Chapter 13 Process Overview

This chapter provides an overview of the Catalysis development process, describing how development artifacts can be structured, developed, and evolved on a typical project. The case study used in the next four chapters illustrates specific application of this process to a sample problem but does not illustrate all nuances of a complete development.

Section 13.1 provides a short recap of modeling, designing, and implementing with objects, from business to code and specification to deployment.

Section 13.2 is a general introduction to the process. It covers various routes through the method, its parallel and iterative nature, the continuous attention to QA and testing, and its early emphasis on architecture (both static dependencies and dynamic aspects).

Section 13.3 outlines how a typical project might evolve over time and explains how the development of various artifacts may overlap.

Section 13.4 describes a typical structure of packages for a project. This structure shows up in project planning, certain elements of system architecture, and the structure of documentation.

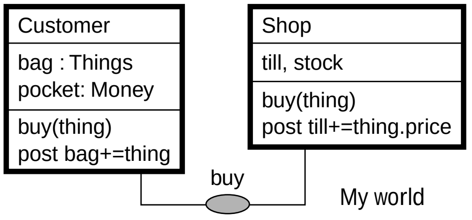
Section 13.5 introduces a set of process patterns that are elaborated on throughout the case study. These patterns describe some of the broad contexts for development and a reasonable strategy for each one. *Pattern 13.1, Object Development from Scratch,* outlines an approach for developing a system from scratch. *Pattern 13.2, Reengineering,* addresses the case when an existing design is being reworked. Pattern *13.3, Short-Cycle Development,* motivates development in short incremental cycles as a useful basis for many projects.

13.1 *Model, Design, Implement, and Test—Recursively*

The process of modeling and designing is recursive throughout business, component specification, and internal design. Similarly, specification and implementation activities are also recursive across the business or domain model, component spec, and internal design.

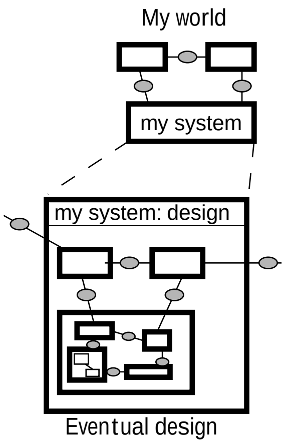
13.1.1 Models and Designs: Business to Code

A model of real-world objects and their interactions—or rather, some users' understanding of them—is called a *business model*. The outcome of each interaction depends on (1) the types to which its participant objects belong and (2) the states they are in at the time. For any participant types, you can describe the effects of an action by relating the values of the attributes before and after any occurrence of that action.

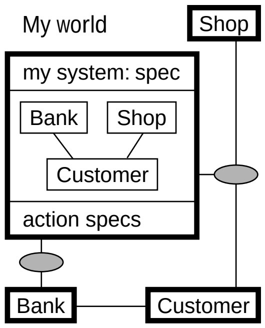


Components of a designed system can be described in the same way: as interacting objects. The same notation can be used for designing interacting software objects as is used for analyzing things and concepts in the users' world.

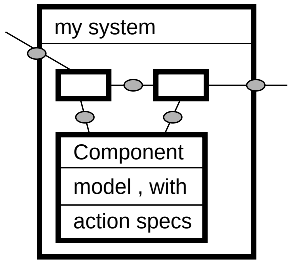
Each component is then designed as a collaboration of interacting objects in the same way, and so on recursively until we get to programming-language primitives or (preferably) an existing component. The same techniques are used to design the internal structure of a piece of software as are used to describe or design an organization or machine, which may or may not use software. Although it is not necessary or practical to produce the layers in top-down order, there is a notional hierarchy of designs. In the topmost layer, the complete system consists of a few interacting objects.



A useful principle of object-oriented specification (and design) is that the structure of a software system should be based on a well-chosen model of the world with which it deals; that approach makes the design easier to update with business changes. Each real-world object (whether physical or more abstract: orders, meetings, menus) has its counterpart inside the system. A first step in building an object-oriented program from scratch is therefore to make a business model and then declare it to be the first draft of a type model of the software. The objects and their relationships are used as the type model of the system.

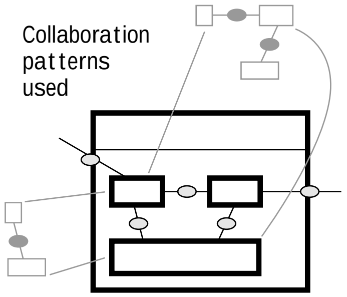


The attributes of a complex object are usually expressed in terms of types, which themselves are modeled with other attributes and so on. Drawing them pictorially shows the relationships more clearly. When an object is specified, the types of attributes are chosen for their expressiveness rather than for execution efficiency. Some of these "specification" types invented to help model a set of actions may never be implemented directly; however, the type model must constitute a valid abstraction of the implementation, as documented in a refinement and its justification. The types that are implemented are those generated by decomposition: they must exist because the design calls for them to interact with one another.



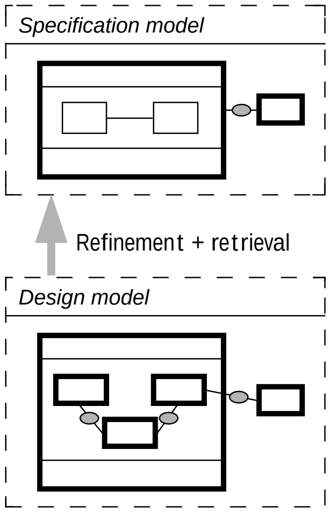
Intermediate step

The behavior that each object provides to the others is specified, and the specification is kept separate from its internal collaboration design and code. Thus, each internal designed component has its own type model, with some refinement that maps it to the type model of the containing component. This arrangement allows us to define a more general architecture, with a variety of different components able to plug in to the basic design. Some components may be bought, and others may be purpose-built. The idea of specifying the interfaces well is crucial for component-based development.



The design of any system embodies several partial designs of how the pieces collaborate. These designs constitute some of the key architectural elements used in the system design. The resulting design is a composition of these partial collaborations, unified via objects that participate in more than one collaboration. The architecture is often best described and understood in terms of recurring patterns of such collaborations.

An important principle of this recursive decomposition is that, at each level, the components can be designed independently if the specifications are accurate enough.



Accurate specification of the actions depends on careful models. When all the components are designed by the same team, we can afford to be relaxed about this: If the spec of a component you use is unclear, you can walk down the hall to whoever is designing it. But in a world that takes component-based design seriously, you won't know the designers of many of the parts you use, nor many of your own clients. More effort must be put into specification, with the understanding that the economies of CBD will pay it back.

The extent to which the code reflects the business model is related to the architecture. If the code must be changed every few days to keep competitive with others (for example, financial dealing systems), the requirements of users should translate directly to code changes, which must closely reflect a suitably flexible business model. If high performance is needed and changes are rare (as in an undersea multiplexor), optimizations must be done. The analysis and design models are kept separate. If the refinements are localized and well documented, the benefits of traceability are preserved. Refinements fall into a small number of categories and useful patterns that are documented in standard ways (see Chapter 6, Abstraction, Refinement, and Testing).

13.1.2 Specify, Document, Implement, and Test: Business to Code

For every specification activity there is a corresponding activity that deals with implementation and testing; every refinement claim has a corresponding justification and test (see Figure 13.1). At the level of a single interface, the implementation may be a single class that is tested against operation specs. For a collaboration, several classes or components are implemented and integrated, and then their interactions are tested against the collaboration specification. At the level of the system specification, all external operation and abstract action specifications are tested. At the level of the business or domain model, to implement means to deploy the "to-be" model, conducting training and other more conventional forms of upgrades, and acceptance testing.

Documentation is structured around specifications and implementation and their refinement and import relationships. A user manual—a description of how a user accomplishes tasks by using the system(s)—is a particular form of documentation associated with a refinement: how the abstract business model and actions are realized by more-detailed actions performed by the users and systems. Test specifications are also associated with refinement relationships. The rules for system architecture are documented in a package that specifies the patterns and frameworks that will be used in other packages that import it.

13.2 *General Notes on the Process*

Before we jump into specifics of the case study, let's review some general notes on how the method is applied.

13.2.1 Multiple Routes through the Method

There are many possible routes through the method, each with a different prototypical sequence of tasks and deliverables that is better suited to certain project characteristics; one route may omit activities and deliverables that another one includes. Different routes can be used or combined for any project or subproject. A lightweight route would be a good way to get started with Catalysis.

For example, for a project in which requirements should be specified early and accurately, we might follow the "Build" route in Figure 13.2; the case study in this section of the book roughly follows this route.

1. Build1. Build a domain model to capture terms, domain rules, business tasks.
2. Build2–3. Refine it to specify the system by building its local type model of the domain; the retrieval mapping is defined top-down.
3. Build4–5. Partition and refine it to build the internal design model; again, the retrieval mapping is defined by forward-engineering the system type model and distributing it across the design components.

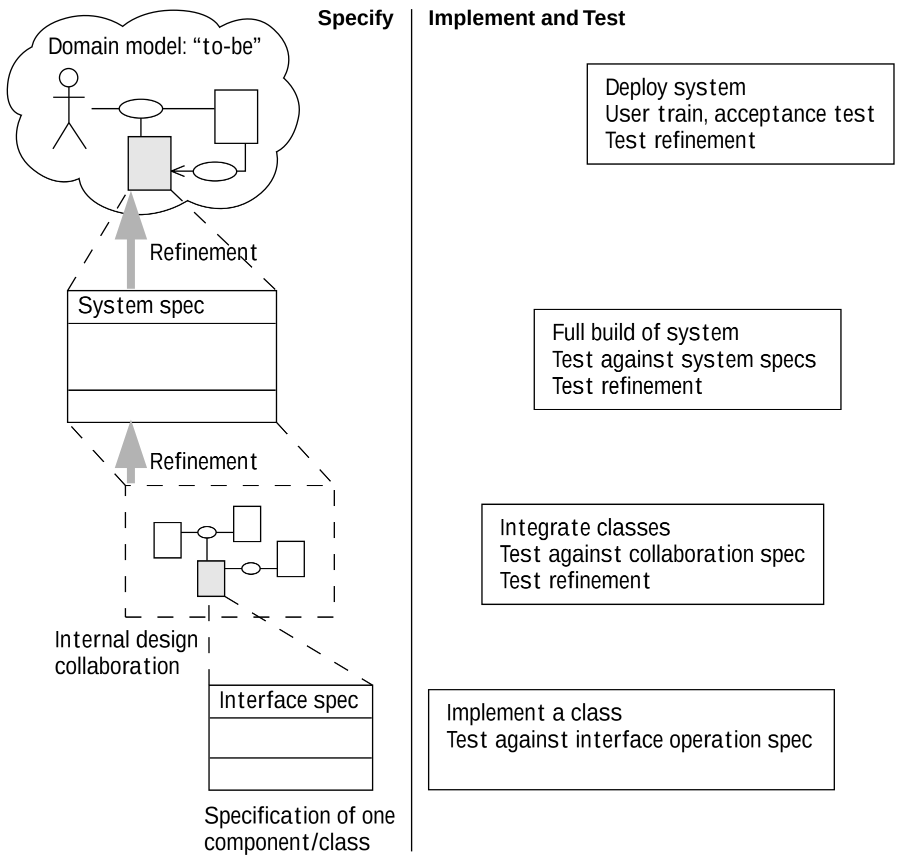


Figure 13.1 Specify and implement: business to code.

In contrast, if the system must be built from many heterogenous components and if the requirements can be shaped significantly by the ease of assembling those components, we can follow an "Assemble" route like this:

1. Assy1. Start with a domain model—optionally, a rudimentary system spec.
2. Assy2. Build type models of each component, reverse-engineering if necessary.
3. Assy3. Define the retrieval mappings between type models of individual components and the domain model or system type models.
4. Assy4. Define an achievable type model and behavior spec for the system.
5. Assy5. Refine the domain, defining how external components and users interact with this achievable system to accomplish the original tasks.

See Section 10.11, Heterogenous Components, for a complete example of how this might be done in practice.

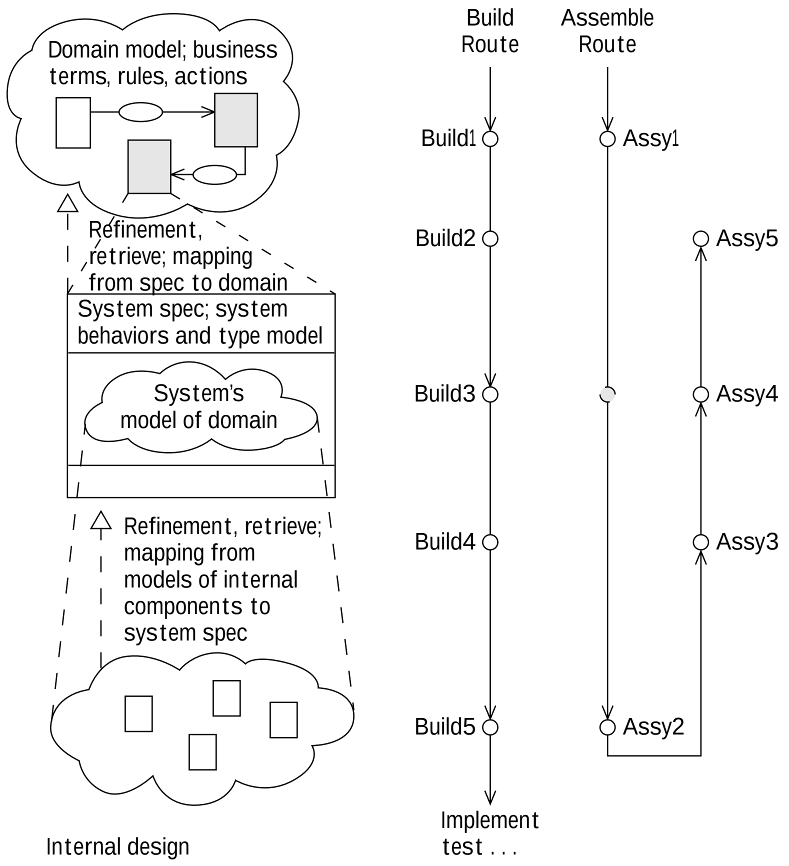
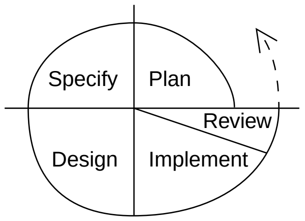

Domain model; business terms, rules, actions
Refinement, retrieve; mapping from spec to domain
System spec; system behaviors and type model
System's model of domain
Refinement, retrieve; mapping from models of internal components to system spec
Internal design
Build Route
Assemble Route
Build1
Build2
Build3
Build4
Build5
Implement test . . .
Assy1
Assy2
Assy3
Assy4
Assy5

Figure 13.2 Many routes through a method.

The relationships between all the models and documents are clearly defined in Catalysis and can be clearly documented regardless of the sequence in which they are built. When one model or document is changed, those that are related to it may also have to change and must be reviewed.

13.2.2 The Process Is Nonlinear, Iterative, and Parallel

Although they are shown in these chapters in a linear sequence, these documents are almost never produced in sequential steps. There will be a gradual shift in focus from the "earlier" aspects to the "later" ones, but the creative process and prior constraints may in general develop the various parts in any order. Good management should impose discipline, not by forcing a linear sequence but rather by seeing the process as one of successive enhancement. Appropriate milestones, corresponding to multiple iterations, are the completion of *vertical slices* of functionality visible to the end user or *horizontal* services, components, and end-to-end implementations of the technical architecture that will be used for end-user functions.



The *spiral* model works well in any context in which the current plan must depend on the outcome of earlier work expenditure increases along the spiral; each cycle includes a review of results and risks, and it drives a refinement of the goals and plans of the next cycle. An early spiral typically covers a much broader area in requirements and a much narrower piece of design; the situation reverses in later cycles. Lessons learned from any spiral feed back into the most abstract level of models that are affected. There are many colorful names for variants of this basic idea, including fountain, tidepool, and tornado.

The design of the spiral cycles is usually driven by an understanding of project risk, with high-risk items being tackled early. Some of the highest risks in a project come from unclear requirements,[1] so they should be tackled early. For large projects, the technical architecture—with all the infrastructure needed for communication, transactions, messaging, and systems management—also poses a serious risk. To help reduce this risk, practice explicit and early architectural modeling, evaluation, and implementation, and build on standard infrastructures. Other risk factors include team communication and dynamics, the age and maturity of technologies, management commitment, and project funding.

Cycles can be overlapped and carried on concurrently, because many activities can proceed in parallel with low risk of rework; these activities should not be needlessly serialized. Catalysis provides the appropriate consistency rules between artifacts even if they are developed in parallel. However, in some situations the best approach is for a small group of people to resolve critical issues, often at the requirements or architecture level. It is a mistake, simply to keep some developers from twiddling their thumbs, to force concurrent development if it is dependent on these issues being resolved.

Iterations and increments involve development cycles but play different roles. An *iteration* aims to improve the design of existing work; iteration is fundamentally about rework. In addition to dedicated iteration cycles, each development spiral might have a *consolidation* stage at the end, for refactoring the design and making it more robust for downstream work. In contrast, an *incremental* cycle adds new functionality to what already exists: either a new end-user function or supporting functions that will be used toward that end.

Iterations and increments must have clear objectives or else they can become a euphemism for structured hacking. Increments should be planned at multiple levels, from end-user increments to those that are internal to the development team or visible and demonstrated up to the project manager. Packages and their import relations are used to plan iterations and increments.

1. Many requirements "changes" are actually the result of initial lack of precision.

13.2.3 Rigor, QA, and Testing Are Continuous

Quality assurance is not an after-the-fact activity in Catalysis. From the onset of requirements, the method is focused on ensuring quality in intermediate deliverables and documentation and ensuring a quality final delivered product.

Pre- and postconditions, invariants, and refinements offer "on-demand" rigor. Experience shows that writing in a more precise notation flushes out ambiguities and questions that would otherwise lie dormant until coding and testing. Even after there is code, an abstract model can help make a clear picture, whereas design pragmatics might obscure the big picture with complications. This kind of rigor saves time later, so the benefit is long-term rather than short-term. Using the formalisms is a skill akin to programming; some aspects may be a bit less familiar to some people, but no more difficult.

Adjust the degree of formalism to the life expectancy of your product. Components that will be reused a lot justify (and need) more investment in getting things right up front. "Quick and dirty" developments can be hacked to appear to work, but bear in mind that these solutions tend to end up being the bedrock! Cool-headed management is required. In general, you should adjust the degree of formalism within the development. Always use informal descriptions, but use greater formalism for crucial issues. Interleave narrative descriptions with formal diagrams and specifications.

For example, the GUI of a typical development might be documented only with storyboards and GUI mock-ups that are annotated with explanatory notes, with an optional mapping to the type model. Other design discussions, and specification of business behavior, use proper action specifications. Exceptions may be done more or less formally, depending on project needs.

All behavior specifications in Catalysis are precise enough to be used for testing, both at the unit level (type specifications and class refinements) and at the level of integration (collaborations and action refinements). In particular, specifications written in terms of abstract attributes become testable against the implementation by virtue of the *abstraction functions* that are part of the refinement; user tasks are specified as abstract actions and are documented and tested as sequences of refined actions (yes, the user manual starts early!). Static invariants can be tested at any time the system state is stable. More-general constructs, such as effects and effect invariants, also map cleanly to test specifications. In general, any claim of refinement between a concrete and an abstract model has a corresponding strategy for testing (see Chapter 6, Abstraction, Refinement, and Testing).

Catalysis models have clear semantic relationships to one another. At any level of abstraction, they form an important part of the inspection criteria for those models. Across levels of refinement, these rules, together with the rules for refinement, provide a concrete basis for design reviews.

The impact of change in Catalysis is clearly defined. A change at any level of description must be propagated up to the highest level at which the change has an impact. Specifically, the change at a concrete level need not be propagated to an abstract level if the refinement mapping can be updated so that the abstract specs are still valid.

Packages provide the unit of configuration management and release control. Catalysis packages are flexible, because one can model different aspects of the same type or action in two different packages. A versioned package is frozen; every package it imports is also versioned.

Safety-critical projects can further exploit the precision available with Catalysis, using advanced facilities (see Section 3.5.2, Redundant Specifications Can Be Useful) to formally check important properties that the design should exhibit.

13.2.4 Emphasis on Architecture

There are at least two interesting aspects to architecture: (1) static dependencies between units of work and (2) the runtime patterns of component and object structure and interactions.

The structure of packages defines one aspect of architecture: the static dependencies between units of development work, whether business models, interfaces, or implementations. The documentation and code structure reflects the package structure. Documentation, always combining informal with rigorous descriptions, is part of a package; within a document you can refer only to model elements visible to that package. Packages also provide the unit for change management and upgrades.

Using refinement, you can model objects and interactions at all levels of granularity. This arrangement provides a “fractal” view of architecture, from the business roles and processes to large-grained interacting architectural components including a “system” to individual interfaces and classes. Any refinement has associated architectural decisions.

In addition, good use of packages facilitates full separation of interfaces from implementations. As an extreme example, a single class might be implemented in a package that imports the packages with the type definitions of all types that class must implement; each such package contains the minimal model of any other types that it must interact with.

Therefore, you should structure the macroscopic view of the development around type models, frameworks, packages, and refinements.

Collaborations, as partial definitions of object roles and types and their interactions, provide support for a “pattern” view of the dynamic aspects of an architecture. A design is often best understood as a composition of such patterns onto the objects involved rather than in terms of individual objects or interactions.

The basic constructs of type, collaboration, and refinement support all levels of specification, architecture, and implementation. However, we also pay explicit attention to specific levels of architectural design: logical and physical database mapping, technical architecture (including client-server and multitier peer-to-peer architectures), and user-interface modeling. The case study touches only on some of these aspects.

13.2.5 Unambiguous Notation

The notation used is based on that of the Unified Modeling Language (UML 1.1). What we add is a systematic way to use this language, a way to establish and maintain the relationships between the documents, and a clear semantics for abstract models.

Much of the notation is useful throughout different stages of the process. For example, we use the same tools to describe the interactions between people and a machine we propose to build, to describe the interactions between objects collaborating inside the machine once designed, and to describe business tasks and processes. For this reason, many of the notational tools are introduced early in the case study and are then reapplied in each phase.

The notation here is not limited to complete documents delivered within a standard development process. Designers sketch these diagrams on whiteboards when they are discussing their designs with their colleagues. In short, this is a specialized language for communication models and designs.

Some informal and ad hoc notations are always useful as long as you recognize that they are informal; they should sometimes be cast into a more precise form as their purpose becomes clear. Useful new formal notations also will no doubt be invented; their semantics should be described clearly using frameworks, as illustrated in Section 9.8.3, Examples of Semantic Rules for a Dialect.

It is our experience that familiarity with the toolbag demonstrated here makes such discussions much less ambiguous and far more productive than they are when the only tools for debate are ad hoc pictures and natural language. And a single designer’s own thoughts are clarified when cast into the forms shown here. This is often the greatest benefit of taking up a more rigorous notation.

So although this study demonstrates (parts of) an analysis-to-code document structure, there are other equally valuable ways to use the tools in the bag, including completely different routes (see Section 13.2.1, Multiple Routes through the Method).

13.2.6 Typical Process for Business Systems

A typical large business system has human users and a back-end database. The development process for these systems still goes through the levels outlined earlier, with some specialized activities required within the levels. Figure 13.3 outlines these activities and shows how they map to the three essential levels we discussed; the corresponding implementation and test activities are not shown.

There are still three conceptual levels: the domain models (the outside, describing the environment in which the software will reside), the component specifications (the boundary, describing its externally desired behaviors), and the component implementation (the inside, describing its internal design). These three levels continue recursively: Each subcomponent itself has a context (the collaboration with others that should realize the external spec), a specification, and its own implementation.

In traditional terms, requirements and analysis are mostly concerned with the outside and boundary, from identifying and understanding the problem to specifying each externally visible component of an envisioned solution; design focuses on internal structure and architec-

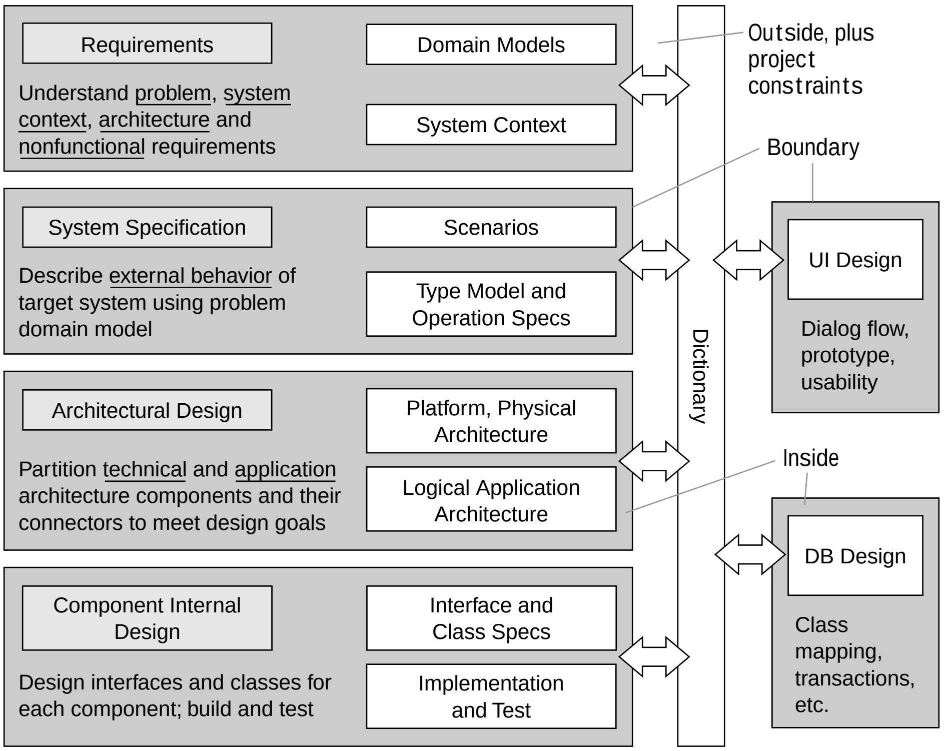

Requirements
Understand problem, system context, architecture and nonfunctional requirements
Domain Models
System Context
System Specification
Describe external behavior of target system using problem domain model
Scenarios
Type Model and Operation Specs
Architectural Design
Partition technical and application architecture components and their connectors to meet design goals
Platform, Physical Architecture
Logical Application Architecture
Component Internal Design
Design interfaces and classes for each component; build and test
Interface and Class Specs
Implementation and Test
Dictionary
Outside, plus project constraints
Boundary
UI Design
Dialog flow, prototype, usability
Inside
DB Design
Class mapping, transactions, etc.

Figure 13.3 Main activities for a typical business system.

ture. However, externally visible decisions often have some design flavor; think of these as "external" design or "business" design.

13.2.6.1 Requirements: Spanning Outside, Boundary, and Inside

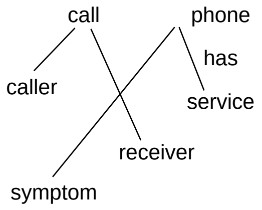
The requirements activity is aimed at understanding the problem and how the proposed solution will address it. The primary deliverables are as follows:

- *A business model:* Collaborations, types, and glossary (possibly an as-is model and a to-be version that includes the envisaged systems)
- *Functional requirements on the system:* Usually in the form of a system context diagram with use cases and scenarios
- *Nonfunctional requirements:* Performance, reliability, scalability, and reuse goals

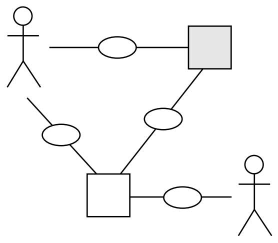
- *Known platform or architectural constraints:* Machines, operating systems, distribution, middleware, legacy systems, and interoperability requirements, all captured by a package structure and collaborations
- *Project and planning constraints:* Budgets, schedules, staff, and user involvement

Specific techniques and constructs used to describe the business model and requirements include the following.

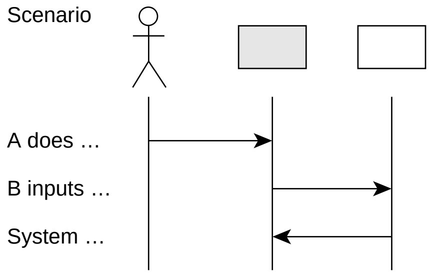
- *Storyboards:* Sketches of different situations and scenarios in the problem domain, possibly using a domain-specific notation.
- *Concept maps:* An informal but structured representation of related terms in the domain. The notation, which is similar to a mind map**,** is simply concepts or phrases with labeled directed lines between them indicating a relationship; it can include rich pictures or storyboards of the domain.



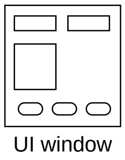
- *Business collaboration:* Identifies the actors in the domain and their interactions (the actions, or use cases, and information exchanged). The actors typically represent the roles of people (such as buyer) or software systems (such as inventory system). The as-is and to-be versions form the basis of deployment and transition plans for the systems.

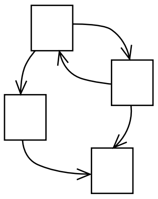


- *System context:* Acollaboration centered on the target software system, intended to clearly define the boundary of "the system." The use cases in which the system is an actor are those that must be developed as part of the system.
- *Scenarios:* Aprototypical sequence of interactions in a business collaboration or the system context. These scenarios can be used to formalize specifications of the actions, even at the business level.

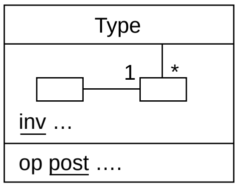


13.2.6.2 System Specification: The Boundary





System specification proceeds much as before but has an added element of user-interface design. The normal artifacts of system specification—a type model and operation specs—are now accompanied by prototypes and UI specs describing the screens, dialog flows across windows, information presented and required, and reports. These user-interface elements are kept consistent with the type model and are reviewed through scenarios.

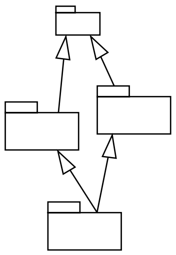


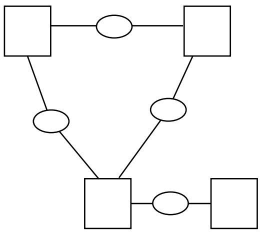
The primary deliverable is a *type specification*: the system being developed specified as a *type*, with a *type model* (attributes and associations) and a set of *operations* specified against that type model. Defining the type involves identifying each action in which the system participates and specifying it as an operation on the type. The behavior can be described with state charts in addition to operation specifications and can be exercised with scenarios and snapshots.

The specification can be split across subject areas—broad areas of usage or function that help partition the system behavior—so that one area can be analyzed somewhat separately from the others. Packages can be used to structure all work on a large system across multple vertical views or horizontal layers.

13.2.6.3 Architectural Design: The Inside

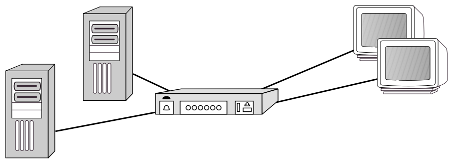
The internal implementation of the system is split into two related parts: the application architecture and the technical architecture. The main deliverables are described on the next page.





•*The application architecture:* A package structure and collaborations. This implements the business logic itself as a collection of collaborating components, with the original specification types now split across different components. The components can range from custom-built to common off-the shelf components, such as spreadsheets, calendars, and contact managers, to purchased domain-specific components such as factory-floor schedulers. This application architecture lives “atop,” and uses, the technical architecture.

- *The technical architecture:* Apackage structure (for static dependencies) and collaborations (across technology components, such as UI, business object servers, and databases). These cover all domain-independent parts of the system: hardware and software platforms; infrastructure components such as middleware and databases; utilities for logging, exceptions, start-up, and shutdown; design standards and tools; and the choice of component architecture, such as JavaBeans or COM. It also includes the design rules and patterns that will be used in the implementation. The technical architecture is designed and implemented early and is evaluated against nonfunctional requirements such as throughput and response time.


Hardware architectures and distribution

- *Database architecture:* The design of the database portion should start at this stage and includes mapping of the design object model to the database, definition of transaction boundaries, and so on. Depending on the choice of database and supporting tools, this activity may or may not take significant effort. Database performance modeling and tuning usually take some effort.

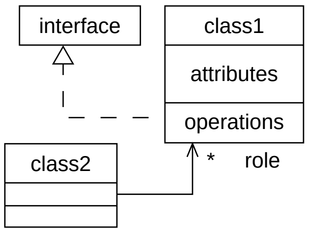
13.2.6.4 Component Implementation: The Inside

Individual application components are designed and built down to the level of programming language interfaces and classes or preexisting components, or to a point where the implementation can be mechanically generated from the detailed design.

The goal of component implementation is to define an internal structure and interactions that satisfy the behavioral, technological, nonfunctional, and software engineering requirements for a component. In Catalysis the component specification (type) mentioned earlier identifies the behavioral requirements.

In simple cases, you can start with the assumption that every type identified in the specification model will be implemented directly as a separate class; for large components you may go through a recursive step of subcomponent partition and specification. We determine intended responsibilities of each class and then build interaction diagrams to design their interactions to realize the specified behaviors.

We can now define the design's class model: the classes that compose the system, the interfaces they implement and use, their attributes and operations, and the references between them. The initial class model is derived by reifying each model type to a class, adding the operations that the class must implement from the interaction diagram, and designing the attributes and directed associations that the class needs to implement those operations.



This design can then be refactored. In this activity, you separate interfaces on a class for different clients, migrate behaviors to abstract superclasses or collaborating classes, split data and behavior of one class into two, and refactor packages to separate architectural layers.

13.2.7 Modeling the Catalysis Process

Many aspects of the Catalysis process can be modeled using the concepts of type, collaboration, and refinement. When you're modeling the development process itself, the interesting development objects and actions include the following:

- Objects
  - *Packages:* Get populated, documented, import other packages
  - *Model types, actions, refinements, etc.:* Get specified, implemented, tested
  - *Other deliverable artifacts:* Documents, source code, tests, executables
  - *Developers:* Create and modify artifacts, get trained
  - *Development tools:* Evaluate, purchase, install, use to create artifacts
  - *Project and iteration plans:* Get reviewed, revised
  - *Deployment plan:* From as-is to to-be via a transition plan, including installation scripts, upgrade and migration tools, user documents, user training
  - *Paths and routes:* Different sequences of activities for various project types
  - *Project, subproject, activity, task, cycle:* Things to be done following some route
- Actions
  - *All development activities:* Actions by developers using tools on artifacts
  - *All planning activities:* Planning actions by developers
- Refinements
  - Detailing of development activities, artifacts, plans, and schedules

Although the rigor supported by the method may not be fully utilized in this context, its facilities for types, refinement, and frameworks become more interesting when the process and project management needs are supported by tools. We will not expand further on this point here but mention it for completeness.

13.3 *Typical Project Evolution*

A project produces a subset of models and diagrams that describe the design. For some projects, the order in which these are produced will be mostly top-down; for others, more bottom-up. In almost all cases there are multiple development tasks that can proceed in parallel depending on project resources and constraints. In all cases, the relationships among the artifacts are the same, and the most important initial methodology question for any project to answer is

"Which of these decisions will I explicitly document? And when and how?"

Figure 13.4 illustrates a typical project evolution for a large business project; it shows the activities that take place across time (horizontal, not to scale) and different levels of development (vertical). Numbered items in the figure are discussed next. The circles indicate interesting points in time, perhaps with an associated artifact. The partitioning of work across packages, the degree of opportunistic development, and the specific path through the method vary across projects. The figure does not show iterations or increments explicitly; all deliverables are subject to revision or refinement based on downstream work within the rules established for configuration management.

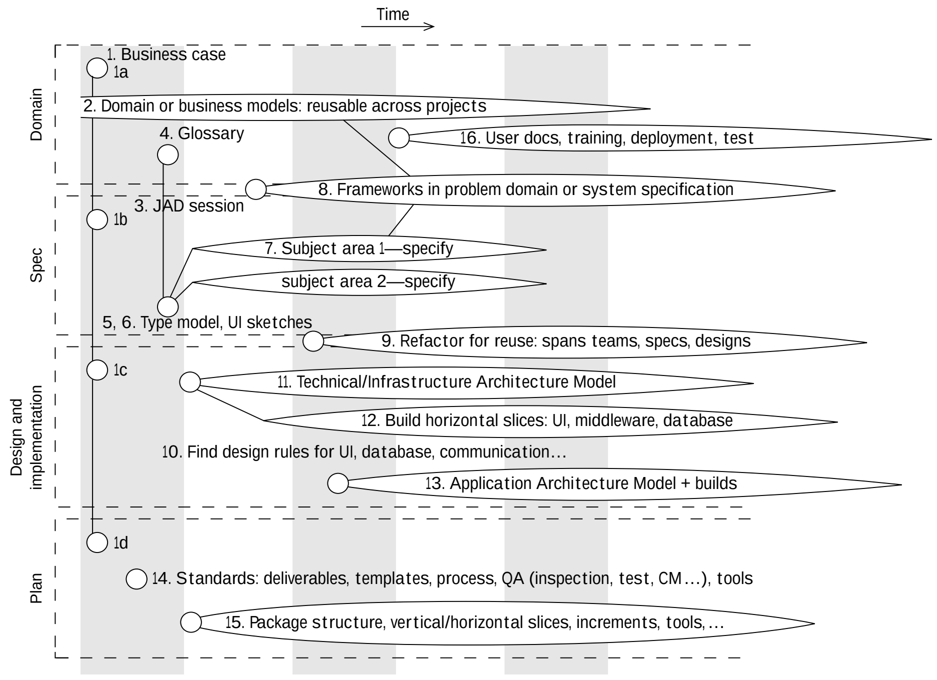

Time
Domain
Spec
Design and implementation
Plan
1. Business case
1a
2. Domain or business models: reusable across projects
4. Glossary
16. User docs, training, deployment, test
8. Frameworks in problem domain or system specification
3. JAD session
1b
7. Subject area 1—specify
subject area 2—specify
5, 6. Type model, UI sketches
9. Refactor for reuse: spans teams, specs, designs
1c
11. Technical/Infrastructure Architecture Model
12. Build horizontal slices: UI, middleware, database
10. Find design rules for UI, database, communication…
13. Application Architecture Model + builds
1d
14. Standards: deliverables, templates, process, QA (inspection, test, CM…), tools
15. Package structure, vertical/horizontal slices, increments, tools, …

Figure 13.4 Typical project evaluation.

1. *The business case:* This provides initial requirements, defining the business problem or opportunity that this project addresses. It typically includes a high-level list of numbered functional and nonfunctional requirements[2] (1b), the business reasons and risks for the project (1a), the scope of the project in terms of things definitely included or excluded, linked clearly to a business model in terms of business objectives, actions or use cases, and user roles that must be supported (1a), known requirements on the architecture, design, and implementation (1c), and constraints on project budget and schedules (1d).

2. See Section 12.2.1, Multiple Stakeholders and Their Requirements.

2. *Domain or business models:* These describe the domain or business at hand, often independently of particular software solutions. It can include as-is and to-be models of the business. It is sometimes useful to analyze the as-is model, decide which aspects of it represent essential requirements, and abstract out an essential domain model that can then be refined to the envisaged to-be solution model. Domain models are reusable across multiple projects.

3. *Joint-Application Development (JAD) sessions:* Many projects can benefit from early structured sessions conducted by the development team and customers or users, whose purpose is to build a joint understanding of the requirements of the system to be built. They typically produce a running list of issues and the items listed under 4, 5, and 6.

4. *Glossary:* This is an initial set of definitions of terms used to define the problem or requirements. The glossary is developed in parallel with a type model, either for the system specification or of the domain itself. The glossary will be maintained through the iterations of the system specifications.

5. *Type model plus system specs:* The system context is defined, and the primary actions that the system participates in are first specified as abstract actions, and then refined to a level of an approximately *atomic* interaction with the system. Atomic interactions are those that, unless completed, would not constitute meaningful or useful operations on the system.

6. *UI sketches:* For systems that have user interfaces, initial sketches of the UIs are produced to define how user tasks might translate into system interactions, what information is exchanged in these interactions, and the dialog flow through these tasks. The user-interface elements are tied to the underlying type model. The names and labels for UI elements must map directly to attributes and types; their visual relationship to one another maps to relationships in the type model; the nature of the UI widgets used (lists, trees, fields, and so on) maps to static type model constraints such as multiplicity; and visual feedback, such as colors and highlights, maps to states. Do not get bogged down in UI design at early stages. The underlying type model, information exchange, and action specifications are more important.

7. *Subject areas:* These are broad areas of usage or function that help partition the description of the system behavior so that one area can be modeled and specified somewhat separately from the others. They need not correspond to separately implemented components; that would be an internal view.

8. *Frameworks:* Across the business models and system specification, there often are generic problem frameworks that appear in specific forms. For example, the business description (and hence system spec) for a seminar company might have frameworks for resource allocation (assign instructors and rooms), inventory and production (maintain course notes inventory for deliveries), and customer loyalty (monitor product preferences and usage levels of clients). Factoring the models to use generic frameworks (see Chapter 9, Model Frameworks and Template Packages) simplifies the descriptions and also gives you a basis for downstream design partitioning of application components and code frameworks (see Section 11.4, Frameworks: Specs to Code).

9. *Refactor for reuse:* This activity spans system specification and internal design. It involves rearranging parts of the models, sometimes extracting them into newly created packages and frameworks that are promptly reimported. This is an ongoing activity.

A common version is type-centric refactoring. A package containing a type model, either in specification or in design, shows a set of related types. However, to implement a particular type we need not know all the context-specific properties of all other types, only a subset. We could refactor the contents of this package into separate parts, each centered on a single primary type of interest and including only those aspects of other types that it must rely on to provide its services (see Section 7.4.1, Role-based Decoupling of Classes). This form also often shows up when you're designing class-based frameworks with plug-points (see Section 11.3, The Framework Approach to Code Reuse).

10. *Design rules for technical architecture:* This defines the elements used and patterns followed systemwide for dealing with the computing infrastructure aspects of the design. It includes hardware and software platforms and tools, middleware and databases, and the choice of API standards and component architecture, such as JavaBeans or COM. These rules emerge in parallel with the activities in items 11 and 12: architecture models and implementations.

Creating design guidelines against commercial or custom components and tools (some such design patterns can be formalized as frameworks), it defines standard mechanisms for mapping a type or class model to the chosen database, for presenting to and interacting with the user, for system boot-up and graceful failure, error handling, and so on.

11. *Technical architecture model:* This describes the package structure of and collaboration between infrastructure components at the technical architecture level, often treating each component as a single large object. All the techniques and diagrams for describing interactions apply, including collaboration, refinement, scenarios, and state models. The import structure of packages defines the static usage dependencies between component definitions and should be explicitly documented. The architecture should be explicitly evaluated against the runtime quality objectives (behavior, performance, scalability, and so on) and nonruntime quality objectives (modifiability and maintainability) for the system and should be documented and enforced in the implementation.

The component descriptions here typically belong to different domains than the primary business problem at hand. Database components are described by generic models of tables, columns, and keys (or types, classes, attributes, and IDs); transaction servers are described by transactional object, recoverable object, and resources; communication components could have sessions, channels, and messages; user-interface components have buttons, lists, panels, and scrollbars.

12. *Horizontal slices:* The technical architecture model should not be just a paper exercise. Instead, slices of the architecture model should be prototyped in a horizontal fashion: where each slice helps complete the end-to-end communication paths but does not introduce new end-user functionality. Nonfunctional requirements—throughput, scalability, data volumes, response time, and so on—should be tested carefully by using architectural simulation tools or by writing drivers to load the prototype appropriately.

13. *Application architecture:* This is the design of application logic itself as a collection of collaborating components. The nature of the component connectors—events, properties, workflows, replication, and transactions—is dictated by the technical component architecture selected. The specification types in the system spec are split across different components, possibly with multiple threads or processes.[3] These components can range from custom-built to common off-the shelf components, such as spreadsheets, calendars, and contact managers, to purchased domain-specific components such as factory-floor schedulers. The development activities here may include reverse-engineering, for those cases in which the application is being built from heterogenous off-the-shelf components (see Section 10.11, Heterogenous Components).

14. *Development standards for the project:* This project planning activity, which starts as early as possible, defines the deliverables, standard development tools, documentation structure and templates when appropriate, process guidelines, team and stake-holder roles, quality assurance standards including inspections and metrics, testing, change and configuration management, and so on. All these standards should themselves be documented in some of the top-level packages for the project.

15. *Project plan:* This project planning activity also starts early but is subject to monitoring and refinement as the development progresses. It includes defining the high-level structure of packages. These packages define the basic units of work and configuration management and serve to separate different subject areas, separate interfaces from implementations, separate business logic from infrastructure components, and also enable parallel development. Appropriate iterations and increments are defined, following a typically spiral development model, and new tools may be introduced in the process.

A large part of project planning in Catalysis is centered on the structure and intended content of packages, including documents, models, tests, and code. All other development work is done within the context of a package; the project planning directly depends on the architecture, because it is concerned with partitioning, relating, and scheduling of these packages.

It is quite common for a Catalysis project leader to set up a structure of empty packages and use them to enter "stub" specifications, designs, and relations (such as refinements) between them that must be completed in downstream development. These packages also need an appropriate build mechanism that traverses and evaluates the contents of each package, generating results that will be checked, compiled, and linked.

16. *Deployment:* It is in this phase that the business or domain makes its transition to the to-be model, adopting new processes, hardware, and software. It involves things such as software and hardware installation, tools to upgrade or migrate to new releases, documentation, acceptance testing, and user and administrator training. Note that user docu-

3. Some of these choices will be influenced by the technical architecture. For example, Enterprise JavaBeans does not like its components to implement multiple threads, because it tries to manage that at the level of the vendor-provided containers.

mentation is not an after-the-fact activity. Because modeling starts with abstract user actions, refined eventually to system operations, the refinement definitions form the basis for many user documents: *If you want to accomplish task X, then do operations a, b, and c.*

13.4 *Typical Package Structure*

The separation of outside, boundary, and inside—covering business models, system or component specification, architectural design, interfaces, and implementation—holds for all projects (see Section 7.3, How to Use Packages and Imports). The package structure reflects static definitional and usage dependencies, including third-party interface and implementation units and even development and test tools. Packages reflect versioned units, documentation structure, and even tools for builds (see Section 7.8, Publication, Version Control, and Builds). A package can contain collaborations to define the dynamic aspects of architecture. Figure 13.5 shows a typical package structure.

13.4.1 Levels of Description in the Case Study

In this case study we found it useful to distinguish six descriptions; a different project might make a somewhat different separation. The primary separation of domain or business (the outside), system specification (the boundary), and internal architectural and detailed design (the inside) will always hold. The following two features are common to each level.

• *Progressive formalization:* A progressive formalization turns natural-language statements into a less-ambiguous specialized notation. Casting statements into this more precise form clarifies them, exposing gaps and inconsistencies. (Programming language has comparable precision but is designed to deal with the solutions rather than the problems.) Although this formalization lengthens the analysis phase (compared with more traditional natural-language requirements), the design part of the process is clearer and shorter; and although the overall process is longer, the resulting system is more flexible and less costly to adapt and maintain.

• *Refinement:* Different levels of abstraction are supported by refinement. Giving an overall view helps everyone gain a better understanding of the problem. Sometimes the abstraction is written before the detail. In other cases, an abstraction is made after an initially detailed view—for example, to clarify issues after interviewing end users or to revise an external specification after a cycle of prototyping.

Following is a summary of the levels used in the case study; subsequent chapters treat each level in more detail.

13.4.1.1 Business (Domain) Model

This model describes the processes going on in that part of the world in which we are interested. These processes might be interactions between people or companies, physical processes, or the design of an existing computer system. The idea is to get a clear and

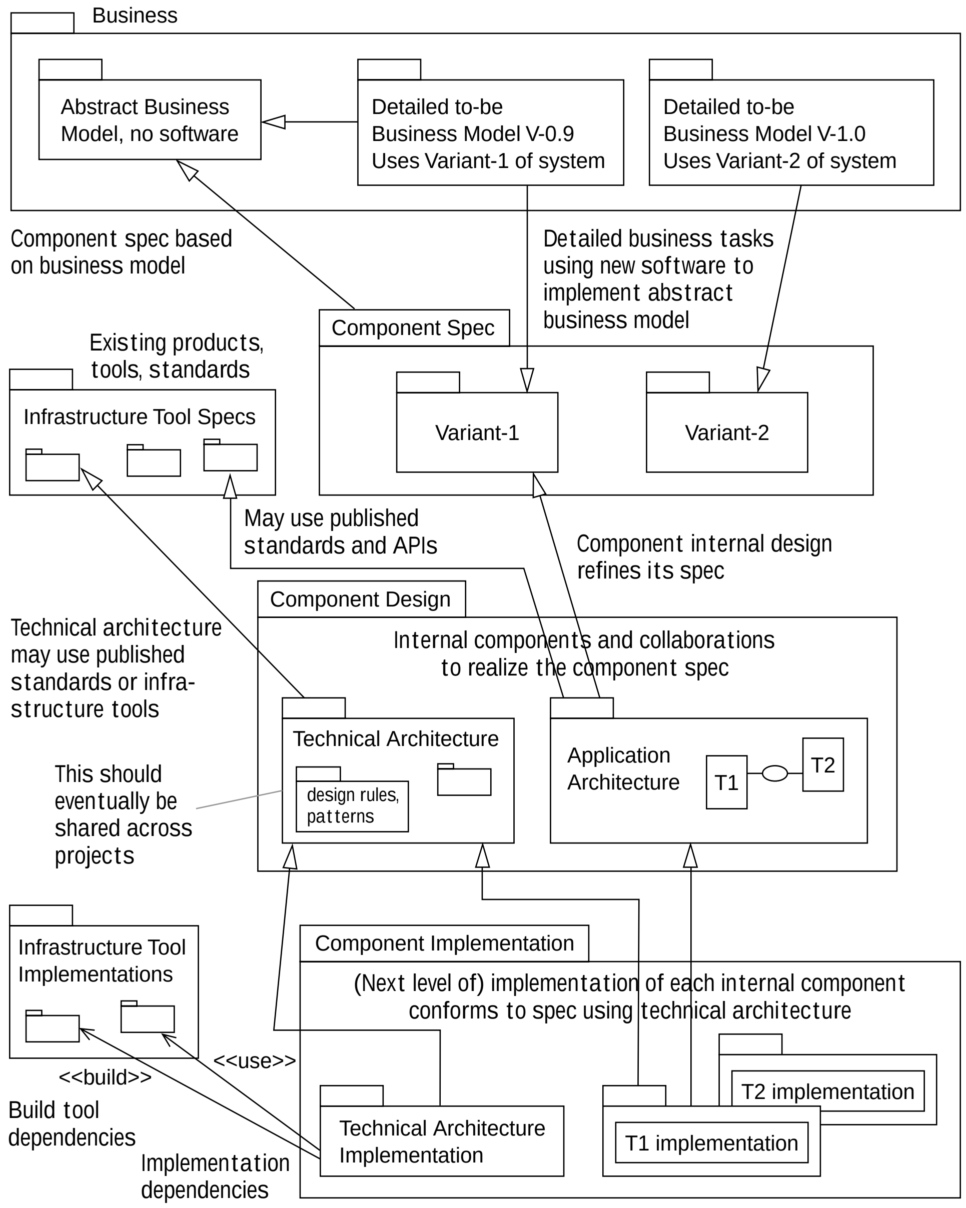

Business
Abstract Business Model, no software
Detailed to-be Business Model V-0.9 Uses Variant-1 of system
Detailed to-be Business Model V-1.0 Uses Variant-2 of system
Component spec based on business model
Detailed business tasks using new software to implement abstract business model
Existing products, tools, standards
Component Spec
Infrastructure Tool Specs
Variant-1
Variant-2
May use published standards and APIs
Component internal design refines its spec
Component Design
Technical architecture may use published standards or infra-structure tools
Internal components and collaborations to realize the component spec
Technical Architecture
design rules, patterns
Application Architecture
T1
T2
This should eventually be shared across projects
Component Implementation
Infrastructure Tool Implementations
(Next level of) implementation of each internal component conforms to spec using technical architecture
<<use>>
<<build>>
T2 implementation
Build tool dependencies
Technical Architecture Implementation
T1 implementation
Implementation dependencies

Figure 13.5 Typical package structure.

agreed understanding of the concepts and rules important to all parties, especially the users and the designers of any existing systems.

The lightest-weight domain model is simply a problem statement, with a glossary of terms that are used in the statement. More-precise models use types, collaborations, and refinements to define static and dynamic business rules.

The description can be made without reference to a particular system we propose to design. It can be used either as the basis for a system analysis or just to get a clear view of the business processes, with a view to improvement or reengineering of business processes.

If the model is to be used as the basis for a system development, we would expect to see many of the object types identified at this stage carrying through the system analysis, design, and ultimately into the program code. The same business model should be relevant to many systems.

13.4.1.2 System Context

This involves understanding the role the system we wish to build will play in the context of the business: the interactions it will have with other objects in that world. Our system is part of a larger design, whether of software or hardware components or people, that may or may not have been explicitly described elsewhere. The objective of this part is to capture that information.

13.4.1.3 System Specification

This is a precise description of what is required of our system. It should be documented independently of how the requirements may be achieved. Clearly it will be necessary to think ahead, and consider some implementation questions, in part because there’s no point in specifying something that cannot be made and in part because the creative minds of the analysts inevitably speculate about how it will work. We’ve already discussed the difference between doing these steps in parallel and documenting them separately.

13.4.1.4 Component Models

In this study, we assume that there are several large components from which we wish to build. They may already be purchased or may exist from a previous piece of design.

We make specifications of each component; again, they may already exist, or we may build them from our understanding of the components. In dire cases, this means reverse-engineering models for these components, experimenting with them in testbeds. The process of formalizing the spec exposes the questions that need to be answered.

13.4.1.5 System Architecture: High-Level Design

Here, the components are connected to form a design that meets the system spec. It is also important to document a justification for believing that it does so properly. As a high-level design, this describes only the overall scheme of interactions: large-grained components and their collaborations as well as corresponding static dependencies needed between different development packages. The detailed coding will be left until coding.

In this case study we have not adopted any particular component architecture such as Cat One (see Section 10.8.1, Cat One: An Example Component Architecture). So our connectors have been limited to standard requests or responses between large-grained components rather than higher-level facilities such as events, properties, and workflow transfers.

13.4.1.6 System Detailed Design

Further work is required for those components that are used in the design but not yet implemented—not purchased or adapted from previous projects. Just as we had a specification of the whole system and broke it into successive pieces, so each component can be broken into subcomponents, which in turn are specified and implemented until we get down to units directly implementable in program code.

13.5 *Main Process Patterns*

The patterns in this section form the basis of a customizable process of developing software; they show how the techniques are also applied to engineering business processes. Customization of these patterns would result in different routes through the method. Many, but not all, of these patterns have been applied and tested in practice.

13.5.1 Development Context: Defining Routes

The first few patterns offer a breakdown of how to proceed along a route through the method, given different assumptions about your goals.

- *Pattern 13.1, Object Development from Scratch*, shows how to proceed assuming that you have no existing design.
- *Pattern 13.2, Reengineering*, assumes an existing design and shows how you should go about improving it.
- *Pattern 14.1, Business Process Improvement*, is about applying object technology to organizations other than software.
- *Pattern 16.6, Separate Middleware from Business Components*, offers one strategy for handling legacy systems as well as for insulating the project from certain technology changes.

Regardless of which route, or combination, is appropriate, most projects would benefit from carefully managed iterations (see Pattern 13.3, Short-Cycle Development) and from concurrent development work (see Pattern 13.4, Parallel Work).

13.5.2 Phases

Each of the development context patterns applies some combination of the following patterns, which deal in different phases and activities of the development process. The idea is that they form a kit of tools rather than a fixed procedure. To begin with, you apply them by following one of the development context patterns; with experience, you apply them as needed.

- Business or domain models
  - *Pattern 14.2, Make a Business Model*: Understand the terms your clients are using before anything else; capture business rules, behaviors, and constraints that are independent of software solutions.
- System or component specification
  - *Pattern 15.5, Make a Context Model with Use Cases*: Understand the collaborations with and around an object.
  - *Pattern 15.7, Construct a System Behavior Spec*: Treating your system as a single object, define the type of any implementation that would meet the requirements.
  - *Pattern 15.8, Specifying a System Action*: Specify an action with the help of snapshots.
  - *Pattern 15.9, Using State Charts in System Type Models*: Building state charts of specification types is a useful cross check for completeness and uncovers missing cases of actions and effects.
  - *Pattern 15.12, Avoid Miracles, Refine the Spec*: Go into more detail about actions and the corresponding attributes; alternatively, abstract away from details to a higher-level, more task-oriented model.
- Internal design: technical architecture
  - *Pattern 16.6, Separate Middleware from Business Components*: Keep separate your legacy systems, business models (newly built or wrapped around legacy systems), and infrastructure middleware elements.
  - *Pattern 16.7, Implement Technical Architecture*: Define major technical components of your design as an architectural collaboration. These components might be GUI, business logic, persistence (database or file system), communications, and other middleware elements.
- Internal design: application architecture and detailed design
  - *Pattern 16.8, Basic Design*: Take each system action and distribute responsibilities between collaborating internal components. For small systems, begin by assuming that there is a direct implementation for every specification type of the system type model. For larger systems, use intermediate levels of collaborating large-grained components.
  - *Pattern 16.10, Collaborations and Responsibilities*: Document and minimize coupling by designing responsibilities and collaborators; compose patterns of collaborations to define the design.
  - *Pattern 16.11, Link and Attribute Ownership*: Decide and document the directionality of links and the visibility of attributes.
  - *Pattern 16.12, Object Locality and Link Implementation*: Locality of an object—where it resides and executes—can be decided independently of basic design, employing appropriate patterns to implement links and messages crossing locality

boundaries. This applies across boundaries of hosts, processes, applications, and media.

  - *Pattern 16.13, Optimization*: Apply localized refinements to make the project run faster.

- Implement, test, deploy

  The line between design and implementation/test is a fine one; unfortunately, some people believe that textual descriptions, including code in C++ or Java, represent implementation and that diagrams represent design. Others use terms such as *diagrams* versus *specifications*. But that is simply a matter of notation; you can draw diagrams to assemble components and generate executable code. The Catalysis distinction between interface specification as a type—refined to an internal design as a collaboration—covers implementation in any form, whether expressed as diagrams or text.

  A Java class chooses its internal data members that collaborate to implement a Java interface; similarly, a traditional program with a set of global variables referring to its collaborating top-level objects implements a specification of the entire process as a type.

  Moreover, as discussed in Section 13.1.2, Specify, Document, Implement, and Test: Business to Code, activities to build, install, and deploy a system are consistently thought of as implementation steps at different levels: implementation of internal design, system spec, and the to-be business model.

Pattern 13.1 Object Development from Scratch

This pattern describes how to build a design starting from scratch. This route is suitable when you're designing a computer system or subsystem with no existing installation, no available major components to reuse, and no existing model of the business.

Considerations

Is there really no legacy? Consider people who have worked on an earlier system, or existing implementations that could perhaps be componentized and reused to reduce the development effort (see Pattern 10.5, Using Legacy or Third-Party Components). What are the business changes that accompany deployment? Use Pattern 13.2, Reengineering, to plan the transition and successive deployments.

Strategy

Apply Pattern 14.2, Make a Business Model, and Pattern 15.13, Interpreting Models for Clients, to the following series of phases.

• *Pattern 14.2, Make a Business Model*: Describe your understanding of the users' concepts and concerns and the vocabulary in which they express them. This activity is not limited to any single set of requirements and can serve as the basis for more than one development project.

• *Pattern 15.5, Make a Context Model with Use Cases*: Focus on the collaborations between your proposed system and other objects—people, machines, other software systems—with which it will interact. Also include relevant collaborations between them not involving your system, improving your understanding of what's going on around it.

• *Pattern 15.7, Construct a System Behavior Spec*: Treating your system as a single object, create a type specification for any system that would meet the requirements. Actions (and hence the type model) should be as abstract as possible at first cut—not individual keystrokes!

• *Pattern 15.12, Avoid Miracles, Refine the Spec*: Define more-detailed actions and attributes as a refinement.

• *Pattern 16.7, Implement Technical Architecture*: Define and implement major components of design as a collaboration. Typically, these might be GUI, client, business logic, persistence (database or file system), and communications.

• *Pattern 16.8, Basic Design*: Take each system action and distribute responsibilities among collaborating internal components. Begin by assuming that there is a class for every type of the system type spec. For larger systems, design an intermediate level of large-grained collaborating components.

• *Pattern 16.11, Link and Attribute Ownership*: Extract common components and recast the design in terms of the components.

- *Pattern 16.12, Object Locality and Link Implementation*: Decide how the basic design is split among machines, applications, and hosts.
- *Pattern 16.13, Optimization*: Perform localized refinements for performance.

An object-oriented design according to these principles can be fully traceable from business requirements through to code, whether or not the correspondence is direct.

Pattern 13.2 Reengineering

In this pattern you make an OO system using both the knowledge derivable from legacy code and the legacy code itself.

Intent

The idea is to gain the benefits of OO, but without throwing away old code. You want to be able to make systems that remain flexible as your organization and its structure and working methods change. You want to build many applications from a set of basic components. These are features of object-oriented designs.

You have a great deal of code written in an older tradition—for example, a relational database and its driving software. You cannot afford to hold everything while you rewrite it all.

For example, one of the authors worked a while ago with a GIS system that had been conceived 10 years earlier and written in Fortran. It was successful, so over the decade many features were added by popular demand. As the system gradually lost its original coherence, it came to the point that you couldn't tweak one end without the other falling over. Most of the patch-makers weren't around during the original development, so they understood only as much as they needed to. Many local changes were made that should ideally have been done at a more global level, and even from the outside it looked like a bit of a mess. It was decided to reimplement the system gradually as an OO design so as to give it another lease on life. However, there was no way for the organization to stop for a couple of years and rewrite it all: The company depended on income from maintenance contracts, and that meant providing customers with regular new features. How should it proceed?

In another case, a bank was writing an inquiry and loan application processing system. We wanted to get the benefit of object technology so that we could easily fashion variants of the system for different banks. But there was no point in building it entirely from scratch: The conventional databases were already in place, although each bank had a different one. How could we integrate the new technology with the old?

Considerations

Are you going to reuse the old code; or only the old design ideas; or only the old business model? In general, each of these elements can be reused using appropriate techniques. In each case, it is essential to capture the existing concepts first (using the modeling notation in earlier chapters) and, separately, to document where you'd like to be.

Are you going to wrap old code and gradually rework it, or leave it wrapped and rework only when essential, doing redesign in the wrapping layer?

Are you going to reengineer the business process of which the software system forms a part? (See Pattern 14.1, Business Process Improvement.)

Strategy

Make As-Is and To-Be Models. This can be done at several levels: business, system requirements, and system design.[4] The as-is model keeps fairly strictly to what exists at the present. (Because of ad-hoc changes over the years, it may be horrible. You're allowed some omissions when early reworking is an obvious necessity.) The to-be model is your vision of the ideal future. In practice, you should also make an intermediate near-term objective. It is often useful to abstract from the as-is model a truly "essential" model of what must be done; the as-is was the previous refinement of this model, and the to-be will be an alternative, improved refinement (see Figure 13.6).

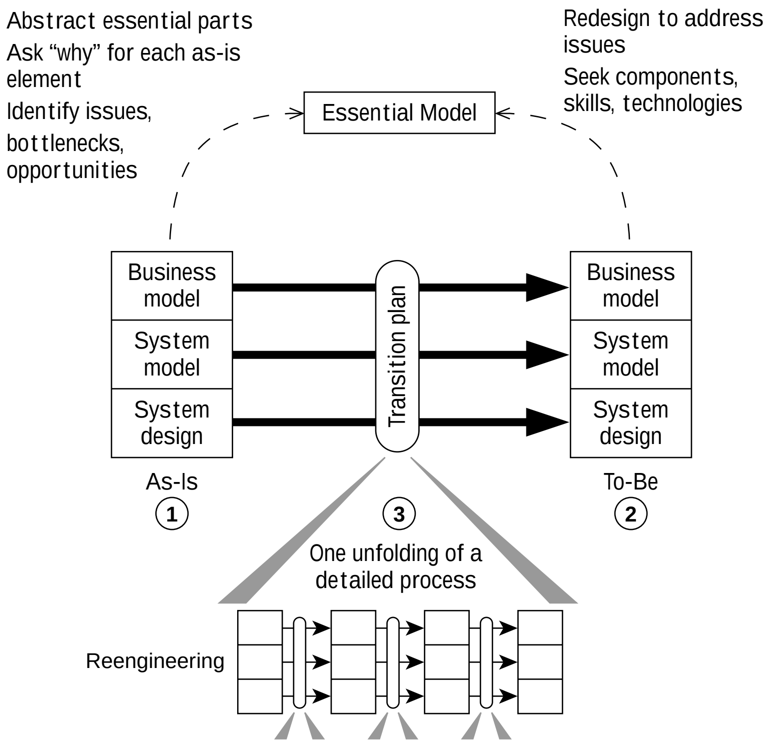

Abstract essential parts
Ask "why" for each as-is element
Identify issues, bottlenecks, opportunities
Essential Model
Redesign to address issues
Seek components, skills, technologies
Business model
System model
System design
Transition plan
Business model
System model
System design
As-Is
1
3
To-Be
2
One unfolding of a detailed process
Reengineering

Figure 13.6 Reengineering and transition plan.

Plan the transition from as-is to to-be via intermediate steps, perhaps based on your chosen path or process. Be more specific in planning for the earlier transitions, because subsequent ones will inevitably be replanned as you approach them (see Pattern 13.3, Short-Cycle Development).

4. This strategy was formulated by the methods groups at TI-Software, London.

As-Is informs To-Be. The as-is model may be the best starting point for the to-be model. The original architects of the old system probably had a clear model, which has been obscured by numerous fixes and add-ons through the years, typically by younger folk who haven't entirely cottoned to the beauty of the original conception. (You can often find some of the old-timers still hanging around: They get excited about building these models, because they see it as a great opportunity to clear out all the indignities their brainchild has suffered over the years. Be sure not to leave them out of the activity, but don't be surprised if they come to blows with one another about the precise details of the original big idea.)

On the other hand, do not take legacy terminology as Gospel truth; always seek a more essential description if the terms used seem contrived or artificial. Be particularly suspicious of legacy terms that are purely artifacts of an ingrained way of doing things—for example, complex "codes" that are used to label business objects with compound meanings.

Abstract and Re-refine. This is an important technique for reengineering (see Chapter 6, Abstraction, Refinement, and Testing). Abstract the as-is model to form a more general statement of requirements. For example, the sequence of actions <customer inquiry; issue loan application; receive application; approve application> can be abstracted to the single action make loan. Then consider whether you can redesign the abstraction a better way (but don't do so just because you can!). For example, you might redesign the abstraction to <customer-inquiry(details); approve application>, eliminating the steps to transfer an application.

Apply Abstraction and Re-refine to Each Development Layer. Do this for the business model, the system context model, and the abstract and detailed layers of design, as listed in Section 13.5, Main Process Patterns. Each of the three principal layers can be approached differently.

Reengineering Business Model and System Requirements. These can often be reengineered directly toward the ideal. There isn't always so much complexity in the processes of human interaction that they can't be altered, although you should proceed with caution through people's sensitivities (see Pattern 14.1, Business Process Improvement).

Reengineering System Designs. You can apply small changes by the usual patching process. But radical changes require bigger solutions, and some systems should be explicitly designed to allow upgrades while they're running. Furthermore, the prospect of continuing change demands that business logic be decoupled from the underlying system. These considerations lead to the three-layer model, or middleware (see Pattern 16.6, Separate Middleware from Business Components).

An alternative is to consider gradually carving out and redeveloping the software. In many cases, realistically, such a process would not end within the life of the software and could ultimately represent much wasted effort. Instead, it is better to develop new components in OO terms, leaving the old stuff cleanly wrapped.

Pattern 13.3 Short-Cycle Development

In this pattern, you set specific short-term targets and more-general longer-term ones, and you use early feedback through scoped and managed short development cycles. This pattern is also known as "one step at a time," "don't chew off too much in one go," "walk before you run," the "spiral model," and "proceed with caution."

Intent

In any project in which the outcome of one phase will affect the plan for what follows, we need a systematic and realistic approach to planning by successive approximation, when the outcome and success of each piece of work is known only when complete. This does not apply (apart from exceptional circumstances) to building a house or baking bread, when the requirements, inputs, and outcomes of all steps are fully understood in advance. It does apply to all design projects, whether software or hardware, to all changes of organization, such as adopting new design methods, and to all research and development.

Considerations

It is unrealistic to write a linear plan for a research project; if you knew how each stage would go and what it would lead to, it wouldn't be research. The same holds for any intellectual work. But we need some way of planning.

Reaching a project goal is a bit like leaping from one spot to another. If you try to do it all in one leap, you must aim carefully to begin with, because it's difficult to change course in midflight. Object technology is good at handling changes and so enables us to do the job as a series of shorter jumps. There's less investment risked in each one, and because we can correct our direction at each step, we're surer of reaching the goal in a predictable and controlled fashion.

Customers like short cycles because they see results early and can take part in the development early. They don't like hearing nothing for a year and then getting a system that doesn't do what they want.

Developers like short cycles because it gives them a periodic feeling of achievement.

Project managers like short cycles because they feel they're in some sort of control as long as it does not degenerate into a euphemism for undirected hacking. They also like the Böhm's spiral model because it gives a respectable name and rationale to the fact that they don't know exactly what will be happening in week 42. That was always the case anyway, but it could feel a little uncomfortable in front of senior management.

Strategy

Plan in short cycles, each of which ends with the assessment of a deliverable that is measurable in terms of the ultimate goal, which then feeds into the planning of the next few cycles. Expect only an approximate idea of the cycles ahead of time.

Begin with cycles that use small investments to tackle issues that represent high project risk. Typically, these risks fall into two categories: requirements and technical architecture. More resources can be fed into successive cycles as confidence is gained. The idea is fractal: a big project's single cycle can be composed of several smaller cycles of subprojects.

Let cycles overlap and proceed in parallel (see Pattern 13.4, Parallel Work).

Each cycle consists of plan, execute, and evaluate phases. Set goals, level of investment, and acceptable risks; plan and decide what will be evaluated and how it will affect future cycles; determine fallbacks; execute the plan; evaluate the outcome. For software development cycles this often translates into plan, specify, design and implement, and evaluate.

Typical cycles in software development might include feasibility study, GUI mock-up, requirements analysis and prototype, single-user, single-machine vertical incremental slices, tightening of relations between documentation layers, distribution across hosts, and multiuser deployment. A typical 10-person software development might use cycles ranging from two to six weeks.

Make your package structure reflect the separations needed across iterations so that different work products can be managed and developed separately.

User-visible cycles should deliver meaningful functionality to users. Plan deliverables for these cycles in terms of abstract business actions (use cases) for the users based on their prioritization and on dependencies between these actions that are uncovered by system specification. For each use case delivered, track all refining system actions and corresponding internal component interactions, and schedule cycles accordingly. For any design element (action, effect, attribute, invariant), schedule its development based on the earliest scheduled use case that uses it.

Other cycles can be horizontal: one that does not deliver new user-visible functionality but instead carries a minimal use case through increasingly deep layers of the application and infrastructure components, exercising all communication channels. An example is a single user interaction carried from the user interface through the business object layer via an object request broker (ORB) to the applicable databases and back.

Early user cycles need not build on the technical architecture; instead, treat them as prototypes that will yield early feedback from users. These cycles—vertical slices of user-visible functionality—are focused on correct visible functional behavior at the user interfaces. They might be implemented purely as a single-machine, single-process prototype.

Adoption of new methods within a company might have cycles that begin with a small demonstration by a few people. Their experience feeds into something bigger, training for more people, selection of tools, and so on. With careful planning, the new methods may be taken on by a large project or complete division over a period of many months to years. (Trying to do it overnight doesn't work!)

Pattern 13.4 Parallel Work

Structure your packages to enable units of work to be done in parallel without excessive risk of rework. Do this early in the project, and sustain it throughout the project.

Intent

A project development team completes a multitude of tasks over the project lifetime. Some tasks are heavily dependent on others and should be serialized, perhaps using Pattern 13.3, Short-Cycle Development; others are largely independent and should proceed in parallel because downstream work is dependent on them.

Considerations

Some tasks are best done by small groups. For example, clarifying requirements and defining a precise, even if high-level, system specification is an activity best done with a few key people. Similarly, coming up with an initial definition of system architecture, and subsequently maintaining and evolving this architecture, is best done by a small team.

On the other hand, projects are often staffed early. Keeping the larger team constructively involved throughout the project, while minimizing rework, is good for the project schedule as well as team morale.

Strategy

As early as possible, define the system context and all known constraints on the system, its initial architecture, and internal components. Examine interfaces to external systems. There is often significant work in realizing the connections to those systems, whether it involves communications, hardware interfaces, database requests, and so on. This work can be started early.

Across the entire specify-implement-test cycle (Figure 13.3), there are several mostly independent tasks that can be run in parallel, ranging from setting up development tools and environment, to writing installation programs, to defining process and QA standards.

Partition system specification into subject areas. Define early the key type model attributes or effects that one subject area depends on; another subject area can specify the detailed constraints on those attributes or effects. Always have a mechanism (such as attributes or effects) that relates subject areas to one another.

Start work on the technical architecture early. This may involve acquiring and learning third-party packages. Build architectural prototypes based on a test slice of the system spec, and evaluate the architecture against nonfunctional requirements—throughput, scalability, data volumes, response time, and so on—by using architectural simulation tools or by writing drivers to load the prototype. Understand the capabilities of middleware packages; databases and transaction servers often offer services that can reduce the development work on the main application logic.

If there are external components—software or hardware—that define objects you need to use, spin off a task to evaluate whether to use these objects exactly as defined or whether to build a layer that offers a model closer and more natural to the one you would like to use internally in your development. If a core component defines widely shared and widely used objects, you may need to design a generic architectural scheme for extensible object data and behaviors.

Start user-interface prototyping when the high-level system actions and type model have been identified. Focus on a consistent UI metaphor, and cross check the user interfaces against the type model as the latter becomes better defined.

Structure packages to consistently separate interfaces from implementations, and aim for early initial definition of interfaces. Also, designate team roles that cut across parallel activities, watching for issues or overlaps that should be addressed separately.