Using RUP to Implement a Packaged Application

by **Serge Charbonneau**
Software Engineering Specialist
IBM Rational

The software engineering best practices implemented in the Rational Unified Process,® or RUP,® have been applied successfully many times on "green-field" development projects. Now, the emerging trend toward basing new applications on Commercial Off-The-Shelf (COTS) components and packaged applications raises an important question: Is the RUP well-suited for developing applications based on COTS components and packaged applications?

Based on Rational's experiences with customer projects as well as an internal project to implement a corporate-wide Customer Relationship Management (CRM) solution with the Siebel packaged application, this article shows that the RUP's software engineering best practices are just as applicable to COTS and packaged application implementations as they are to green-field development. It also proposes a set of best practices specific to COTS and packaged application implementations, which range from simple libraries that provide services (e.g., math libraries) to complex packaged applications that automate entire business processes (e.g., CRM packaged applications).

Project Avalon, Rational's internal CRM solution implementation, began in 2000 and will require another few years before all the CRM capabilities are fully deployed. So far, we have deployed three major releases and multiple maintenance releases either on time or very close to schedule. Also, we have replaced multiple legacy systems with the new product capabilities. Over a thousand users worldwide from Rational's marketing, sales, and customer support organizations currently use the product.

For a large organization, implementing a corporate-wide CRM solution is a long-term process spanning many years. As some of the information presented here is based on ongoing projects, the practices we describe may continue to evolve.

Preparing for a Packaged Solution Implementation

Let's begin by looking at some of the organizational and technical preparation that should take place before you implement a packaged application but that is not specifically mandated in RUP.

Implement Organizational Changes

In any organization, there is always resistance to implementing changes, and implementing a CRM solution is no different. Two groups are key to successfully implementing any change that impacts both an organization and its people:

- **Executive managers** *deal with change at the organizational level. They are responsible for defining the strategic direction and vision for new organizational practices.* They need to understand where the organization is and define where it needs to be. They also need to understand the risks associated with the proposed changes and the impact these changes will have on the organization. Executive managers maintain the new course when the organization experiences an initial decrease in productivity and individuals resist the new ways of doing things.
- **Champions** *are responsible for communicating the strategic direction and vision of executive management to the employees.* They need to be agents of change who can translate abstract, high-level strategic planning into concrete, tactical execution. They need to lead the way in blazing trails of change that other people will follow. Champions enforce the adoption of new automated and manual practices at the individual level.

When automating business practices with a packaged application, an organization must find its balance between two extreme approaches:

1. *Deploy the packaged application as-is, with minimal or no configuration,* and force the organization to adapt all its practices to the capabilities of the new application;

   or

2. *Completely adapt the packaged application to the current practices of the organization* with either minimal or no changes to these practices.

Organizations that use the first approach often impose an automated solution that is so ill suited to the needs of the organization and that

represents so much risk that workers prefer to keep using their legacy automated and/or manual practices. In such cases, workers *never actually transition to the new automated solution, and the funds, time, and effort invested in the new solution is ultimately wasted.*

At the other end of the spectrum, organizations that use the second approach often find that the application *requires so much customization effort that the organization may lose all the benefits of having purchased a packaged application in the first place*. In the end, they may have been better off simply building an application from scratch. In such cases, the organization's expectation that it would be able to provide a quick solution for the whole enterprise, based on all the out-of-the-box capabilities of the packaged application, cannot be met because the IT team must spend too much time customizing the application in every release. Also, migrating to a new version of the packaged application becomes a real nightmare because the customization is so extensive. Finally, the problem is amplified if the packaged application is hard to customize.

For most organizations, a successful approach to automating business processes using a packaged application is one that is half way between these two extremes, and that meets both of the following conditions:

1. The organizational practices are adapted to the new capabilities of the application;

   and

2. The application is configured to adapt it to the organizational practices.

Any organization implementing a packaged application should find its balance between these two extremes and stick to it. During the implementation, there will surely be pressure from different directions to go more one way or the other. Again, it is the responsibility of the organization's executive management to keep the initiative on the right path.

Manage Data

There are two important aspects to managing data in preparation for implementing a packaged solution: migrating legacy data and enforcing data quality.

Migrate Legacy Data

If an organization has an existing CRM solution, implementing a new CRM solution often involves retiring parts of or the entire legacy solution. Before these systems can be retired, some of their legacy data may be migrated to the new CRM system.

Migrating legacy data involves the following six activities:

1. Identify data to be migrated.
2. Create a logical mapping.
3. Create a physical mapping.
4. Automate the data migration process.
5. Validate the migration process.
6. Migrate legacy data to production.

Identify data to be migrated. Systems that have been active for many years often have much legacy data. If poor rules were implemented to enforce data quality in these systems, or rules were nonexistent, a significant portion of the data may be either incomplete or obsolete. In these cases, candidate migration data should be selected with care and cleansed before it is migrated. Even before that, however, you should *discard irrelevant data* to avoid wasting effort on migrating and cleansing it.

Create a logical mapping. Before comparing the details of source and destination data attributes, such as physical size (byte, double byte, quad byte, etc.) and type (numerical, string, char, etc.), it is *important to identify which attributes of source objects will be migrated to attributes of destination objects*. It is also important to map values between source and destination multi-value (list of values) attributes.

Data modeling should be performed on legacy data objects using legacy system information, and on destination data objects using the CRM packaged application. If a domain model or business object model exists, you can leverage it to identify important legacy data that needs to be preserved. The domain model and the legacy data model also help identify gaps between the legacy data structure and the CRM packaged application data structure.

Create a physical mapping. Once you have a logical mapping between source and destination objects, the next step is to resolve incompatibilities between data types, data size, and list of values.

Automate the data migration process. The Extraction, Transformation, and Load (ETL) process is usually automated using scripting capabilities. Some tools can help in automating the migration process.

Validate the migration process. To mitigate the risk of migrating data from legacy systems to the new production environment, it is wise to rehearse the migration in a test environment. Any problems identified during this rehearsal should be corrected prior to migration, either by updating the migration process or by cleansing the data.

Migrate legacy data to production. Before migrating data to production, back up the production database. Then, when migration is complete, perform a smoke test on the new production database that includes the newly migrated data. If test results indicate that the production database is unstable because of the migration or that data is corrupted for any reason, then roll the backed-up data back into the

production database. Perform an analysis to determine the root cause of the migration problems so you can apply any corrective action before the next migration attempt.

Enforce Data Quality

Data can be cleansed either prior to or after migration, depending on the situation. If the new system has rules to enforce cleanliness, such as mandatory attributes or well-formed rules, the legacy data may need to be cleansed prior to migration because the new system may not accept an object that is missing a mandatory attribute or that has an attribute value with an incorrect format. For a state-driven object, the list of mandatory attributes may change as the object evolves through its lifecycle, which makes the issue even more complex.

There are two ways to enforce data quality rules:

1. **Automated data quality rules.** These affect quality of data at two levels.
    - **Attribute Level.** This level validates the range of values and/or value format for an individual attribute. For example, the birth date of a contact is a date in the past, not the future, or an e-mail address has the following format: xxx@yyy.zzz.
    - **Object Level.** This level validates that the attributes within a group are consistent with one another. For example, the address city and state are consistent with the address zip code.
2. **Manual data quality processes and procedures**. These include written instructions on how to enter data into the system. End users entering data into the system enforce these processes and procedures.

Adopting Software Engineering Best Practices

Now that we have looked at practices outside the RUP that help in implementing packaged applications, let us turn to how standard RUP best practices apply in this situation. This section describes how Rational's Project Avalon and other projects applied RUP best practices for a packaged CRM implementation. The following subsections will look specifically at six core practices:

- Develop iteratively
- Manage requirements
- Use component architectures
- Model visually

- Continuously verify quality
- Manage change

Develop Iteratively

For medium to large organizations, developing and deploying a CRM solution, or other complex, pre-packaged application, is usually a multiyear, multi-release endeavor. Each release provides users with added capabilities, and each encompasses the entire RUP lifecycle, which consists of four phases: Inception, Elaboration, Construction, and Transition.

Multiple Development Cycles. The first release is the outcome of an initial development cycle, and follow-on releases are the outcome of evolution cycles. Table 1 lists the major activities undertaken in each phase for each cycle.

- The **initial development cycle** focuses on defining the high-level vision and plan for not only the first release, but also for the development and deployment of the entire product, including all its incremental releases. It is in this cycle that the packaged application is selected and purchased. Some core capabilities that follow-on releases depend on are usually enabled in this development cycle.

- The follow-on **evolution cycles** focus on adding new capabilities and/or on refining existing capabilities by implementing enhancement requests and fixing defects. The enterprise architecture evolves through all these cycles as legacy systems are retired and replaced by the new solution and/or modified to make them compatible with the new solution. Each cycle ends with a deployable release. Cycles can overlap in the same way that iterations do.

Table 1: Activities for Multiple Development Cycles

Initial Development Cycle

- Inception Phase
  - Create initial business use-case model and business-object model.
  - Define a vision for the CRM initiative.
  - Capture stakeholders' main needs.
  - Derive system features from main stakeholder needs.
  - Create initial system use-case model based on business use-case model, business object model, and system features.
  - Prioritize system features and use cases based on various drivers, including business drivers and risks.
  - Define scope of current development cycle based on

prioritized list of use cases.

- Elaboration Phase
  - Drive CRM packaged application selection using prioritized list of system features.
  - Create an executable architecture based on selected CRM packaged application and architecturally significant system use cases and/or scenarios included in the scope of the current development cycle.
- Construction Phase
  - Implement remaining use cases and/or scenarios included in the scope of the current development cycle.
  - Deploy partial release internally.
- Transition Phase
  - Implement remaining use cases and/or scenarios included in the scope of the current development cycle, if any.
  - Correct defects.
  - Deploy release externally to end-user community.

Evolution Cycles

- Inception Phase
  - Refine business use-case model, business-object model, and system use case model.
  - Define scope of current development cycle, based on outstanding use cases and/or scenarios not realized in previous development cycles.
- Elaboration Phase
  - Refine architecture using architecturally significant system use cases and/or scenarios included in the scope of the current development cycle.
- Construction Phase
  - Implement remaining use cases and/or scenarios included in the scope of the current development cycle.
  - Deploy partial release internally.
- Transition Phase
  - Implement remaining use cases and/or scenarios, if any, included in the scope of the current development cycle. Correct defects.
  - Deploy release externally to end-user community.

Note that the initial development cycle for these projects differs significantly from that of a green-field development projects. For green-field development projects, the initial development cycle (a full RUP lifecycle that includes Inception, Elaboration, Construction, and Transition phases) focuses on building an executable architecture from scratch and adding in the initial set of product capabilities. *For packaged application implementation projects, the initial development cycle focuses on building an executable architecture based on a COTS packaged application, and on enabling and configuring an initial set of product capabilities.*

Prioritize Release Content. To plan multiple development cycles, you must first determine what content is most critical to develop and release. Prioritize release content based on three main sets of drivers:

1. **Business drivers.** These typically include deriving the strongest return on investment from the early releases. For instance, if the CRM solution will be deployed across the organization's marketing, sales, and customer support departments, early releases should emphasize sales capabilities, since the sales department generates the organization's revenues.

2. **Packaged application/business process dependencies.** Application architectures often reflect business process dependencies, so these should also influence release priorities. In CRM, for example, opportunity management requires contact and account management, so opportunity management should be implemented either in the same release as contact and account management or in a follow-on release.

3. **Enterprise architecture.** This also influences release priorities. Even if the new application is meant to replace existing systems, it is unlikely that you will be able to disable all the legacy systems in the same release. Or, if you are integrating the new application with existing systems, it is also unlikely that you will be able to implement all integration points in the same release. Furthermore, you may need to build temporary interfaces or capabilities to manage the transition from your legacy systems to the new solution.

On Rational's Project Avalon, our development cycles focused on implementing and deploying incremental capabilities for the main CRM application stakeholders: marketing, sales, and customer support. We prioritized our release content as follows:

- **Release 1** (initial development cycle) delivered basic contact and account management for all stakeholders.
- **Release 2** (first evolution cycle) delivered basic campaign management capabilities for marketing and basic opportunity management capabilities for sales.

- **Release 3** (second evolution cycle) delivered extended opportunity management capabilities for sales, and service request management and solutions management capabilities for customer support.
- **Future releases** will include capabilities for quoting, forecasting, order management, and others.

Figure 1 shows dependencies between some of the CRM packaged application features. For example, forecasting is dependent on opportunity management, and opportunity management is dependent on both contact and account management.

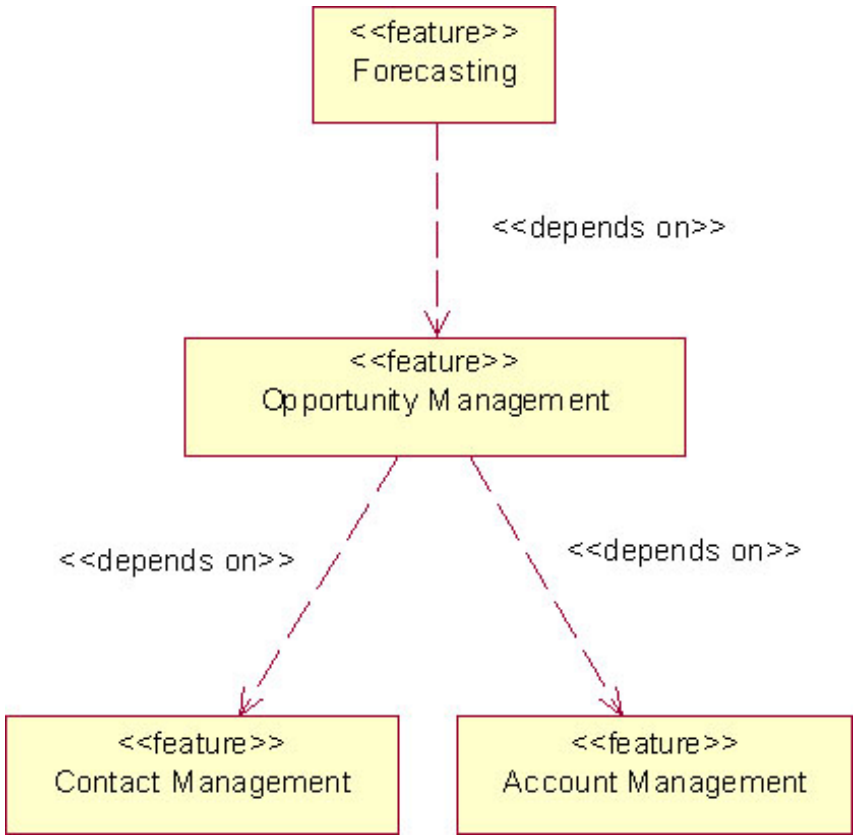

Figure 1: Dependencies Between CRM Packaged Application Features

Prioritizing release content effectively can help spread out the overall purchase cost for the packaged application. If some modules or features need not be implemented within the next six months to a year, it may be possible to negotiate a deal with the vendor to purchase them later.

Manage Requirements

To manage the complexity of a CRM packaged application implementation, you need to manage requirements at many levels:

- **Stakeholder needs.** First, capture the needs and desires of project stakeholders.
- **System features.** Based on stakeholder needs and desires, along with your understanding of the desired business behavior and

organizational structure, describe system features.

- **Software use cases and supplementary specifications.** Specify use cases that fulfill the functional and nonfunctional requirements, for the automated portion of the new CRM solution.

As we will describe in more detail later on, stakeholder needs and system features are high-level requirements that should be elicited prior to selecting the packaged application and then used to drive the selection. Use cases and supplementary specifications are software requirements that should be elicited based on the stakeholder requests and system features, and on the capabilities of the packaged application.

Traceability. It is also important to define and implement a traceability strategy to ensure that all higher-level requirements are realized by lower-level requirements and verified by test cases. Traceability also helps you analyze the potential impact of implementing change requests and correcting defects.

Associating Attributes. When documenting requirements, it is useful to associate them with a comprehensive set of attributes:

- **Priority** to manage scope and ensure that high-priority requirements are implemented before low-priority requirements.
- **Status** to allow progress tracking.
- **Level of effort or cost** to manage scope and estimate effort required for the realization of a set of requirements allocated to an iteration or release.
- **Other attributes** may also be useful, such as risk, stability, difficulty, and so forth.

Eliciting Requirements for a Packaged Application. You should elicit stakeholder needs and define system features *before* you select a packaged application and then use this information to drive the selection process. (We'll say more about this in the **Additional Best Practices** section below.) Then, you can define your software requirements after you actually select the packaged application. In defining requirements for the Project Avalon CRM system, analysts quickly realized that it was not productive to define requirements without reference to the packaged application's out-of-the-box capabilities. If your requirements are not "aligned" with the capabilities of your packaged application, then implementing them can require so much customization that you lose all the benefits of starting with a packaged application. A more successful approach is to **use the packaged application itself as a main source of input for defining software requirements.** Stakeholders responsible for providing requirements input should be aware of the packaged application's capabilities so that they have a framework in which to express their needs and desires. Better yet, you can involve a CRM packaged application expert -- a Subject Matter Expert (SME) -- in the requirements elicitation effort.

Before they elicit software requirements, it is imperative that analysts

have a good understanding of the business processes that the new CRM solution will impact. Modeling business processes and structure in isolation from the packaged application facilitates this understanding. System analysts and business representatives can then capture stakeholder needs, derive features, and define a preliminary use-case model, also in isolation from the CRM packaged application. *They should take the packaged application's capabilities into account only when refining the use-case model and fleshing out each use case.*

Figure 2 shows the different types of requirements managed on Project Avalon and the traceability between them. High-level requirements in the hierarchy shown on the diagram, including stakeholder needs, features, use cases and supplementary specifications, were defined in isolation from the capabilities of the CRM packaged application. The CRM packaged application was used as input in defining lower-level requirements, including use-case details, supplementary requirements, design specifications, and test cases.

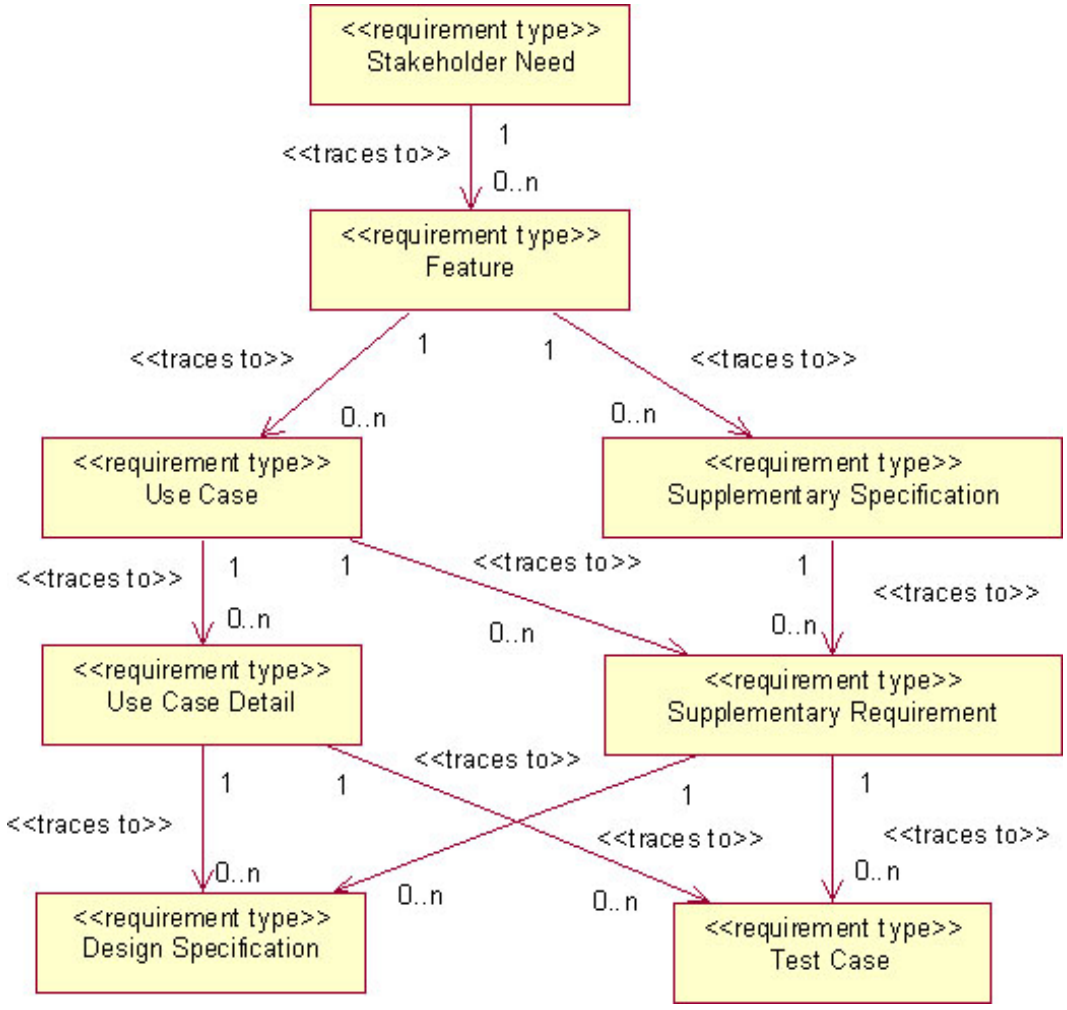

Figure 2: Rational Project Avalon Requirements Traceability Strategy

Specify Design Textually. After Project Avalon had selected a packaged application, Rational invited several CRM packaged application experts to join the Rational Project Avalon team members for a while. These experts played different roles on the project: system analysts, database analysts, and developers. They came on board with their own baggage of software

development practices that were very different from the RUP's software engineering best practices. As a result, Project Avalon process engineers created a hybrid process combining RUP and the experts' process. *This **RUP variant** leveraged the best of both worlds for implementing a packaged application*. Textual design specifications were one of the most successful aspects of this process.

Design specifications textually describe what needs to be implemented in the CRM packaged application to make it compliant with a set of software requirements. The design specifications are expressed using semantics specific to the CRM packaged application. Categories for these specifications, with examples (in parentheses), are listed below:

- *Create a workflow to automate a business rule* (e.g., setting a flag on a record when someone other than its owner modifies that record).
- *Extend a base table to add attributes to existing business objects or components* (e.g., adding a Cellular Phone attribute to the Contact object).
- *Modify the properties of an existing record field* (e.g., making a field mandatory on the creation of a record).
- *Create rules for the assignment of records to users* (e.g., assigning an opportunity based on the address of the account this opportunity is associated to).
- *Create new (or modify existing) user interface elements such as screens, views, or applets.*
- *Associate a script to a business object, business component, or user interface element.*
- *Modify the properties of the application through administration* (e.g., changing a list of values associated with a field).

Textual specifications offer two main advantages:

- They provide a description of what needs to be implemented that developers can understand without having a UML background.
- They provide a design representation of required changes to the application without describing capabilities that are already implemented.

Use Component Architectures

This best practice emphasizes building an *executable architecture* and making it component-based. Focusing on architecture means that a project should implement the 20 percent of the application that provides 80 percent of the value first. It also means that an infrastructure should be created to facilitate the implementation of the remaining 80 percent. The architecture should be component-based because components are self-contained, cohesive elements with well-defined interfaces that can be purchased, reused, or custom-built.

The package application itself should also be based on a sound architecture that is flexible and adaptable. This allows for the incremental implementation of application capabilities. In addition, the entire package application can become a component of an enterprise architecture, which may include components such as a CRM application, a financial application, and an Enterprise Resource Planning (ERP) application.

Obviously, a packaged application has a major influence on the system's architecture. CRM systems often integrate with other systems, including financial and ERP systems. So in this context, building an executable architecture would include enabling some of the CRM packaged application's basic, out-of-the-box capabilities, and creating integrations between the packaged application and these other systems.

Select a Packaged Application That Matches Your Feature Needs and Business Practices. On Project Avalon, the CRM packaged application selection was driven by the system feature requirements and CRM business practices. Rational compared its desired features and business practices with the features of different CRM packaged applications and selected the one that offered the greatest correspondence. Of course, other business-related factors were taken into consideration as well: the quality of the vendor's professional services, the vendor's market position, product price, and many other non-technical factors.

Another difficult-to-measure aspect was also taken into account: *how easily the packaged application could evolve as Rational's CRM practices evolve*. If you implement a solution to streamline your CRM practices, once you reach your goal, you may want to optimize business practices by making further improvements to some of these practices. Will the CRM packaged application facilitate the optimization and evolution of these practices? Again, if the CRM packaged application is based on a sound architecture, it should be resilient to the changing CRM practices requiring automation.

Model Visually

To get a correct and complete understanding of all the different aspects of implementing a business solution, you need to model behavior and structure from different perspectives:

- **"As-is" business model perspective.** This includes a business use-case model and business-object model to capture current business practices and organizational structure before implementing the new solution.
- **"To-be" business model perspective.** This includes a business use-case model and business-object model to capture desired practices and organizational structure once implementation of the new solution is complete.
- **"As-is" system model perspective.** This includes a system use-case model and analysis model to capture the organization's current

solution (e.g., a multi-application CRM solution) if one exists.

- **"To-be" system model perspective.** This includes a system use-case model and analysis model to capture the planned packaged software implementation once it is deployed.
- **Packaged application system model perspective.** This includes a system use-case model and analysis model, capturing the out-of-the-box packaged application behavior and structure.

Modeling all these different perspectives requires significant effort and should only be done if it adds value to the implementation initiative. A project should identify and create only those models that will provide high return on investment.

Project Avalon leveraged business modeling to understand Rational's current and desired CRM business practices. The project team saw a lot of value in capturing business processes that would affect, and be affected by, the new CRM automation. They saw less value in creating detailed system models, knowing that 80 to 90 percent of the behavior and structure they needed was available out of the box. So the project performed very little system modeling.

Gap Analysis. *Gap analysis identifies the differences between system requirements and the packaged application's evolving capabilities as the implementation progresses.* The analysis describes how the packaged application still needs to be modified to make it fully compliant with system requirements. Figure 3 visually shows the gap-analysis approach.

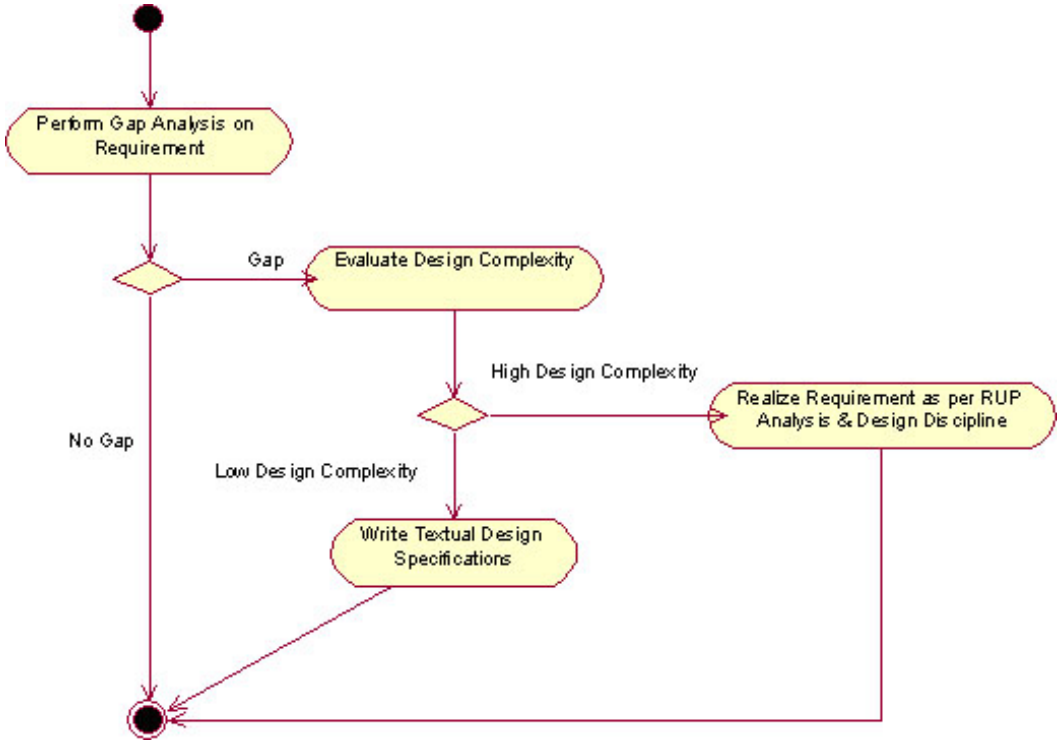

Figure 3: Gap Analysis Process

Project Avalon performs gap analysis once per iteration based on the software requirements, including use cases, scenarios, and/or

supplementary specifications, allocated to the iteration. During gap analysis, the team "compares" each requirement with the CRM application's capabilities to date. If the requirement is fully implemented in the application, then no action is taken. If a requirement is partially implemented or not implemented at all, then the team evaluates how "complex" the requirement will be to realize. For complex requirements they do further analysis, using activities defined in the RUP Analysis and Design discipline, including visual modeling using UML constructs. On the other hand, if the realization for a requirement is determined to be "simple" then the team writes textual design specifications instead, identifying what to change in the application to comply with the requirement.

For Project Avalon, *use cases and scenarios involving integration with other systems turned out to be rather complex*, so they were realized using the RUP Analysis and Design discipline approach. However, most of the *requirements that did not involve integration with other systems turned out to be simpler* and were realized using a textual design description approach.

The CRM packaged application for Project Avalon is very much a data-centric system in which each class of persistence objects has some Graphical User Interface (GUI) and associated behavior. Because these elements are so closely tied together, it makes sense to express the behavior and structure of the analysis model in terms of the application GUI screens and views. "Aligning" analysis classes and objects with implementation elements makes the effort much more efficient and pragmatic.

When realizing requirements using the RUP Analysis and Design discipline activities, it may make sense to perform the standard analysis activities but limit design to gap analysis. This provides enough information to create textual design specifications without modeling the design visually. Typically, the latter does not pay off, *as the biggest benefits of having a design model -- such as the ability to forward engineer the code from the design -- are not applicable or available if you are using a packaged application*.

Continuously Verify Quality

To ensure that the implementation fulfills stakeholder needs, the project team derives test cases from software requirements. These test cases focus on a variety of concerns that come into play throughout the iterative development lifecycle: functionality, performance and load capabilities, and other issues. The development team is responsible for unit and integration testing in the development environment, and the test team is responsible for system testing in the test environment. Business stakeholders are responsible for User Acceptance Testing (UAT), which is used to decide whether a build is ready to be released to the end-user community.

On Project Avalon, business stakeholders are responsible not only for UAT, but also for testing the application at the end of internal iterations, even when those iterations are not released externally. Involving business

stakeholders early and often is a good strategy to mitigate the risks of adopting new capabilities and of undiscovered defects.

Manage Change

This best practice deals with management of the changing artifacts and changing requirements of a project. It describes how to manage incremental versions of artifacts and their baselines as well as change requests that may impact already baselined project requirements. Briefly, here is how Project Avalon and other Rational-led projects adopted the *manage change* best practice:

- Rely on the limited Configuration Management (CM) capabilities of the CRM packaged application's Integrated Development Environment (IDE) for managing daily updates to the application database repository schema.
- Put archived application database repository schema under CM only at milestones and/or at regular intervals (at the end of an iteration, phase or release, and/or once a day or once a week).
- Put only application schema under CM control, not user data. Use standard back-up procedures at regular intervals instead to preserve production user data (daily).
- Control versions of evolving supporting artifacts including models, documents, and archived requirement and change request databases.
- Baseline all project artifacts at minor (end of iteration) and major (end of phase or release) milestones.
- Manage change requests and defects.

The discussion below explains this strategy in more detail.

Configuration Management of a Database-Centric Application. The CRM packaged application is a database-centric application. Both the database schema -- which includes the definition of the persistence layer, business logic, and the user interface -- and the user data reside in a single database. The application is composed of modules that are further subdivided into projects. Each project contains some persistence layer elements, some business logic, and some user interface elements.

A packaged application's IDE allows developers to modify the database's out-of-the-box schema. Using a basic checkin and checkout mechanism, they can work exclusively on a single project and concurrently on the same database. Since the IDE doesn't have an Application Programmable Interface (API) that allows integration with a third-party CM tool, Project Avalon made no attempt to manage the configuration of units at the project level outside the IDE. Instead, the database schema, or repository, was archived at regular intervals and put under CM.

Because of the large amount of user data in the database, our recommendation is to archive only the schema portion of the database,

not user data. Traditional database backup mechanisms should be used for archiving user data.

Development, Test, and Production Environments. Project Avalon manages three environments, each with its own database: Development, Test, and Production. These environments are used for different purposes:

- All new capabilities, as well as enhancement requests and defect fixes, are implemented in **Development**.
- Unit and integration testing are also performed in **Development**.
- All other testing, including internal system testing and UAT, is performed in **Test**.
- End-users perform their daily activities using **Production**, where user data is created and maintained.

Daily backups are performed on both Development and Production. Development is backed up to preserve its schema, and Production is backed up to preserve its user data. The project did not see any value in backing up Test, because its schema is based on an archived baseline version of the CRM Repository that is preserved under CM, and because its user data is extracted from Production.

Figure 4 shows the different Project Avalon environments along with actions performed on them. Note that:

- The schema of *Development* is archived and checked into the CM repository at regular intervals or at project milestones.
- The Development schema is migrated to *Test* at the end of an implementation cycle, before the beginning of a testing cycle.
- User data is migrated from *Production* to *Development* and *Test* to allow testing using up-to-date user data. Once a release has successfully passed UAT, its schema is migrated from *Development* to *Production*.
- Both *Development* and *Production* are backed up at regular intervals to safeguard their schema and user data, respectively.

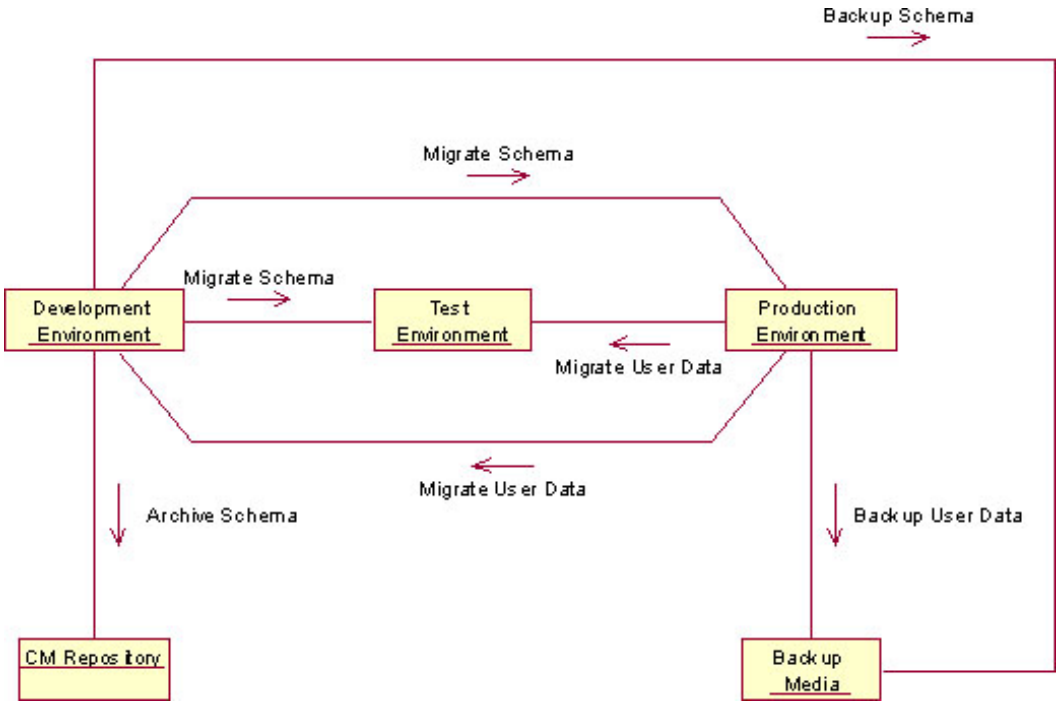

Figure 4: Project Avalon Environments and Actions

Additional Best Practices for Packaged Application Implementations

Based on our experience with Project Avalon and other Rational customer projects, we propose the additional best practices shown in Table 2 specifically for packaged implementation situations. As you'll note, we've already mentioned most of these recommendations in the course of discussing RUP best practices, but it is worthwhile to distinguish them here and explain them in greater depth below.

A cautionary note: Although these best practices are experienced-based, they were derived from only a few projects. Before we can legitimately use them to build a packaged application implementation process like a RUP variant, they need to be validated, challenged, and refined by other practitioners involved in similar projects. These proposed practices are discussed in detail in the following subsections.

Table 2: Proposed Best Practices for Packaged Application Implementations

- Apply all software engineering best practices.
- Drive packaged application selection from stakeholder needs.
- Understand the business and the application.
- Maintain the right balance between adapting the business to the application and the application to the business.
- Model analysis and design visually only when appropriate; otherwise, textually describe analysis and design.
- Implement a data quality strategy early on.

- Migrate valuable data only.

Apply All Software Engineering Best Practices

As we have discussed, software engineering best practices are as applicable to packaged application implementations as they are to green-field development. We noted that:

1. **Developing iteratively** allows the project team to implement packaged application capabilities incrementally, in multiple releases, each of which mitigates risk early.
2. **Managing requirements** ensures that both stakeholder needs and packaged application capabilities are taken into account in requirements elicitation.
3. **Using component architectures** allows the team to implement the purchased packaged application as one component of the overall enterprise architecture. It also encourages the team to purchase a packaged application with a well-designed internal component architecture.
4. **Modeling visually** encourages project analysts to model an organization's behavior and structure, as well as areas of the application requiring more complex design and implementation.
5. **Continuously verifying quality** ensures that stakeholder needs will be implemented as intended.
6. **Managing change** ensures that changes will be managed properly and reflected in the application in a controlled way

Drive Packaged Application Selection from Stakeholder Needs

The more closely the selected packaged application meets the needs of an organization, the easier it will be for the organization to implement and adopt it. Carefully define stakeholder needs and system features before proceeding with the packaged application selection process. Once you document these needs, create a scorecard. For each need, note whether each solution under consideration meets the need or not, and to what extent. If the application does not meet that need out of the box, keep in mind that you will have to find or create an alternative solution to satisfy it. The application with the best mapping between stakeholder needs and vendor features should receive the highest score. You should, of course, consider things like price, reliability, and so forth in selecting a vendor. However, comparing stakeholder needs to packaged application features is a necessary step in making the right selection.

Understand the Business and the Application

Before implementing an automated solution that will have a significant impact on the way people work, it is imperative to understand the practices, both manual and automated, that these people use daily. It is equally important to understand how the business is structured. Then, use this knowledge of business behavior and structure to derive system requirements, identify gaps between what is required and what is available in the packaged application, and identify what portions of the legacy data need to be preserved and hence migrated.

Before capturing detailed requirements on what needs to be automated in a business, it is also imperative that the stakeholders involved in requirements definition, including both system analysts and business stakeholders, have a good understanding of the packaged application's behavior and structure. If possible, involve a packaged application expert in eliciting system requirements with the system analysts and business stakeholders. This expert need not know all the details of the application implementation but should have a deep understanding of the business practices embedded in the application.

Maintain the Right Balance Between Adapting the Business to the Application and the Application to the Business

A packaged application comes out of the box with behavior and structure that automates best practices in the business area the application supports. As these practices are widely accepted and well thought of, it makes sense for an organization to adapt most of its business processes to these practices. In fact, an implementation of this sort often represents a good opportunity for an organization to improve or optimize its current practices, both manual and automated, and to standardize these practices across all groups in the organization.

However, it is also important to remember that a packaged application is a generic solution for a given business area. Since every business has its own specific practices, the application needs to be adapted to the business' specific needs.

Sometimes the desire to customize the application to the specific needs of an organization conflicts with the desire to adopt the practices of the application. For each business process, the organization needs to decide which of these two desires should prevail, based on objective criteria, if possible.

Model Analysis and Design Visually Only When Appropriate; Otherwise, Textually Describe Analysis and Design

With a packaged application, implementing capabilities available out of the box can satisfy many stakeholder requirements. So creating a visual model to analyze and design these requirements may be a bad investment. Often, textual analysis and design specifications are sufficient to let developers know how the application must be modified to comply with the requirements.

On the other hand, when requirements correspond to complex implementations, it is justifiable to express the behavior and structure in a visual model. As architects and designers gain experience in realizing requirements for a packaged application, they also gain confidence in quickly identifying that requirements will require either complex or simple implementations.

Implement a Data Quality Strategy Early On

As soon as the first release of the application is pushed to a production environment, users start entering information into the database. Even if no data is migrated from legacy systems, the amount of information usually grows rapidly. Since it is easier to enter clean data into a database than correct dirty data after the fact, manual procedures and automated rules for data quality should be implemented with the first release available to users. These rules should be extended to enforce cleanliness of all data as new features are deployed to production.

Migrate Valuable Data Only

Any database-centric system accumulates a significant amount of data over time. Some is high-quality and valuable data, but much is low quality and/or obsolete. Since there is no need to preserve data that has no value to users, only high-quality valuable data should be migrated to the new system. To mitigate the risk of not migrating data that needs to be preserved, legacy systems should be kept operational in read-only mode (no new information entered) for some time after you migrate data to the new production system.

An Invitation

I invite readers of this article to view our recommendations for implementing packaged applications as "work in progress" rather than absolute truth. If you apply them in your environment, please tell me about your experiences. Write to scharbon@ca.ibm.com

References

Cecilia Albert and Lisa Brownsword, "Evolutionary Process for Integrating COTS-Based Systems (EPIC): An Overview." *SEI Technical Report*, July 2002.

J. Dyche, *The CRM Handbook*. Addison-Wesley, 2002.

Philippe Kruchten, "A Rational Development Process." *Crosstalk*, July 1996.

Philippe Kruchten, *The Rational Unified Process: An Introduction*. Addison-Wesley, 2000.

Philippe Kruchten, "Using the RUP to Evolve a Legacy System." *The Rational Edge*, May 2001.

Rational Unified Process 2002.05.01.01.

Walker Royce, *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.

Acknowledgment

I would like to acknowledge the contribution of Dennis Moya from Rational's Strategic Services Organization (SSO), who applied tenacious objectivity in reviewing and providing feedback on this article. Thanks, man.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!