Service- and Component-based Development: Using Select Perspective™ and UML

Monday, January 13, 2020 7:55 AM

Service- and Component-based Development: Using Select Perspective™ and UML

0 REVIEWS

by Chris Simons, David Piper, Barry McGibbon, Barry Maybank, Steve Latchem, Ralph Hofman, Hedley Apperly
Publisher: Addison-Wesley Professional
Release Date: January 2003
ISBN: 9780321159854
Topic: Financial Analysis & Reporting

Introduction

Every business faces constant change, and these changes increase the pressure on IT development to deliver successful solutions. Changes to the business make new demands on IT developers to respond quickly to the opportunities or threats to the core business. Solution development is measured in weeks rather than months, which means that the development process must be responsive to the different needs of each project. This approach is known as agile development, of which eXtreme Programming [Beck] is a well-known example.

Changes in technology also increase the demands on the IT community. Many of these changes are improvements of an existing technology that requires a re-evaluation of current systems. Such re-evaluation can result in significant work in upgrading these systems, which can be mitigated by the development of adaptable systems, i.e. systems that have been designed for change.[1] The key to providing adaptable systems is components. Each component delivers a set of services published as clearly defined interfaces, behind which is the encapsulated behavior and information to support the required operations. Components are designed to be highly flexible, i.e. any changes are contained within the boundary of the component without affecting other components. Every attempt is made to minimize any dependency on the development or deployment environments, especially the user interface. The delivered solution is then an assembly of components that can be deployed in a number of combinations.

[1] Older systems generally do not have this characteristic and need to be re-engineered to provide an adaptable solution (see 'Componentization of legacy systems' in Chapter 3)

Chapter 1. Introduction to contemporary(modern) software development

Although contemporary software development is founded upon a solid background of experience within the software industry, it is a relatively immature engineering science. It is our experience that successful processes need to be shared and evolved into best practices if more organizations are to succeed with component- and service-based development (CBD). The first problem is that software development organizations and information services (IS) departments are often in competition, with closely guarded proprietary processes. For example, if an IS manager fails to deliver a software system, one option may be to outsource the development wholesale. If you assume that the in-house and outsourced staff and development tools are equivalent, then the primary difference is the process followed to develop the software system.

In this chapter, we outline the experiences you need to derive real business benefits from using software components. Second, we explain the 'experience trap', which suggests that because components have evolved from object-orientation (OO), many so-called 'experts' take credit for their days in OO. Obviously previous design experience is invaluable, but you must understand that CBD is both a new perspective in software engineering and a tested solution in other engineering disciplines.

Many industries have evolved to use components over a long period of time, including manufacturing, electronics, construction, and automotive [D'Souza and Wills]. Most of you reading this book have been involved with the construction industry: either you live in a building, have made changes to a building, or in some cases been involved in construction. So a 'high-rise construction' scenario will therefore best describe the opportunities and problems inherent in component-based construction.

Step	Construction phases	Software development life cycle phases
1	Analyze business/owner's processes	Model business processes
2	Define business/owner's requirements	Manage requirements
3	Design building	Model system design
4	Hire construction company	Select integrated development environment (IDE)
5	Lay foundations	Build database
6	Construct infrastructure	Build middleware
7	Build façade	Build client software
8	Decorate	Test
9	Occupy	rollout / deploy

10	Remodel foyer	Software maintenance
11	Build extension	System extension
12	Join two office blocks together	Merge systems

Table 1.3. Building software

Componentization

Over the years, the construction industry has realized that it is not practical to build everything from scratch every time. Doors, windows, and bricks are all built off site, often by a different company or team. In most cases, standard construction components will fit the needs of the designer and builder. At other times, a new component will be specified and built to suit the specific needs of a single building. We assert that no industry can become componentized until it has successfully shown how to consume, supply, and manage components.

Consume

The basics of building a house map to the construction of a solution or application. The focus for this 'solution building' process [Allen and Frost] is provided by the customer or the business [Eles and Sims]. In the software engineering world, this is usually achieved by modeling your required business processes [Jacobson]. The contract between the customer and the designer can be achieved by documenting requirements or by reviewing the designs as they progress. Once solution design starts, components begin to have an impact. The group responsible for building solutions is actually consuming components. This highlights our first problem: not only is it important to stockpile the components ready for the construction process, but it is also important to design the solution using information about available components. The Select Perspective solution for the design of interacting components is based on the Unified Modeling Language (UML) and is described in Chapters 5 and 9. However, we believe that UML is even better for designing the internals of black-box components rather than designing static and dynamic relationships between components. For this very reason, we provide extensions to UML for component-based design. The challenge for the solution builder is to assemble a large system from components; to do this successfully, they will need information about the available software components or web services, their required interfaces, and their services/operations/methods.

The process of finding the right components is important to the architect and is usually termed **gap fulfillment** [Bellows]. The gap fulfillment process can have one of two outcomes: **either an ideal (or similar) component is found and can be used, or no relevant component exists.** **This introduces new issues to the stockpiling process, because you may need to create a sub-project to build a specific component**. This means that the stockpiling process is not just about static holding areas for built components but is also an enabling process. The real productivity benefits of CBD will only be realized by enabling parallel solution and component development, and this is achievable only with planning and design. For components that cannot be found, the stockpiling process needs to allow consumers to post a 'wanted' notice for others to respond to.

Once you use a component, whether off-the-shelf or bespoke,[1] you need a process for configuration management. New versions of a component may be provided that the solution builder may want to take advantage of. There needs to be a process to inform component consumers when updated components are added to the stockpile. The solution builder must be able to easily replace a component with a newer version of that component. This can be achieved, for example, if the solution builder is on a mailing list to receive notification when new component versions are stockpiled.

[1] A term used to describe unique, specially developed components.

Table 1.4 shows the application development life cycle with these additional component-based steps. This table is again simplified, but it communicates the component considerations that improve the standard solution-building process. We can take this model one step further by considering the actual deployment of the components on the runtime environment. Typically, the particular computers on which the components run are called 'nodes' [UML 1.x], and mapping these nodes to the components allows us to manage the deployment configuration. This is particularly important when components are widely distributed across organizations and updates occur. To handle this complexity, the process should be enhanced to include deployment recording during the application rollout stage.

Step	Construction phases	Software development life cycle phases
1	Analyze process	Model business processes
2	Define requirements	Manage requirements
3	Design building	Model system design (components)
CBD	**Find windows and doors**	**Gap fulfillment**
CBD	**Subcontract roof joists**	**New component specification**
CBD	**Design in windows**	**Component use**
CBD	**Get on mailing list**	**Get on mailing list**
4	Hire construction company	Select integrated development environment (IDE)
5	Lay foundations	Build database (comp. assembly)
6	Construct framework	Build middleware (comp. assembly)
7	Build façade	Build client software (comp. assembly)
8	Decorate	Test
9	Move in	Rollout
CBD	**New catalog**	**Receive notification, new component**
CBD	**Read catalog**	**Review new component**
CBD	**Update design**	**Update design**
10	Remodel foyer	Maintenance
11	Build extension	Extension
12	Join office blocks together	Merge systems

Table 1.4. Solution life cycle with CBSE

This generally completes the steps in the solution-development process or, as we have called it, component consumption.

Supply

The component supplier is responsible for building completed components, usually to predefined component specifications. In the construction industry, for example, there are window supply companies whose sole source of revenue (and hopefully profit) is window manufacturing. Windows generally come in standard sizes, although builders can request special sizes and materials as described by a comprehensive specification. Roof joists are a good example of specified components that are often distinct to a particular building. Even with bespoke supplies, standards will be used to define basic interfaces. As well as the customer's component specification, there may be industry standards to apply. In the case of the high-rise building, there will be safety standards such as elevator standards, structural standards such as stress loading for girders, emergency standards such as fire exits, and materials standards. These standards map to the ideas of quality, performance, error handling, and technology standards in software components.

Applying similar logic to the discipline of CBD, the component specification may come from the component producer's desire to provide a generic component or the solution builder's specific needs, communicated via the specification. A component specification defines, as a minimum, the interfaces, services, parameters (plus descriptions of all three), required technology type, and business requirements for the component. The specification may also derive from an industry standard for component interaction or a component kit [D'Souza and Wills].

A component supplier starts by identifying the discrete requirements of the component specification. This is an important step in the supplier/consumer process, as the component specification forms the contractual expectations of both parties to the component-development process and final product. This approach is termed 'design by contract' [Meyer].

Starting with this specification and perhaps some component-specific use cases (defined in later chapters), the component supplier iteratively and comprehensively develops a design that satisfies the specification. Typically, the design and development of the internals of the component will be carried out using various techniques and documented using UML and object-oriented (OO) languages, or even plugging together smaller components. Tools for design and code generation may also be used. Testing, debugging, and compilation are generally applied. The important result, for the subject of this chapter, is the completed component.

A component typically has four levels of abstraction, as shown in Table 1.5. The component specification or 'façade' [Gamma et al.] includes the technology-independent definition of the component, as well as information to facilitate gap fulfillment. The minimum requirements for a component specification are defined above, and these form the binding contract between the supplier and consumer. Component management tools can help with this process, as well as ensuring that authority to proceed is granted.

1	**Component specification**
2	**Component implementation**
3	**Component executable**
4	**Component deployment**

Table 1.5. Component abstraction levels

The component specification is the primary reference resource used at design time. It has been our experience that component specifications change during the development life cycle. Therefore, component producers will generally develop multiple versions of a particular component.

A component implementation occurs when you decide on the language to use to develop the component. This defines the 'inside' of the component, with its internal parts and their collaborations. The implementation source code is typically what will be stored in a configuration management or version control system. By 'implementation' source code, we do not mean the formal OO 'implements' statement. We mean all of the source code for the component. Because a component specification can be implemented using more than one language, there is a 'one-to-many' relationship between a specification and an implementation: multiple versions of an implementation will occur during the course of the life cycle.

Finally. the component executable is the real 'pluggable' component or web service used in the assembly of the solution. Each executable/service may result in more than one version, and there may be more than one executable per implementation. This component executable is deployed on a number of nodes to provide services. In turn, all nodes need to be inventoried and tracked so that updates are managed and a stable production environment is maintained.

Accurate information about a component, such as its available interfaces and its services, needs to be published by the supplier. Most important in publishing the component is communicating the specification of the component and its interfaces with a focus on making the physical component available. Updates to software will occur, so new component versions will need to be published in the same manner as the original. As a component consumer, you may expect to receive the latest catalogs and updates, and perhaps even be informed about product recalls.

Building software components is similar to any company that builds windows or doors for the construction industry. A window can be viewed as a black-box component. The high-rise contractor will not need to know the size of each pane of glass or the size of nails used in the windows' construction. The contractor does need to know the outer size (say 3 × 2), the thickness, and information about its functionality. The functionality information is required because the builder's component specification, acquired from the developer of the high-rise building, probably in consultation with an engineer, forms the contract between the developer, the contractor and the subcontractor. Double-or single-glazed, top or side opening, and tinted windows are all examples of functions of a window that do not relate directly to the interface. In the case of a window, the interface is where it touches the brickwork, i.e. the 3 × 2 hole in the wall.

You can see from this short discussion that you need more than just a single interface specification to find and use a component. A degree of functionality and how it is to be achieved also needs to be communicated as part of the component specification.

Manage

The third high-level role in general component-based engineering is component management. Previously, we have referred to a stockpile of components; this describes exactly a typical component library.

A component library is similar to a builder's yard or warehouse; both environments are where construction companies – software or building – order and purchase their components. Component suppliers publish their components in both settings. Builders' yards and warehouses are staging areas between suppliers and consumers; they serve as the 'middle men' for the construction component businesses. In the construction world, there are various levels of interacting component store as in Figure 1.1. The top level is the warehouse of the component supplier. Component suppliers that assemble windows will typically have a central store that is used to supply all the builder's merchants. The builder's merchants will have their own warehouses, which are closer to the construction sites and will typically service many construction companies. In this example, a third level would be the builder's yard for the construction company. This stock will consist of newly acquired components and components available from previous jobs. Therefore, construction companies develop an inventory from previous contract assignments and attempt to use existing stock before they purchase more, and before building it themselves. This technique is typically termed 'reuse before you buy, and before you build'. However, modern construction and manufacturing companies are increasingly moving towards the just-in-time (JIT) approach to component supply. In this approach, the components are moved directly from the supplier to the construction/assembly site with a minimal inventory store. Software components do not suffer from the same capacity issues as there is little or no reproduction cost, although the ideas are similar.

Figure 1.1. Interacting component stores in the construction industry

Warehouse organization is important, as cataloguing helps to manage and locate components. The standard way to support multiple sites and interacting levels of warehouses is usually an integrated set of component repositories or catalogs.

Component suppliers do not simply ship new components to the builder's yard. Effective builders' yards employ a receivable-goods clerk, who assures that the products arriving at the warehouse are the correct items and of acceptable quality. If the components do not meet quality standards, the shipment is rejected. If the components comply with the standards, the clerk certifies the components for component consumers and end-users of the warehouse to purchase the stock.

New components are the other key factor for component librarians, as they may not always arrive fully documented or designed. Generally, new components require extra information to be recorded if the components are to be easily retrieved and used. Software components especially require information, often referred to as meta-data, which includes technology type, features provided, where to get support, examples of use, help

files, and installation procedures.
It is worth clarifying here that components are typically physical items but that in the construction industry pure services may also be available. In fact, you may never buy a crane, but you would typically use crane services over time. For this reason, components and services have been used somewhat interchangeably in this chapter. The differences are clarified in following chapters.

CBD techniques apply equally to small teams and large inter-enterprise CBD endeavors.

Conclusion

The development and maintenance of software components is similar to other engineering endeavors, especially the analysis, design, component warehousing, construction, and maintenance of high-rise buildings. Constructing from components is more complex than building from scratch. The transition from building the same sets of software products repeatedly to componentization demands new and existing roles; it can be seen as requiring a cultural shift. This is not really the case.

1. **Constructing from components is more complex but reduces cost & time.**
2. **Constructing from scratch is more easy but increases cost & time.**

Building systems from components is a natural evolution from existing methods and can always be related to other industries. Car manufacture, electronics, construction, and many other industries' engineering disciplines have adopted componentization because of economics and matured into more productive and profitable industries.
CBD techniques apply equally to small teams and large inter-enterprise CBD endeavors. The rest of this book enlarges this underlying concept and provides the detail of the Select Perspective, which is a set of CBD best practices focused on the software development industry.

Component benefits

For any new approach to be adopted there must be tangible benefits. The benefits of component-based development have evolved over time and fall into a number of categories: technical, business, and economic.

Technical benefits

The technical benefit is 'simplicity through abstraction'. This springs from the underlying nature of a component: its boundary (e.g. the encapsulated functionality, the service contract). A strong boundary encloses complexity. If there are complex functions to perform such as tax calculations, then these can be enclosed within the boundary of a tax component. Component developers focus solely on the tax formulae, rules, etc. Focus gives more benefit than a small scope of interest; it also increases productivity by ignoring irrelevant topics.
Containment of complexity also leads to assembly of solutions rather than full-scale development. Components can be plugged together to assemble complex functionality quickly and may be substituted later if necessary; this is often called '**plug-ability**'.
When you have a clear delineation of a boundary and can specify the services – the component with its **interface** contract – then it becomes possible to distribute the work necessary to construct the component itself. This may be distributed within the organization or outsourced to another reliable partner. Maintenance for the component may then be the responsibility of the supplier – the trusted partner – or may be undertaken by the component consumers themselves.
As the supplier improves the component, say including up-to-date tax rules, the consumers of the component benefit from these upgrades when and if applicable.

Business benefits

Massive parallel development and the capability to outsource development results in faster development cycles – you simply get there quicker. There is more work perhaps, in locating or buying suitable components and services, and certainly more effort in testing and integrating these into a possible solution. But this effort is less than building everything from scratch. Solution delivery becomes more assembly, less development, and new recruits to the team can be productive earlier as they have 'building blocks' of components readily available. Also in the case of our tax component, tax experts will work with the component developers to encode their rare expertise into the final component; we don't all have to become taxation experts!

Economic benefits

In the language of the markets, a commodity is any item that may be freely bought and sold; the key phrase here is 'freely bought and sold'. Components used to be restricted to the confines of an organization. Now there are marketplaces and brokers that buy and sell components and services. These market forces are lowering costs through multiple sales and raising the quality of the supplied component as greater numbers of buyers reuse it within their solutions. As a result, it has become increasingly uneconomic to build a complete system from scratch.

Component-based development

Components have numerous and varied descriptions. Despite this perceived confusion, there is a small set of characteristics that component developers agree upon:

- Components are units of deployment, i.e. they run within computer systems.
- Each component has a published interface that defines its responsibilities as a set of services, e.g. knowing customers' personal details.
- Components interact with other components according to defined communications standards.
- Components are assembled to realize solutions.

Component-based development (CBD) combines the construction and testing of components and services with other new or reused components to form the system solution.

Component suppliers and consumers

We think globally, specification is in Europe, construction is in India, and assembly is in the USA; Select Perspective supports this completely.
—Programme manager, ERP product supplier

All successful economic systems are based on the supplier–consumer model; this supply chain is the engine of economic success (Figure 2.1). We can enjoy that success by applying the same model to delivering software solutions.
Figure 2.1. Supplier–consumer model
Obviously, business solutions are responses to the needs of the business. This is worth restating, because to build solutions without a need is a waste of precious capital and human resources. This solution delivery process is a consumer of components and services. It may construct other software parts, but generally it defines the particular services, such as tax calculations, required for the solution.
The component suppliers have their own delivery process, which responds to requests for services and supplies matching components. It is driven by a need for a particular set of services rather than a business need. To connect suppliers to consumers there is a person, team, or technology that acts as a broker to the parties. This 'broker' facilitates the flow of requests and responses between the different groups. The implication of this model is that the suppliers and consumers are different organizations, groups, teams, etc. This may be so, but they are in fact different roles that can be fulfilled by people within a small team on their particular project. For larger projects, the roles may be filled by other teams, groups, etc. Select Perspective encourages and supports the community of component suppliers and consumers.

Real-world process

Proven in real-world, small- to large-scale IT efforts, Select Perspective has been refined over several years into a collection of highly practical, tightly focused best practices for solution development. It enables IT organizations to take safe, low-risk steps into CBD for both small- and large-scale projects.
Select Perspective has three enduring principles:

1. Small set of key deliverables: e.g. a use case diagram or a class model. If any diagram is not necessary to a particular solution then you do not produce it; anything not on the delivery line has been ruthlessly pruned.
2. Based on experience: experience gained by many consultants working over many years with numerous customer teams to deliver successful solutions; team sizes have ranged from a few developers to large globally distributed teams, all with different skill levels.
3. Designed to fit most organizations and recognizes that each organization is different. It can therefore be adapted to fit existing processes within an organization, or it can be a new foundation from which to start and then adapt as experience grows.

These principles make Select Perspective fit for real-world solution delivery.

Some methodologies tend to revolve around one technique, such as use case modeling or database design. This is simply too limiting for IT organizations that need to build sophisticated applications. To more fully support design activities, Select Perspective seamlessly integrates the three major kinds of visual modeling technique – business process modeling (BPM), Unified Modeling Language (UML), and data modeling. This allows designers to start anywhere and use the right tool for the right job, producing a better result.

Select Perspective development life cycle

The iterative life cycle and parallel working not only delivered the project on time, it gave me and the stakeholders confidence in the team at every stage.

—Project manager, leading fashion retail organization

The Select Perspective software development life cycle is a set of workflows that are based on an iterative and incremental development approach. Iterations allow projects to meet changing requirements and modify remaining activities and deliverables as necessary. Increments deliver functioning and tested slices of the solution, which are treated as project milestones.

In practice, increments are determined from the priorities of the business and/or the removal of any perceived technical risk such as a new middleware platform. Iterations occur both within and between increments. During an increment there will be a number of iterations of deliverables such as component specifications, and screen layouts. Further activities in the increment are then adjusted if necessary. Between increments, there is a re-evaluation of the business and technical priorities.

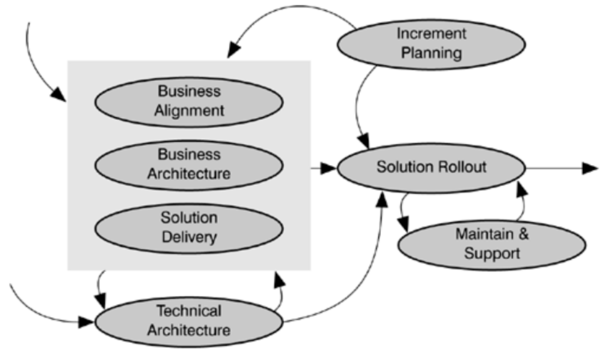
Consume workflow

We were very clear on our roles and what to specify for the developers.

—Business analyst, major insurance organization

The consume workflow (Figure 2.2) delivers the solution then maintains and supports that solution. This workflow contains five major workflows, with two administrative workflows for planning and support.

Figure 2.2. Consume workflow



Activities in business alignment, business architecture, and solution delivery are such that these workflows can start at the same time. For example, business alignment captures the business requirements, which may require the development of some prototype screens to aid user understanding; prototyping user screens is part of solution delivery.

Business alignment delivers the business process models, the use cases describing the functional requirements, and the user acceptance test scripts. Business architecture delivers the component specifications together with their test specifications. Solution delivery consumes the components from the component suppliers; it adds the user interface components and then merges with the technical architecture components to deliver the solution. This solution may or may not be rolled out depending on the delivered functionality, e.g. incremental delivery.

Key to this parallel working is increment planning. This determines which activities within these workflows are to be undertaken for each new increment. Essential, but often forgotten until the last minute, is the technical architecture. The technical architecture workflow delivers base and facility components for the target solution platform, which are merged via Select Perspective Patterns™ with the business components to provide a reliable framework for the solution.

Supply workflow

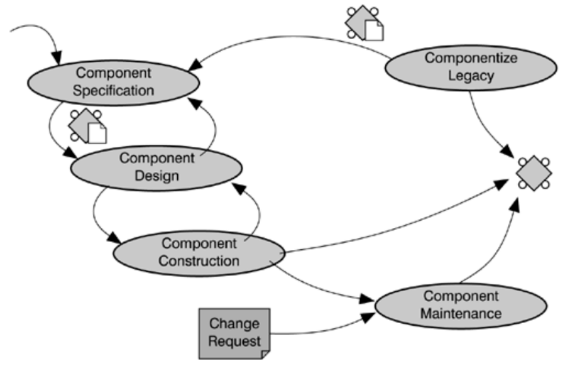
Even though we are remote from the business users, the component specifications made it so easy.

—Systems developer, major insurance organization

The supply workflow (Figure 2.3)

delivers and maintains components. When a request for services is received there is a negotiation on the specification. For example, this may involve splitting some services, or supplying two components, or even altering the service specification to take advantage of existing components. This results in an agreed component specification, which is a form of contract between the component developers and their consumers.

Figure 2.3. Supply Workflow



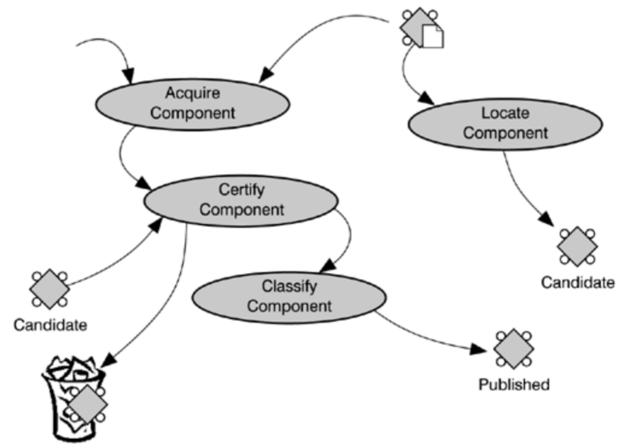
The component is designed, constructed, and fully tested before being delivered to the consumers. Subsequent change requests cause the normal impact analysis, updates, and regression testing activities to be performed before the update is issued.
So far, this workflow description has been about new components, but components can also be extracted from existing legacy systems. The supply workflow assesses the componentization strategy – wrapping, re-engineering, reverse engineering – and tackles the changes, hopefully without disrupting the operational system. Some areas of the legacy system will no longer be suitable, so component specifications, derived from the existing code, need to be drafted and used to construct new components. This typically happens when the existing functionality lacks the performance for a new solution. Select Perspective integrates legacy harvesting into CBD.

Component management workflow

In the component management workflow (Figure 2.4), there are two distinct streams. One stream is concerned with the acquisition, certification, and publication of components and services. The other stream centers on locating and retrieving candidate components for reuse.

Figure 2.4. Manage Workflow

Figure 2.4. Manage Workflow



Components and services can be acquired from a number of sources. First, there has to be a definition of the acquisition strategy, e.g. business components to be built, all others to be bought. Based on this strategy, component specifications can be accumulated and issued to the organization's development teams, or to trusted partners, or sought from commercial sources.
When the candidate component arrives it undergoes formal testing and certification, which means that it can be rejected. If the component is certified, then after classification and storage in a component repository it is published for access by potential reusers.
When solution or component developers have specified the services required for their construction work, they then search the component repository for suitable components and services. Whenever components are discovered they can be retrieved and examined, perhaps even tested, before they are reused. Select Perspective supports all forms of component and service acquisition.

Project types

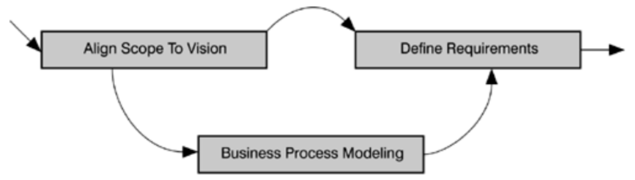
A danger with most process descriptions is that they fit only one kind of project, most commonly a greenfield (empty field on which to build something) development. Select Perspective caters for this kind of project, but it recognizes that there are different kinds of project that demand adjustments to the process workflows:

- Brownfield development projects are treated as part of the normal (greenfield) process on the assumption that there will be legacy systems.
- Web-driven development has extra activities associated with the web content delivery, technical architecture and the deployment of the solution, e.g. to 'web farms'.
- Legacy rejuvenation projects need extra care in key areas to ensure a successful rejuvenation.
- Integrating packaged solutions into the business and systems environments involves changes to the basic process.

Different types of project appear at regular intervals, usually driven by some new technology such as mobile agents. Select Perspective is able to adapt to these new project types.

Business alignment

Figure 5.2. Business alignment workflow



Summary

Business alignment ensures that the requirements of the solution are fully defined. At the hub of these requirements is a use case model supported by the other deliverables in the domain model. The main deliverables from this workflow are:

- domain models
  - business process model
  - use case model
  - business rules catalog
  - non-functional requirements
  - definition of constraints
- user acceptance test plan.

Maintain and support solution

Summary

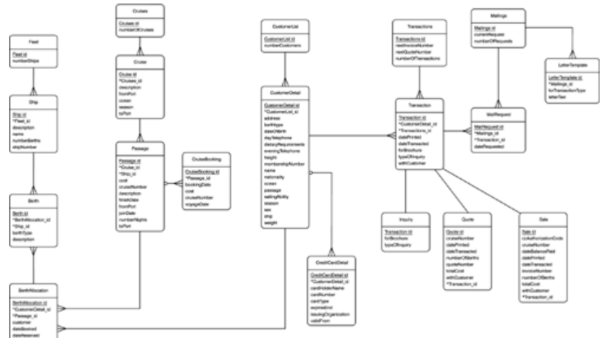
Maintain and support provides resources that ensure the operational system continues to meet the needs of the business and its users by responding to requests for help, changes, and reports of failures. The main deliverables from this workflow are:

- incident log
- change request
- change schedule
- defect log
- risk register.

Design data environments

The content of each data environment should be designed in detail. Design requires a close knowledge of the performance characteristics of the data environment to ensure that the non-functional constraints imposed on the data can be met. For example, while a data model in Third Plus Normal Form is considered an adequate analysis model, the data implementation in a relational database will usually be partially de-normalized to meet the performance requirements. If a non-relational database is used, then the final implementation may bear little resemblance to the logical data model; see Figure 6.3.

Figure 6.3. Example of a logical schema



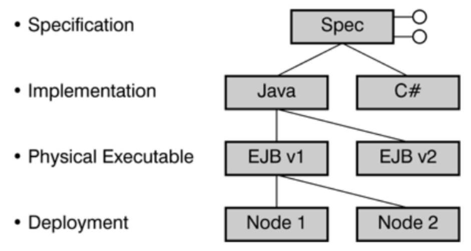
When the **data schema** has been designed, test planning can begin. This plan will define the various tests that need to be performed, e.g. integrity tests, volume tests, loading tests. Much of this can be covered during data migration.

Summary

Data architecture delivery ensures that the organization's informational needs are met in an effective and efficient manner. Organizations deal with information in different ways and at different levels and scales of working: **component**, **solution**, and **organization**. **At the component level**, specific databases may be provided to meet the needs of that individual component. When there are many components in a solution, then a cohesive data architecture and databases are needed to support the different components' information requirements. **Finally, when the solution** is to be integrated with other solutions, then the combined requirements of the data architectures and databases means that organizational level has been reached.
In all cases, the presence of existing data schemata and databases may place constraints and restrictions on the design of data architecture and services. Data migration planning then become necessary to ensure that the new solutions maintain information integrity and reliability.

To model the design, consider the component as a micro-system where the **interface services** are the **use cases**.

Maintenance ensures that not only classification standards are met but also, more importantly, that configuration control policies are in place and adhered to. To achieve this, both the component specification and the actual component need version identification. New versions of components and services can be created and linked to older versions. It will also be necessary to archive unused and older versions of components and services. This concept is similar to that of source code configuration management, but it refers to component specifications and completed components; see Figure 4.2. The component specification exists well before any code is created, and this is one of the reasons why mainstream configuration management systems are currently not suited to component management.



q
Publication and republication of components requires that consumers (re-users) be notified of the availability of software components. This task may be automated with technologies such as e-mail and web-based catalogs. New components and services will typically be advertised, and e-mails could contain a copy of (or pointer to) the new component.

Establishing service-level agreements with suppliers is critical for ongoing maintenance of the component repository. The service levels are very much dependent on the type of supplier and the associated risk. For example, a component supplied in-house with source code represents less risk than an externally supplied component with no source code agreement. Service levels should also enable policy definition regarding the responsibility for maintaining the component. If a **component** is supplied in **binary form (black-box reuse),** then maintenance lies with the supplier. When a component is supplied in **source form (white-box reuse)**, either by an outsourced contract or internally constructed, then maintenance usually lies with the supplier. The danger with source code availability is that reusers may affect the code themselves, leading to project-specific clones of components. This underlines the need for service levels and the appropriate policies to be put in place.

Making the supply decisions

Supply decisions can be made once you have defined your units of functionality then identified and allocated these to an architectural layer.[2] These decisions consider whether each functional unit should be a component or a web service and whether they can be **reused/rented/bought/built (R/R/B/B)**. For most applications, the considerations in Table 3.2 hold true.

Briefly, these architectural layers consist of:

- **low-level infrastructure**, including the hardware, operating systems, and virtual platforms such as Enterprise Java and .NET;
- **web services applied as part of the technical infrastructure**; examples include authentication, long-running transactions, and management;
- **web services applied at the business level**; examples range from relatively fine-grained services such as currency conversion and payment processing, through to coarser-grained services such as credit checks and hosted ERP functionality;
- **the application's business logic**;
- **the application's user interface**.

Components identified within the application reside in one of these layers.

The central issue in the use of services as opposed to components is that of provenance. In addition to its functionality, certain quality and cost judgments must be made about a service that are not necessarily required for a component. For example:

- Will the service provider or the specific service still be here next year, i.e. for the anticipated lifetime of the application? This is much more important in terms of rented services than it is for bought components. Should a component provider go out of business, the component will not stop working. However, it is quite a different matter to run a service yourself.

- What is the cost model of the service? There may be a purchase or licensing cost, as there would be for a component. Many services will charge a per-transaction cost as well as or instead of an up-front charge. There may be a significant cost associated with the development and testing of the service if a service charges per transaction.
- Use of a service may increase the cost of the application over the life cycle. Any subscription-based service will need the subscription renewing every year as well as an ongoing service or per-transaction charge.
- Does the service provide the appropriate level of systemic quality? In some cases such as availability, the required level can be gained through the use of multiple services.[3] Other qualities such as security, require a minimum level since a breach in security of the service may lead to a breach of the application security as a whole.

  [3] You could gain 99.99% availability through having a certain number of 99.9% availability services you can use at any one time.
- If the service is accessed across the public internet, what impact will this have in terms of scalability and availability? All parts of the path between your organization and the target web service must be assessed for the impact of traffic peaks and service outages. For example, what sort of bandwidth and availability can your internet service provider (ISP) guarantee for your internet connection?

Layer	**R/R/B/B decision**	**Rationale**
Low-level infrastructure	Buy	Low-level infrastructure, as defined previously in the application architecture, is very much a commodity. It will be bought in as a component (hardware, operating system, and virtual platform) and installed. Although this could be delegated to an ASP, essentially this level consists of bought-in components (not web services).
Infrastructure web services	Rent or Buy	Web services at the infrastructure level will often be bought in the same way that components at the same functional level will be bought. For some web services, this may actually involve buying in the software and hosting it as an internal web service. Frequently, however, the web service will be rented from a third party and accessed remotely. This level of web service will rarely be built in-house, hence there is also very low possibility for reuse. However, such services may be built in-house during the initial days of third-party web service provision.
Business web services	Build, Reuse, Rent or Buy	Web services at the business level are subject to many of the same considerations as those at the infrastructure level. Depending on the application, there may be specific requirements for business-level web services that are not addressed by third parties. However, these services may not be unique to the business logic for the specific application being built; an example may be actuarial calculations in an insurance firm. In this case, a web service may be built and operated for this application, with the possibility of subsequent reuse in later applications.
Business logic	Build or Reuse	The business logic is such an integral part of the application that it will not be bought or rented from a third party. Business logic available inside the organization may be reused if appropriate. This may take the form of either components or web services.
User interface	Build or Buy	The user interface is such an integral part of the application that it will not be bought or rented from a third party. Since the user interface defines the precise intent of the application, it will not usually be possible to reuse existing components or web services.

Table 3.2. Choosing components or services and how to get them