The Process of Object-Oriented Design

Dennis de Champeaux$^{(1)}$
Doug Lea$^{(2)}$
Penelope Faure$^{(3)}$

$^{(1)}$ *HP-Labs, MS 1U-14, 1501 Page Mill Rd, Palo Alto, CA 94304-1181*
$^{(2)}$ *NY Case Center & SUNY Oswego, Oswego, NY 13126*
$^{(3)}$ *Faure Inc, 1952 Camden Ave, Suite 101, San Jose, CA 95124*

Abstract

The object-oriented design process is investigated within the classic software development classification of Analysis, Design, and Implementation. When all development is performed using object oriented methods, OOD is best characterized as a transformational process, mapping declarative descriptions of objects and classes to implementation plans. Subphases of class design, system design, and program design accommodate and rationalize OO practices including abstraction, prototyping, refinement, bottom-up composition, delegation, interoperablity, clustering objects into processes, resource management design, tuning and optimization.

1 Introduction

As object-oriented concepts increasingly permeate through software engineering methods and practices, some of the clashes between old distinctions and new concepts become a bit uncomfortable. The most notable stress point introduced within attempted integrations of object-oriented notions and software engineering process models lies in the nature of *design*.

Process models describe and/or prescribe software engineering practices by classifying activities into various phases and components (see, e.g., [15]). Nearly all models make at least the coarse distinction between *Analysis*, *Design*, and *Implementation* phases of development.

Object-oriented approaches to analysis and implementation fit neatly within these categories. Even the massive differences between object-based modeling and classic "structured" analysis methods are readily accommodated within a number of general process models. So is the fact that object-oriented programming languages, tools, and environments offer a greater diversity of useful constructs and idioms compared to standard procedural languages. These differences have led to new prescriptive models of good software engineering practices. They include spiral, fountain, prototype-based, and other models that reflect the iterative nature of object-oriented classification and decomposition [9].

However, the very notion of "design" is a casualty of this integration. In classic scenarios, the documents and specifications produced by the analysis phase are assumed to be so far removed from implementable constructs that a great deal of effort must be expended in order to re-address the project from a computational perspective.

This isn't true of object-oriented methods. Among the main attractions of OO approaches to software development is *structural continuity*.

Many classes, objects, properties, relations, and behaviors described by OOA models show up in obvious ways in the corresponding implementations when programmed in languages like smalltalk, C++, CLOS, etc.

Given this observation, one might naively argue that if developers are provided with good OO analysis and modeling strategies, along with good OO programming languages and techniques, then there is little that needs to be "designed" when software is constructed using an object-oriented approach.

In this paper, we will argue otherwise. By examining the nature of object-oriented methods within a tripartite structure, we arrive at a novel characterization of OOD. This view of OOD reveals a sequence of activities that appears necessary for successful object-oriented software engineering, especially when applied to large software development efforts. We also provide rationales for methods and activities that are commonly, but anecdotally, held to be good practices.

2 Separation of Concerns

For small enough efforts, choices of software process models, development philosophy, or orientation don't matter very much. As things scale up, these choices start making a difference. Development roles ("domain analyst", "tester", etc.) need to be assigned, efforts coordinated, schedules managed, and outcomes predicted. This requires a sometimes-artificial separation of roles and concerns across different aspects of development.

As already mentioned, this separation almost universally results in the identification of at least three major software engineering phases, Analysis, Design, and Implementation. Unfortunately, none of these terms really capture the nature of the constituent activities. This makes it too easy to fall into meaningless debates, like "Is Design really Analysis?". But the terms aren't *completely* devoid of meaning. The main conceptual distinctions may be characterized as follows:

- Analysis activities provide a *declarative* description of what the proposed software is supposed to do.
- Design activities provide a *computational* description of software that meets analysis requirements.
- Implementation activities are *environmental*, providing an expression of the design suitable for the target environment(s); i.e., programs written in particular languages, using particular tools and systems, for particular configurations.

There are, of course, many other subsidiary obligations and activities within each phase. For example, analysts are generally required to produce customer-oriented documents and specifications, implementors are bound to perform testing, etc. Also, there are other sets of activities that don't fall within these headings, like scheduling and maintenance, that we will not have much to say about in this paper.

Within large, managed engineering efforts, these phases are staged sequentially, because of their intrinsic logical dependencies. It is necessary to know what the proposed software should do before thinking about how it will do it. And to know both what it will do and how it will do it before committing to its expression on a particular platform. These observations remain true across different levels of granularity, precision, and interleavings of efforts. They also admit feedback and post hoc refinement, as when experimentation with a prototype demonstrates that models, design specifications, and/or implementation strategies need reworking.

The Role of Design

The design phase serves as a bridge between analysis activities describing properties that a system should possess, and implementation activities describing the language and environment dependent manner in which they are programmed.

The inputs to the design process are, of course, heavily dependent on the nature of the analysis strategy. Until recently, object-oriented designers almost always received as input a set of descriptions and specifications produced through some kind of "structured analysis" method ([20]), or through no systematic method at all. In such cases, a great deal of "design" turns out to be "re-analysis".

However, the nature of design activities change dramatically when preceded by an object-oriented analysis phase, and followed by an object-oriented implementation phase. When analysis models and programs have the same overall structure, the essence of design is *transformation.* The goal of the design process is "merely" to transform a declarative description into an implementable one. But these transformations are by no means trivial. There are an infinite number of possible mappings between any set of declarative specifications and corresponding programs. A good design process model raises the chances of producing at least one.

In order to gain a foothold on this transformational process, we need to better characterize the nature and results of object-oriented analysis. In the next section we sketch out the general methods and goals of OOA, and then return to consider the implications for the design process.

2.1 The Nature of Object-Oriented Analysis

The purpose of analysis is to capture in a semiformal way, the requirements and constraints of a target system. The analysis activity accepts as input a fuzzy, minimal, possibly inconsistent target specification, user policy, and project charter. There are three main outputs:

Functional The purpose of the system, as described by (multiple) declarative models of objects, classes, relations, states, transitions, interactions, etc.

Resource The computational substrate(s) that the system will be built upon.

Performance The expected response times of the system.

OOA differs from other analysis methods in the way it approaches functionality descriptions. The other information is a necessary result of any analysis orientation, and is often obtained within a preceding or concurrent *requirements analysis* subphase that also attends to auxiliary issues like cost estimation.

Object-oriented analysis methods describe systems as sets of objects. Essentially all OOA techniques are modeling-based. They implicitly assume some kind of abstract computational structure. But to remain declarative, object dynamics are described in terms of constraints, conditions, and effects.

For purposes of analysis, the abstract object model, notational system, and associated methods are chosen in order to maximize descriptive power. They need not conform to models and constructs most easily supported using standard computers and programming languages. But they cannot allow commitment to details that will depend on the ways in which objects will ultimately be represented in software. A prototypical OOA framework includes the following:

- *Objects* are described as members of *classes.*
- Relationships between classes are described using *inheritance* and *parameterization* constructs.
- Object *attributes* and logical *states* are described as abstract properties, along with constraints on these properties.
- *Relationships* between objects are defined in terms of the nature of their participants, conditions for entering and leaving the relations, and constraints on the objects while engaged.
- Objects are *active*, process-like entities that may communicate asynchronously with one another.
- State *transitions* are described in terms of their triggering conditions and their effects on the target object.
- *Interactions* between objects are similarly described as conditions and effects on participating objects.

- Complex objects (e.g., subsystems, or, more broadly, *ensembles* in the sense of [5]) are described as constrained object collections, along with additional aggregate properties. One of these complex objects is the main system itself.

There is a lot of room for variation here. For example, communication may be represented in terms of asynchronous point-to-point messages, blocking procedures, and/or undirected "events"; multiple inheritance may or may not be used to relate classes, and so on. Any particular OOA framework represents some balance among expressiveness, power, economy, tractability, ease of use, and related factors.

Models, notations, and methods must also accommodate the analysis process; i.e. the steps involved in identifying, analyzing, and constraining objects, properties, states, transitions, and interactions. Such considerations typically lead to the following:

- *Multiple notations.* A combination of graphical, semi-graphical, structured-text (e.g., [19]), and natural language descriptions.
- *Multiple models.* Information may be spread across class property diagrams, class hierarchies, Entity-Relation diagrams, state charts ([8]), transition diagrams, use-case flow graphs ([11]), and other views.
- *Multiple outputs.* Functionality, resource, performance, quality, platform, and environmental requirements may be isolated into separate documents.
- *Multiple audiences.* At least some parts of analysis models are intended to be readily understandable by non-specialists, especially customers.
- *Variable precision.* Notations and methods allow for as much or as little precision in specification as people are willing or able to provide. Analysis frameworks neither require nor preclude the consistent use of formal methods.
- *Variable depth.* Analysis models may omit mention of classes, constraints, operations, etc., that do not appear to affect the overall functionality of the system, with the expectation that these issues will be further examined downstream.

While the details of the analysis *process* are beyond the scope of this paper, we note a few highlights:

- *Top-down decomposition,* often involving separate, staged treatment of object statics (properties, relations) and dynamics (transitions, interaction).
- *Domain analysis,* including the examination of related systems in order to determine potentially reusable concepts, components, architectures, etc.
- *Feedback,* especially side-channel experimentation with prototypes and existing systems, most notably for specifying properties of user interfaces.

OOA usurps many of activities often ascribed to OOD and OOP (by, e.g., [2]). When analysis is approached in an object-oriented manner, the constituent activities focus on the identification, classification, decomposition, and specification of objects and classes. They do so within a framework intended to maximize the power, sensitivity and generality of the resulting models, without restricting themselves to immediately implementable constructs. While these models may be revisited, transformed, and refined during design and implementation, the vast bulk of important structural information is produced during OOA.

3 Designing Design

We have characterized design as a transformational process that starts with a declarative, non-computational specification, and then applies methods and strategies that result in an implementable software design. By analyzing intrinsic constraints and features of these transformations, it is possible to specify a prototypical design *process*, prescribing activities and orderings among them.

Transformational Criteria

The structure of OOD relies on criteria common to the design of *any* transformational process. These include:

- *Separation of Concerns.* Group transformations into meaningful, tractable phases with well-defined inputs and outputs.
- *Sequentiality.* Obey logical dependencies. Do not schedule a step until its prerequisites are complete.
- *Conservatism.* Keep downstream options open. Avoid premature commitments to inessential details.
- *Generality.* Operate on the most general representation possible for any transformation in order to minimize redundancies when performed on special cases.
- *Continuity.* Output refinements and restructurings using the same representational framework as their inputs.

Several of these considerations were noted in our discussion of overall software process models. But they also govern the architecture of just about any transformational system. Examples include compiler design, simulation system design, and vision processing system design. We will apply them to the human activity of object-oriented design itself.

The differences between the OOD process and, e.g., compilation techniques are mainly matters of degree. The transformational structure of modern compilers is well understood, and contains many algorithmic and semi-automatable components. But the prospects for mechanizing the "compilation" of an OOA specification into an implementable design appear remote. Many OOD transformations are underdetermined, heuristic, and involve substantial creative effort.

Structural Improvement Criteria

The heuristic nature of the enterprise mandates that any OO process model allow for incremental improvement throughout development.

Any design process that relies on the omniscience and perfection of analysts is doomed to failure. Methods must allow for analysis models to have occasional gaps and imperfections. Constraints and opportunities that stem from computational concerns can strengthen, complete, or override those seen from a declarative perspective. Similarly analysts (even domain analysts) will not always recognize and exploit common design idioms, reusable components, and applications frameworks. And regardless of these considerations, the diversity of OO constructs allows many concepts to be described in any of several nearly equivalent ways (e.g., multiple inheritance versus composition). The best choice from a design perspective need not mesh with that from analysis.

Thus, designers must be able to introduce new classes, refactor class hierarchies, and intertransform constructs in the process of meeting other goals. This requirement extends the breadth and impact of continuity criteria. By maintaining continuity, the process may more gracefully accommodate this kind of iterative structural improvement. So long as analysts, designers, and programmers maintain the same general views of objects, classes, etc., these steps may be undertaken throughout the development cycle in a fully traceable manner.

Among the best tools for assessing the need for such improvements is prototyping. The design process should accommodate the creation of prototypes that reflect only those transformations and details already committed to. These steps may then be revisited after experimenting with the tentative system.

Major Design Phases

The simplest way to reconcile the results of OOA with the process constraints listed above is to identify distinct design phases associated with each of the three principle analysis outputs describing functionality, resources, and performance requirements.

These three categories subdivide focal issues in a natural way. We can rename and expand on these groupings in order to clarify their relation to commonly held design phases:

Class Design Definition of representational and algorithmic properties of classes obeying the declarative constraints specified within OOA.

System Design Mapping of objects to processors, processes, storage, and communication channels; along with design of facilities to manage these resources.

Program Design Reconciliation of functionality and resource mappings in order to meet performance requirements when expressed using the target implementation languages, tools, configurations, etc.

The three subdivisions form a set of logical dependencies, indicating that they are best performed in the stated order. It is impossible to assign resources to objects and manage their use until resource demands are at least approximately determined by establishing representational and computational properties. It is similarly impossible to address performance issues until these mappings are known.

This sequencing also meets our other criteria. By dealing with class design issues before systems issues, designers may avoid premature commitments. They may perform transformations that will hold whether instantiated objects reside on individual processors, are passive components within managed processes, share resources with others, and/or hard-wire their communications channels with other objects.

3.1 Class Design

The principle goal of class design is *reification.* Class design methods arrive at internal representational and algorithmic specifications that meet the declarative constraints of analysis models.

Other class architecture requirements are implicitly or explicitly introduced during the class design phase. Analysis models do not address the nature of classes as software artifacts. Additional criteria may include:

- *Software quality* requirements, including reliability, modularity, safety, cohesion, testability, understandability, reusability, and extensibility.
- *Lifecycle* requirements, especially for system evolution, demanding design allowances for reimplementation, repair, extension, and related adaptations necessary for coping with future requirements.
- *Compatibility* requirements, governing interactions with other systems, subsystems, and components (most typically non-object-oriented ones) through constrained interfaces.

Object models and prototyping

Class designs specify internal workings of objects that hold across a range of commitments about active/passive object status, message-passing styles, and implementation details.

In order to guarantee this independence from resource and performance issues, class design activities need to be performed with reference to the most general abstract computational model compatible with that of the OOA framework. However, the design process requires a more concrete reference model for describing computational structure than does OOA.

This computational model is captured in the notion of an object-oriented supervisory kernel. This kernel may take a very general and powerful form for purposes of guiding class design. Moreover, the kernel may actually be *implemented.* An operational high-level kernel serves as an interpretive prototype simulator, useful for experimenting with preliminary class designs.

Prototype interpreters may be implemented using techniques common to simulation, production systems, and logic programming kernels. The basic idea is to create a single active computational agent that receives all events (object construction requests and other messages) in some kind of queue. When conditions allow, it pulls an event off the queue and performs the indicated actions on behalf of the associated objects. In this way, all conceptually active objects may be simulated passively, at the expense of creating an all-powerful super-object forever repeating the following steps:

- Take from the queue any event that has all of its triggering constraints satisfied, and process it:
  - If the event is an object construction, create a new (passive) object with the required initial states and attributes.
  - Else if it is an "elementary" state-change operation on a primitive object, then directly compute it.
  - Else place on the queue all component events listed in the body of the event.

This is actually just a variation on the computational structure implicit in common OOA object models, which are in turn variations of *actor* models [10]. But instead of empowering *all* objects to perform their own transitions and communicate with others, the single super-object behaves as if it were composed of all others, and communicates only with itself via the queue. There are numerous possible algorithmic improvements on this strategy. For example, triggering conditions become easier to deal with when each object's state is specially encoded ([17]), and/or multiple queues are employed, one per condition.

Such simulation systems serve multiple roles in design. Even if they are never built, they provide a concrete reference model helpful in liberating class design from system constraints. They also form the conceptual basis for further design activities described below.

Abstraction

Our structural continuity criteria require that abstract declarative OOA specifications be representable within the class design framework. This implies the need for an initial design step that translates these constructs into a form suitable for further transformation.

This step is a slightly more extreme version of the common OOD strategy of using *abstract classes* (also known as *types* when these are distinguished from classes) that specify attributes without mentioning internals. Concrete classes committing to particular representation and computation strategies may then be declared as subclasses (as is typical practice in C++ and smalltalk), or follow their own separate inheritance structure (as in POOL [1]), or even be designed within a classless prototype system (as in SELF [18]).

The continuity-based view is more extreme only in demanding greater coverage of constructs like triggering conditions, predicate-based invariants, and other features typical of declarative models. The notation must also fulfill the pragmatic need for variable precision of specification. Designers must be able to add information and constraints as they become known. Explicit incorporation of constraints within a computational context also allows partial automation of *design for testability* by facilitating definition of invariant checks, trace checks, and other self-testing strategies.

In practice, these features may be obtained either by extending programming languages with various extralinguistic annotation constructs (as in [7]), or by using a design language supporting simple translation of analysis constructs into those compatible with a broad range of OOPLs (as in [6]).

The translation of OOA models into design notations may be seen as the extraction of *specifications*, in the term's formal sense. Abstract class frameworks may contain information that is equivalently powerful to that found in classic specification systems like VDM [13]. But they represent constraints and assertions via constructs similar to those used in object-oriented design and programming proper. Such attributes greatly enhance the human factors of specification, and allow for routine adoption.

Compositional Design

Essentially all OO methods for establishing internal class properties are *compositional*. Objects obtain their static and dynamic properties by composing, delegating, inheriting, and coordinating those of other objects. The basic idea of compositional design is to build complex objects and operations out of simpler ones. For any given abstract class and/or operation,

- Other candidate components supporting at least some of the required properties are found.
- The declarative constraints of the complex entity are restated in terms of combinations or sequences of those of the identified components.
- A concrete version of the complex entity is then defined to access or employ the components.

OOA models already provide top-down decomposition of a system into abstract classes. Designers may define associated concrete classes in a more productive bottom-up fashion, in which nearly all complex entities are dealt with only when sets of plausible components are available.

Bottom-up composition reflects another application of transformational process criteria. Delays, mistakes, redundancies, and inessential demands are avoided by designing, using, reusing, and testing components before dealing with their clients. This is, of course, standard practice when the "component" is a concrete superclass of the one being designed. It would be unproductive to design a concrete subclass before its superclass.[1] This observation holds for composition more generally.

Compositional design is the most time consuming and creative aspect of the design process. Designers must choose among the multitude of available idioms to find those that best reflect the abstract specifications and other design goals. They must ensure that software quality criteria are met by paying attention to well-known pitfalls like aliasing and faulty dispatching. These, and numerous other issues and concerns fall beyond the process focus of this paper.

[1]Although the definition of a previously parentless class may suggest the opportunity to retrospectively design a new superclass.

Classes as Servers

In a compositional design framework, just about every class should be designed so as to be amenable for use as a component by others. This notion that *models* are to be transformed into software *components* places a different perspective on some standard quality notions. In particular, it focuses on the centrality of *design for reuse*. Component design involves:

- Design of (abstract) classes rather than one-shot objects.
- Design of class interfaces (accessors, methods) rather than of attributes and transitions.
- Standardization of interfaces, leading to the specification of families of interoperable subclasses and the creation of applications frameworks [12].
- Design of reliable interaction protocols, often supplanting pure event-driven models.
- Design of mechanisms and protocols for transmitting state information between cooperating objects.
- Design of service and "enslavement" protocols (access control, locking, etc.) so that objects may be used more predictably and reliably by others.
- Minimization of representational and informational demands upon *clients* (low coupling).

Classes as Clients

In analysis, arbitrarily complex objects and functionality may be considered as "primitive". In design, all but the most elementary objects are explicitly composite. Different sets of design tactics and quality criteria, in particular those revolving around *design with reuse*, stem from transformations needed to explicitly link the attributes and operations of one class with those of other, normally simpler ones it depends on. Many of these concerns are mirror images of those above:

- Black-box reuse. Minimization of representational and informational demands upon *servers*, thus allowing a broader range of concrete object types to be employed as components. This includes the use of capability-based (abstract) rather than implementation-based (concrete) specification of internally accessed objects; along with consequent reliance upon construction-time binding of servers.
- Reliance on implementation-independent *delegation* rather than concrete subclassing as the compositional technique of choice.
- Minimization of protocol demands upon servers.
- Design of coordination schemes (transactions, notifications, triggers) to maintain static and dynamic invariants both among components and between components and self.

Coordination

In design, nearly all classes are *both* clients and servers in an extended sense of the terms:
– During execution, *instances* of the described classes serve and are served by others *computationally.*
– During design, the *classes* themselves serve and are served by others *structurally.*

This correspondence between computational and design roles has wider implications. For example, in large systems the management of client-server relations often requires introduction of mediation strategies including relays, registries, task managers, master-slave configurations, and other idioms that partially centralize coordination of clients and servers. The net effect of these measures is to replace *identity* based communication with *role* or *capability* based strategies. The same principles hold true for the design process itself. Large class libraries should be organized via coordination schemes allowing designers to locate components by capability, by role (usage), and by dependency.

3.2 System Design

Previous steps define a system as a set of interacting objects requiring potentially unbounded resources, possessing unbounded lifetimes, operating under unbounded parallelism, and communicating through media of unbounded channel capacity, all as (conceptually) simulated by an interpretor. For projects with extremely liberal performance constraints, designers might be lucky enough to stop at this point. As noted in [17], a simple simulation-based design may sometimes suffice as a deliverable system.

Designers are scarcely ever this lucky. The goal of system design is to map objects on to managed software constructs that may be implemented using the resources at hand.

Clustering

Most system design activities revolve around the mapping of *objects* to *processes.*

The kernel interpreter system described above readily serves as a basis for systems-level decomposition. As noted, variations in the basic model apply both to "one process per object" and "one process per system" frameworks. Most systems fall in between, as collections of relatively coarse-grained processes, each housing a relatively large number of (passive) objects.

The resulting architecture may be designed by *clustering* objects within processes, each with the same overall structure as the above model, but with explicit *interfaces* allowing other objects to insert messages into request queues, as well as mechanisms (especially *proxies*) that ship remote requests to other clusters rather than locally queuing them.

There are no algorithmic criteria for clustering. On the one hand, performance considerations argue for a strategy that isolates resource intensive objects within their own processes, and clusters together other objects that heavily intercommunicate, or could share resources, or exploit capabilities found on the same physical machines. On the other hand, just about all other considerations (implementability, maintenance, etc.) argue for strategies that identify clusters with "large" semantically meaningful entities; i.e., subsystems and ensembles.

These two perspectives need not coincide, but they can be integrated under a uniform plan of attack. One method is *top-down* decomposition, in which the system is first conceptually viewed as occupying a single process. This view corresponds to the above simulation kernel. Sets of objects may then be identified and broken out as clusters using either performance-based or semantic criteria. This process may be repeated in order to split off additional clusters from the main "parent" and/or to further partition "child" clusters. The main attraction of top-down methods from a design-process and software-lifecycle view is that it becomes relatively easy to add partitions to accommodate new processors and to recombine children into a parent if necessary (possibly even dynamically during system execution).

Clustering criteria go hand-in-hand with the design of interprocess communication topologies. Centrally-arbitrated, broadcast-based, point-to-point, tree-structured, and other frameworks are all possible. Most are supported by contemporary software services, utilities, and management facilities.

Clustering may impact details of class designs. In most systems, embedded object *identities* (usually represented as addresses at the implementation level) may not be passed among processes. Thus, objects within different clusters cannot directly communicate. This often requires the introduction of additional layers of mediation and/or encapsulation. These issues become more extreme when the system heavily communicates with non-object-oriented foreign software.

Because clustering remains something of a black art, it is very useful to implement versions of prototype interpretors that simulate expected delays and loads resulting from tentative clusterings.

Resource Management

The class design level deals with "conceptual" objects. The system design phase deals with "real" ones that occupy physical resources. Clusters form a natural focal point for object management.

Each cluster *houses* and manages a set of objects. The design of internal cluster management facilities will necessarily rely on the availability and/or construction of software services that help allocate and deallocate storage, manage persistence, control concurrency, and route messages. Especially if persistence is supported using non-integrated and/or non-OO database systems, database design is likely to form a central role in this subphase.

Inter-cluster object management design involves the introduction of process control and management facilities, inter-cluster communication, fault-tolerance, and communication with foreign software.

3.3 Program Design

The above activities describe classes and their mappings to resources, but do not fully determine the internal workings of the individual processes and the programs that govern them.

Within-Cluster Transformations

Even though they are partially tied to implementation details, such programming matters remain "design issues" because of the performance obligations of the design process. Designs must ensure that stated and implicit[2] performance goals are architecturally feasible.

There is ample room for improvement. Even a large set of interacting clusters can possess significant performance problems. Previous design steps

[2] An ATM machine that "goes away" for three minutes after taking in an ATM card is obviously unacceptable, while the requirement writers may not have bothered detailing their intentions in this matter.

assume that cluster agents are merely interpreters. They receive all events emanating from their component objects or other clusters, and then either execute them or forward them to other clusters. Most of this interpretation overhead can be bypassed by *sequentializing* within-cluster communication. Interactions between objects residing in a single cluster may be recast using standard procedure calls and related constructs that efficiently operate within a single address space and/or thread of control. These efforts may be accompanied by transformations that introduce error protocols controlling behavior when messages are received while objects are in inappropriate states. This bypasses the need for triggering queues that protect objects from such invasions. Memory management, dispatching, and related resource overhead may be similarly localized and streamlined.

Optimization

Other classic performance tuning strategies are easily accommodated. The original design of interoperable classes allows designers and/or implementors to incrementally improve performance without disrupting overall structure or functionality by swapping in better classes, algorithms, and communications protocols for existing ones. Prototypes may be run to tentatively evaluate questionable components before committing to other details of the final implementation process.

Most fine-grained efficiency matters are bound to the quality of the implementation. In these cases, the best a designer can do is to ensure that clever implementations are at least possible by maintaining interoperability principles throughout the design phase. Also, by working with implementors attempting to diagnose and repair performance failures, designers help maintain design integrity throughout the implementation phase.

4 A Prototypical Design Process

By investigating the role of design within a tripartite software development model, and by analyzing the resulting activities and constraints, we are led to a transformational view of the design process. This model is amenable to a great deal of variation and refinement. Assuming specification of an object model/kernel to be used as a framework for specifying representation and computation, it involves the following steps and activities as minimal components:

1. Class Design
   - (a) Translation of OOA models into corresponding abstract design constructs.
   - (b) Bottom-up compositional design of concrete components.
   - (c) (Optional) Prototyping, restructuring, refinement.
2. System Design
   - (a) Transformation of a master interpreter into as many clusters as necessary or desirable to match resource constraints.
   - (b) Design of resource management facilities within and between clusters.
   - (c) (Optional) Prototyping, restructuring, refinement.
3. Program Design
   - (a) Transformation of event interpretation into procedural message passing, along with other conversions into OO programming constructs.
   - (b) Identification and replacement of poorly performing components with better ones.
   - (c) (Optional) Prototyping, restructuring, refinement.

The basic framework admits several variations. Some of the more obvious ones include:

- Class design, especially, may be interleaved with parts of OOA by dealing with subsystems or other coarse-grained components as they become available.
- Program design and implementation activities may be similarly pipelined.
- Class design activities may be subdivided among designers working concurrently and semi-independently.
- Design of the general form, policies, and internals of cluster agents may proceed concurrently with class design.
- The reference object model may be severely constrained when the system is required to be implemented as a single program written in a single language. In this case, the language's own dispatching kernel may be substituted for the interpreter.

The constituent steps bear similarities to those presented in other prescriptive accounts of design. For example, Rumbaugh et al [16] distinguish class from system design, and Coad & Yourdan [3] describe several subdivisions of design that may be placed within this categorization. Booch's [2] account of design includes many steps that we would consider to be parts of OOA, but otherwise describes similar activities.

The present account differs from these mainly by virtue of its strong emphasis on structural continuity, and its ordering of steps to reflect logical dependencies. These lead to a model that both accommodates and rationalizes methods that are widely held to be good OOD practices. These include the routine use of abstract classes, prototyping, bottom-up compositional design, reliance on a dispatching kernel to guide systems development, and planning for optimization measures through interoperable design techniques before committing to the optimizations themselves.

Perhaps the most novel aspect of our framework is its reliance on "active" object models mirroring underlying OOA assumptions throughout much of the design phase. Our transformational criteria lead to the notion that premature sequentialization is no more defensible within OOD than is premature optimization.

5 Example

Due to limited space, we will sketch the application of the different transformations on a toy example that provides only a few glimpses of key points and issues. We discuss some requirements, and then provide little snippets of the analysis and design. We dwell a bit more on the analysis side in order to illustrate our claims that OOA methods can and should provide the bulk of the high-level descriptive information.

5.1 Requirements

This is a pseudo physics problem simulating a chamber in which particles are injected at side A and diffuse out at side B (Figure 1).

The injection rate at A is constant. The diffusion rate at B is a function of the local density. There are two kind of particles: lightweight and heavyweight. The latter particles decay spontaneously into the former at a certain rate. A simulation is used to study the flow and densities of the particles in the chamber.

Both the space in the chamber and the time are discrete. The cells of the chamber form the following disjoint collections: *Walls* at the top, at the bottom and the two vertical pieces, *Injection cells* at the A-side, *Diffusion cells* at the B-side, and *Interior cells.*

For every non-wall cell, we maintain the densities of the two types of particles, and the flux vectors of the two types of particles. The simulation is to be run iteratively until the flux doesn't change significantly in any cell.

The "behavior" of an interior cell at time step N is split up in three phases:

Account for Decay

(1) Decrease the heavyweight density due to decay;

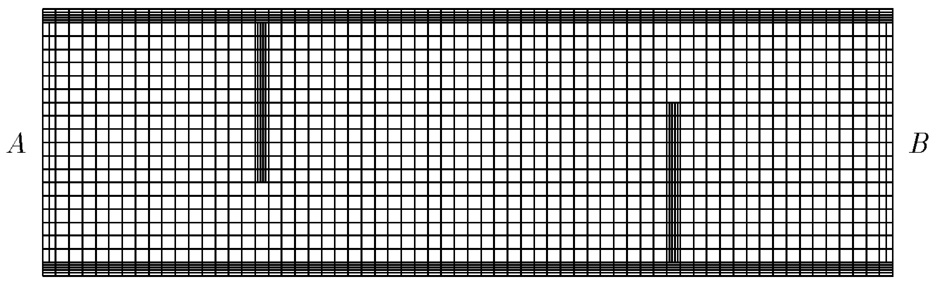


Figure 1: The Chamber

(2) Increase the lightweight density due to the decay.

Determine Flux
Determine the flux vectors, for each type of particle, in the north, east, south and west directions as a function of density differences in each, where a flux vector is zero in the direction of an adjacent cell when the density in the adjacent cell is higher or it is a wall cell.

Adjust Density
(1) For heavyweight particles, increase the density of adjacent cells in proportion to the flux into them; i.e. if a flux into a cell is f then increase the density there with $c * f$;
(2) Decrease the local heavyweight density by what has flown out;
(3) Do the same for the lightweight particles;
(4) Increment N.

The injection cells behave like interior cells except that fixed amounts of particles are added through their west "wall". Diffusion cells behave like interior cells except that particles disappear in the east "wall" as a function of their densities.

5.2 Analysis

We sketch first a few classes. In Figure 2, we have introduced the class *Cell* with has four optional attributes named *north*, *east*, *south* and *west*. The optionality is indicated by the cardinality constraint $[0, 1]$. The fifth attribute indicates whether the cell belongs to a wall or not. This attribute allows us to introduce two subclasses $Wall$ and $NonWall$ in which the *wall*? attribute value is frozen into *yes* and *no* respectively. We can introduce a subclass of $Wall$ called $TopWall$ in which we know for sure that the *north* attribute is absent. $NonWall$ cells have guaranteed *north* and *south* attributes (Figure 3).

The upward pointing vector represents an inheritance link. We have given $NonWall$ the additional attributes *heavy_density* and *light_density* to keep track of the densities of the different particles; and *flux* of type *Flux* (possessing in turn attributes *flux_x* and *flux_y*) as the resultant of combining the fluxes in the x and y directions.

We can introduce $ACell$, $BCell$ and $IntCell$ as disjoint subclasses of $NonWall$ by making commitments about the still variable occurrence of *east* and *west* attributes. For instance, $ACell$'s do not have a *west* attribute while interior $IntCell$'s have all four attributes that refer to adjacent cells.

We proceed by giving a high level view of the transition network of an $IntCell$ (Figure 4). One of the transitions is shown in Figure 5.

The *Chamber* itself forms a natural ensemble class, representing the aggregation of cells and their interconnections within a (single) subsystem.

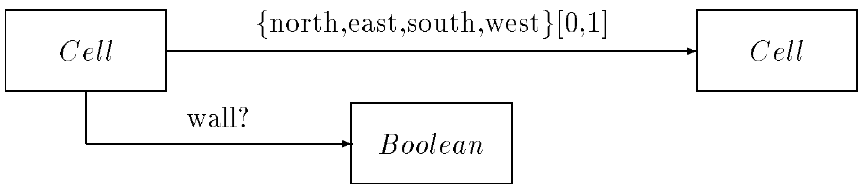


Figure 2: Class Cell

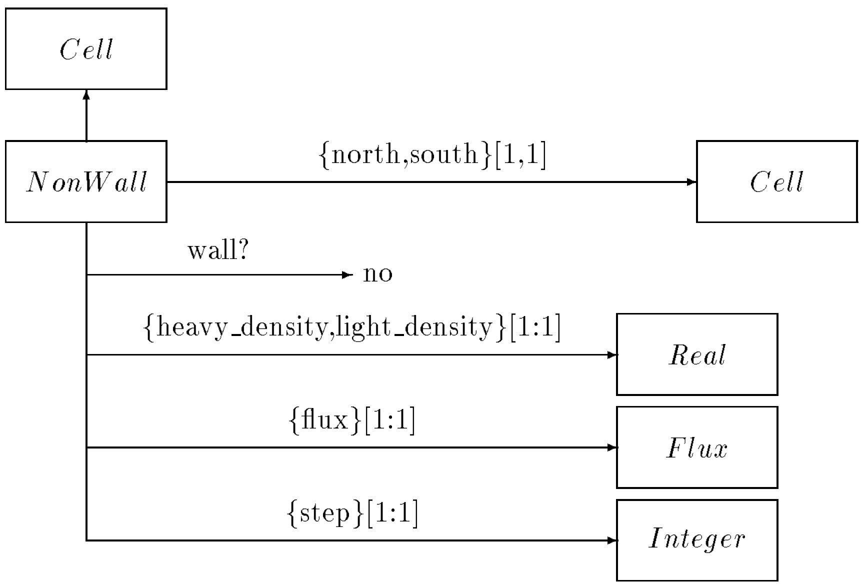


Figure 3: Class NonWallCell

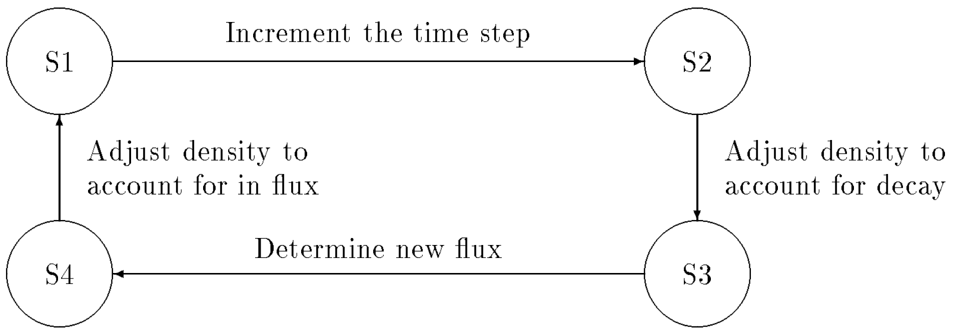
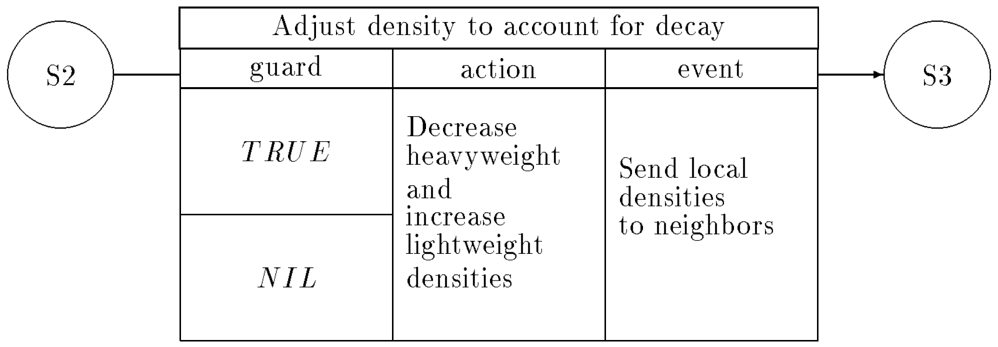

S1
Increment the time step
S2
Adjust density to account for decay
S3
Determine new flux
S4
Adjust density to account for in flux

Figure 4: IntCell Transitions



S2
Adjust density to account for decay
guard
action
event
TRUE
NIL
Decrease heavyweight and increase lightweight densities
Send local densities to neighbors
S3

Figure 5: Detail of an IntCell transition

5.3 Class Design

The output of the analysis phase must be transformed into a set of abstract class designs. For example, class *Cell* might be translated as:

```
abstract class Cell
 isWall: bool;
 optional north: Cell;
  ...
end
```

The *NonWallCell* subclass adds both attributes (densities) and behaviors (adjusting for decay):

```
abstract class NonWallCell is Cell
 heavyDensity: real;
 op adjustForDecay
   ==> ..., heavyDensity' =
        heavyDensity * decayFactor;
  ...
end
```

This may be reified into a corresponding concrete class by finding components that both represent the value properties and support the necessary update operations. In this case, simple number *objects* suffice. The value attributes reflect *states* of component objects, and operations are performed via delegation to them:

```
concrete NonWallCellImp is NonWallCell
 component _heavyDens: RealNumber;
 heavyDensity  { _heavyDens.state }
 op adjustForDecay  { ...;
    _heavyDens.multiplyBy(decayFactor); }
  ...
end
```

Of course, we do not claim that translation of property and effect specifications always transform this transparently into components and executable actions.

The overall design is just complex enough to contemplate introduction of coordination strategies to replace the strictly local state and event driven transitions described in the OOA models. At the analysis level, the *Chamber* was a rather passive class, merely "keeping track" of the cells. This may be transformed into an active coordinator in order to simplify and regularize communication. For illustration, we'll choose an extreme measure, *synchronous control*, in which the cells are treated as "slaves", and are stepped through actions by the chamber object, that now serves as a "master":

```
class Chamber
  ...
  ints : SET<NonWallCell>
  op interiorStep {
    ints.applyToAll(incrementStep);
    ints.applyToAll(adjustForDecay);
    ints.applyToAll(getDensityFromNbrs);
    ... }
end
```

Among other refinements, this scheme leads to the (re)definition of `Cell` operation mechanics (e.g., `adjustForDecay`). `Cell` operations must "call back" senders in order to notify them when transitions are complete. The `applyToAll` operation may then use scatter/gather idioms to control synchronization. Of course, any of several other coordination schemes might have worked as well or better.

5.4 System Design

The requirements did not mention compute resource constraints. Suppose however, that after prodding our hypothetical customer we learn that we should use two compute servers. Thus we can have two clusters. Since the great majority of our objects are instances of `IntCell` we will have to distribute them over the two compute servers.

To minimize communication between the clusters we can split the chamber into a left side and a right side. We can define a corresponding `Side` class, and two such objects `leftSide` and `rightSide` that will serve as the basis of the clusters. They will construct and manage the cells, and mediate communication among cells in each half.

The two `Sides` are (remote) components of a master chamber object residing on either of the machines, perhaps as a stand-alone *process* on one of them. The sides do need to interact. Cells along the borders need to propagate densities to each other. This requires introduction of additional mediation capabilities in the interface of each `Side`, along with all of the other modifications that this in turn entails. One solution is (1) to assign each cell $row, column$ attributes that serve as pseudo-object-IDs, (2) to re-express all neighborhood relations using maps from $row, column$ to cells (i.e., a *matrix* representation), and (3) to send intercluster updates one-by-one through the remote message `neighborDens(row, col, value)` applied to all relevant cells at the end of a suitably modified `interiorStep` operation (4) to create a `BorderCell` class to facilitate identification and expression of cross-cluster processing.

Further suppose that execution is expected to be very long-lived, requiring persistence support to deal with situations in which the cluster objects "crash" during a simulation. Just about any kind of persistence mechanism could be employed for this design. The use of a relational database is a simple and practical alternative since all cell state attributes are simple numerical values (including $row, column$ indices serving as pseudo-IDs). Different kinds of cells might correspond to different relational database tables. A checkpoint/rollback scheme could be added to the `Side` class. Snapshots of states may be stored persistently through the relational database after every k iterations. The `chamber` (or a clone thereof, if *it* crashes) would then manage crash recovery.

5.5 Program Design

At this stage, we can still (barely) perceive the set of objects in a cluster as independent processes. However, all local communication is readily transformable into unguarded blocking procedures that need not be mediated by queues and other active-object mechanics. With only minor refinements, all within-cluster messages may be converted into simple procedure (method) calls. Each program does still need an active-object interface in order to accept update and synchronization messages from the other `side` and the `chamber`, as well as to communicate with database services.

There's still a lot of room for incremental improvement. For example, there is no computational reason for each `Cell` object to explicitly maintain a `step` attribute. A single representation in each `side` would suffice.

5.6 Implementation

There are two major facets in the implementation of this design, *process*-level and *program*-level.

If this were to implemented in C++, process-level implementation matters might be addressed using tools like CORBA [14] to generate cluster process interfaces, proxies, and interprocess communication mechanisms. Additional foreign-system veneers and mechanisms need to be employed and/or constructed to implement communication with the chosen database service.

The C++ code for classes needed within the `Side` clusters could be produced through straightforward translation of the refined versions of `Cell`, etc., resulting from the program design phase. These might first be adapted so as to take advantage of pre-existing `Matrix`, `Set`, and other components.

6 Summary

We have described how OO design can be split up in distinct and ordered phases. Each phase focuses on one item in the list: class design on functional requirements, system design on resource requirements, and program design on performance requirements. We have illustrated the three transformation phases with an example.

By satisfying these requirements sequentially and utilizing the supervisory kernel model, we can obtain an executable high-level design (provided that an interpreter exists for the class design language). Given this proviso and to the extent that OO analysis is a systematic endeavor, we have outlined a disciplined way to obtain early prototypes.

By splitting design into different phases and outlining different sub activities in these phases, we hope to clarify the OO design process and provide guidance to those developing OOD CASE tools.

Acknowledgments

We have benefited from discussions with Martin Griss and Desmond D'Souza. We appreciate the feedback that we obtained on an earlier version from John Burnham and Rajendra Raj.

References

[1] America, P., A Parallel Object-Oriented Language with Inheritance and Subtyping, *Proceedings OOPSLA '90*, 1990.

[2] Booch, G., *Object-Oriented Design*, Benjamin Cummings, 1991.

[3] Coad, P. & E. Yourdon, *Object-Oriented Design*, Yourdon Press Computing Series, 1991.

[4] de Champeaux, D. & P. Faure, A Comparative Study of Object-Oriented Analysis Methods, To appear in *JOOP*, 1992 March/ April.

[5] de Champeaux, D., Object-Oriented Analysis and Top-Down Software Development, in Pierre America (Ed), *Proceedings of ECOOP '91.* Lecture Notes in Computer Science, no 512, Springer Verlag, pp 360-376, 1991.

[6] de Champeaux, D.,D. Lea, & P. Faure, *Object-Oriented System Development.* (Manuscript in preparation.)

[7] Cline, M. P. & D. Lea, The Behavior of C++ Classes, *Proceedings, Symposium on Object-Oriented Programming Emphasizing Practical Applications*, Marist NY, 1990.

[8] Harel, D., Statecharts: A Visual Formalism for Complex Systems, in *Science of Computer Programming* vol 8, pp 231-274, 1987.

[9] Henderson-Sellers, B, & J. Edwards, The Object-Oriented Systems Lifecyle, *Communications of the ACM*, vol 33, pp 142-159, 1990.

[10] Hewitt, C., P. Bishop & R. Steiger, A Universal Modular ACTOR Formalism for AI, in Third International Joint Conference on Artificial Intelligence, pp 235-245, Stanford University, 1973 August.

[11] Jacobson, I., Object-Oriented Development in an Industrial Environment, in *Proceedings OOPSLA'87*, pp 183-191, 1987.

[12] Johnson, R., & V. Russo, Reusing Object-Oriented Designs, University of Illinois Technical Report UIUCDCS 91-1696, 1991.

[13] Jones, C., *Systematic Software Development Using VDM*, Prentice Hall International, 1986.

[14] Object Management Group, *Common Object Request Broker Architecture and Specification*, Document 91.12.1, Object Management Group, 1991.

[15] Reynolds, G.W., *Informations Systems for Managers* West Publishing, 1988.

[16] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[17] Tidwell, B.K., Object-Oriented Analysis State Controlled Implementation, Proceedings Workshop on Object-Oriented (Domain) Analysis, OOPSLA, 1991.

[18] Ungar, D., & R. Smith, SELF: The power of simplicity, in *Proceedings OOPSLA'87*, pp 227-241, 1987.

[19] Wirfs-Brock, R., B. Wilkerson & L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[20] Yourdon, E. & L. Constantine *Structured Design*, Prentice Hall, 1979.