, can exceed the capabilities of traditional management disciplines like PM and CM when applied in isolation.

The study of the growing complexity the model allows suggests that more integrated approaches to managing these highly complex situations is required. These approaches must extend the “classic” single-product, single-project disciplines so that they address the growing interdependencies that characterize modern systems architectures. PM and CM need to play multi-dimensional roles in enterprises employing large-scale component reuse, and need to be applied concurrently, both at the original, single-product level they were designed for, as well as at the enterprise level.

It is increasingly recognized that the CM problem in CBD and PLA environments significantly increase [5, 7, 13], and that common CM techniques are often inadequate or have to be “stretched” to tackle the different levels at which the challenges manifest. Similarly, there have been calls for new paradigms in PM to address increasing complexity [14] and simultaneity [15] of projects. This, and the increasing interest in Agile Methods [16] that question traditional plan-driven development approaches, indicate a growing dissatisfaction with the adequacy of the classical management techniques.
Yet the situations where these disciplines fall short relate to relatively common everyday scenarios for CDB practitioners. Even minor changes to a component based architecture can, because of reuse, impact the plans of numerous related products – and even seemingly unrelated components. The simple example below highlights this situation and indicates the need for the entire enterprise (comprising products and components) to be managed as a configuration itself so that any changes can be analyzed holistically and their ramifications understood and anticipated.

3.3. Release Planning Scenario

As a real-world example of the inter-dependencies that have to be addressed, consider this simple scenario. A Billing System has a high-priority requirement to support a more elaborate bill-print format that displays a variable customer greeting. Achieving this capability might require coordinated changes to (say) the display, storage and print-manager components. Changing just one component alone is inadequate since all three capabilities (and so components) have to be released together to achieve the desired business result.

Add to this scenario the fact that the display and print-manager components are reused by another product that has a low priority request to support a new output device. It becomes evident that whenever there is reuse there arises the possibility of conflicting priorities that have to be managed. Thus the question: with limited resources, which requirement should the print-manager component team work on first? Irrespective of the apparent urgency, it may be that longer timescales for changing the display and storage components make it pointless for the print-manager component team to schedule the corresponding formatting changes with the same priority. In that case the smarter thing for the team to do would be to work on support of the new output device first. After that task is completed, then the formatting changes can be implemented, still allowing the synchronized release of the new bill-print format at the earliest possible time.

Juggling the priorities and balancing the competing concerns across products and components is what release planning in a product line environment is primarily about. To manage limited
resources optimally requires the ability to easily recognize and adjust for the inter-dependencies between the change activities and the components that they affect.

4 Holistic Management

The previous scenario highlights the fact that a web of dependencies, not always recognized or managed, joins all products based upon a shared software framework. Management of these relationships has been shown to be a difficult exercise that can lead to problems [17]. The argument has been made that dependencies in software need to be separately managed [18], and the explicit recording of the component relationships is a key requirement of software release management [7]. But component releases must also be kept consistent so that changes do not render them incompatible with each other. This requires that release planning and management take place holistically, at the enterprise-level, taking into consideration the entire architecture and all the product and component stakeholders involved.

The challenge of managing complex, product-line architectures both at the individual component or product level as well as at the enterprise-level, gave rise to the concept of a Release Matrix that has been introduced as the multi-dimensional generalization of traditional release plans [10] and is summarized below.

4.1. The Release Matrix

In situations where concurrent product development is based upon a reusable, component based framework, there are two distinct, orthogonal management views that can be taken of development – the first based on the products and the second based on the components. A matrix representation has been proposed to capture these two viewpoints, thus explicitly identifying the product and component stakeholders so that their perspectives and interdependencies can be clarified. These matrices can be seen to be a manifestation of the most general, many-to-many case of the meta-model described, and as such, provide an organizing principle to help consider different facets of a complex software program. The matrices offer the ability to record relevant, lifecycle information in their cells, while capturing the relationships that information has with
other members of the ecosystem. This traceability provides a context that can support the incremental planning and evolution of the component based architecture by early identification of the possible impacts of a change.

In particular, the Release Matrix has been proposed as a means of planning and tracking the evolution of the system architecture over time. As shown in Fig. 3, the Release Matrix records the components (x-axis) and the products (y-axis) that use these components, integrating them into the market offering. The matrix can be seen to correspond to the relationships shown in Fig. 2 where the existence of a relationship between a product ($P_i$) and component ($C_j$) results in an entry in the intersecting cell ($r_{ij}$). When no relationship exists between a product and component there is a zero or null entry in the corresponding cell.

The content of the cells of a Release Matrix can be simply regarded as the scheduled dates of the set of dependent releases, however a family of similar matrices can be employed to record different lifecycle data depending upon the utilization of the matrix. For example, in order to derive and coordinate the release schedules for all products and components, a separate matrix can be used to record the product requirements that have been allocated to the different components. With reference to the release planning scenario previously described, the P2 row could represent the bill-print project where C2, C3 and C4 would be the display, storage and print-management components.

![Fig. 3. The Release Matrix consolidates the product and component perspectives](image-url)
The multiplicity of releases that are a feature of CBD environments can benefit from the clarification offered by the Release Matrix. It provides a succinct and explicit means of recording the dependencies between products and components, while supporting the “separation of concerns” principle by extricating the two perspectives. Each row of the Release Matrix represents a product’s release plan that is derived from, and must be compatible with the component releases that the product is reliant upon. Similarly, each column represents a component’s release plan, based upon the total set of product requirements that are to be implemented in the release. As a whole, the Release Matrix represents a master release plan that consolidates and synchronizes the individual release plans of both the products and the components.

By way of example, the highlighted column in Fig. 3 represents the perspective of the team responsible for component C3 that needs to balance the demands of products P2 and P3. The highlighted row corresponds to the reliance that project P2 has on three components, C2, C3 and C4, that may need to have coordinated releases to effect a business change. The intersection of these two perspectives indicates the specific plan or contract between the P2 and C3 teams. Similar plans must be negotiated for all non-null cells as they point to a dependency between the stakeholders that requires coordinated attention in order to achieve an agreed and consistent set of releases.

In general, Fig. 3 shows that each product group must attempt to align the components it relies upon across the row, while each component producer must weigh and prioritize the requirements of its customers shown in that column. These orthogonal perspectives represent the different tensions that have to be balanced in a complex component based environment.
4.2. Validating the Release Matrix

While the Release Matrix offers a simple, integrating technique to visualize and balance the competing priorities of the different stakeholders, it actually resulted from the recognition of the orthogonality of the product and component perspectives and concerns.

![Fig. 4. Normalizing the product-component relationship](image)

The heuristic model has shown that the most complex management pattern is characterized by the many-to-many relationship between products and components. From a data modeling perspective, this is a candidate for normalization and standard techniques for resolving such relationships leads to the introduction of a new associative entity [19]. Therefore normalization of the meta-model introduces the Release Matrix shown in Fig. 4, that represents the time-based record of the evolving software architecture. While this result could be anticipated from the existence of the many-to-many relationship, the refined model helps establish the matrix representation, and the Release Matrix, as appropriate constructs for today’s architectural complexity.

4.4. Application of the Release Matrix

The Release Matrix is proposed as a necessary generalization of an individual product’s (or component’s) release. It captures a snapshot of the evolution of complex, component based
architectures, where an adjustment to any element of the plan can have multiple impacts requiring intense communication and collaboration to achieve an agreed master plan.

The Release Matrix can therefore provide a consolidation mechanism for planning or negotiating incremental release schedules and can document the agreements and expectations of each stakeholder involved. As such, it can be record and be queried for critical management information, including:

- the component releases upon which each project is reliant
- the products that are reliant upon a component’s scheduled release
- the scheduled release dates of all parties forming the matrix
- the component that is released last and so drives the schedule
- the impact of any schedule delays on the master release plan
- the component and product versions that constitute an architectural baseline

5 Conclusion

This paper has utilized a meta-model as a heuristic device to discuss the effects of increased architectural complexity on familiar management disciplines. The model is described in stages that correspond to the growing adoption of CBD and systematic software reuse. As such, it provides a chronological context for the management patterns that are appropriate for the development and release of complex software products. The aim of this exploration is to provide further rationale for the Release Matrix that has previously been proposed as technique for planning and recording the evolution of complex software architectures, such as those used in software product lines.

This research is ongoing, and current limitations include the fact that the Release Matrix is currently presented as a concept that requires further exploration and detailing. Thus, further studies are underway to validate the concept and further develop the technique. Current research is therefore focused on the real-world application of the Release Matrix and a case study can be expected shortly. The extension of the matrix representation to support the entire life cycle is
also being explored with a view to developing a process-model for complex software
development.

References


