Notes

Following is a short chapter-by-chapter account of other published work that is relevant to each topic and has, in some way, influenced our work. All these works are listed in the References. Many of them are good supplementary or follow-up reading. Another good place to find follow-up reading is at www.catalysis.org.

Part I: Overview

Catalysis has been influenced by many sources, including both authors' prior work and existing OO methods. Some well-known work on object-oriented analysis and design methods are listed here. Fusion [Colman93] was one of the first methods that had a clear process and separation of concerns along with a strong foundation in formal methods. Syntropy [Cook94] made a clear distinction between essential, system, and software design models and had strong semantics for the authors' state chart-centric behavior models; their work on viewpoints was very insightful, although it was not made central to their approach. The OORAM approach [Reen95] pioneered role-centric and collaboration-based modeling and developed ideas of composition and abstraction based on it.

The work of Grady Booch [Booch91, Booch96] and the OMT book [Rumbaugh91] were influential and became market leaders; they brought, respectively, a pragmatic description of design, and a good information-centered object-modeling approach and state charts. The use case approach of Jacobsen et al. [Jacobsen92] first drew attention to the glaring lack of focus on user tasks in OO methods; [Jacobsen94] is a more readable account of the main ideas. ROOM [Selic94] offers an architecture-centric approach to modeling using executable state charts and makes a component-port-connector model central to the approach.

de Champeaux and Lea [de Champeaux93] provide a wealth of insights into the subtleties of good object-oriented design. The Shlaer–Mellor approach in [Shlaer88] and [Shlaer92], an early object-modeling approach, was focused on a normalized data model and executable communicating state-machine models for analysis and a translation-driven approach to architectural design. A comparison of OO methods is in [D'Souza93]. Martin Fowler [Fowler98] provides a good review of the UML notation, and the OMG Web site provides reference material [UML].

Chapter 1 A Tour of Catalysis

Earlier writeups and presentations on the Catalysis method include [D'Souza95, 96a, 96b].

Part II: Modeling with Objects

The useful separation of static, dynamic, and interactive modeling of objects has been recognized in formal approaches for some time, although it has not been exploited in mainstream OO methods. Among OO methods, the idea was used in Fusion [Col93] and in a less obvious way in Syntropy [Cook94].

Chapter 2 Static Models: Object Attributes and Invariants

The basic idea of modeling an abstraction of state regardless of concrete data implementation has pristine lineage, including the likes of formal methods such as Z [Spivey86] and VDM [Jones86], as well as more recent OO methods such as Syntropy and Fusion. Parameterized attributes are a simple variation of the many kinds of functions and relations that are predefined in Z or that can be user-defined in VDM; these variations can always be explicitly defined in Catalysis in a generic package.

The observation that a formal model is only a collection of symbols and could equally represent a physical world or software—and the need for a separate dictionary that maps the formal model to the things it represents—is well known to model theorists. The ideas are very well described in [Jackson96].

Logic that can cope with an undefined value—three-valued logic—comes from [Cheng91]. Flat sets were used to navigate object models in [D'Souza94], defined in [Wills97], and subsequently taken up by OCL.

Chapter 3 Behavior Models: Object Types and Operations

The concept of type as object behavior, regardless of implementation, is well known to those working in semantics of OO languages [America90] and in formal approaches such as [Wills91]. Using pre- and postconditions to specify an operation in terms of an abstract model of state goes back a long way, to work by Hoare and Dijsktra [Hoare94]. More current usage in the context of OO methods includes Fusion and Syntropy and, in OO languages, Eiffel [Meyer88]. Permitting and composing multiple specs for an action is based on [Wills91].

Our interpretation of a static type model of a component as being what the component needs to know about the world and happenings around it, and the way we relate a software system to a refinement of a business model, are fundamental to traceability from business to code.

Using snapshots to illustrate actions, conceptualizing the type model as a generalization of legal snapshots, treating attribute types and associations as defining invariants, the

metaphor of the "film strip," and the relation between state charts, action specs, and snapshots are described in [D'Souza94].

The idea of joint actions between stateful objects was inspired by the work done in Disco [Kurki-Suonio90]. These authors describe how joint actions provide powerful abstractions, provide the first precise semantics for state charts in the context of object modeling, and show how entire object behavior models might be refined.

Using invariants to factor common static rules that apply to all pre/post conditions comes from formal methods such as Z and VDM. Our use of "derived specifications" is a simplified approach to what formalists might call "theorems" [Gries96]. Effect invariants—the idea that a dynamic constraint can apply to all actions—is related to a more powerful formula of temporal logic [Manna92]; temporal logic can sometimes greatly simplify a difficult specification, and we have used the more-general mechanisms on occasion.

The two fundamentally different ways of writing multiple specs (pre/post and pre=>post) were inspired by the work of D. Jackson [Jackson95]. He points out the importance of having specs from different views that, when composed, do not do so "monotonically"; we realized we could support both styles with a simple convention on how the specs should be written, while using a uniform rule about composition.

The idea of subtypes as subsets based on external behavior, independent of implementation, independent of classes, and with specs that simply extend (without overriding) is common in type theories, such as POOL [America90].

State types are discussed in D'Souza [D'Souza94]. Our use of a predicate to "classify" an object provides some convenient modeling capabilities.

Introducing convenience state modeling elements to simplify a specification is well known in formal methods; an analogous technique is used frequently by good OO designers in their designs and code. D. Jackson [Jackson95] makes a strong case for allowing different viewpoints to use different models of the state information; our use of packages carries this a step further.

Factoring common postconditions is done routinely in Z using schema composition. Our separation of actions and effects is meant to distinguish these convenience functions from the actual operations that must be supported. We allow "gray-box" specifications, in which certain aspects of the internal behavior, such as call sequences, can be exposed as a part of the specs. In the OO world, related work was done by Helm et al. [Helm90].

Quoting action specs is described in [Wills91] and elsewhere. The distinction between spec types and design types has long been known in formal methods; our highlighting it, and factoring effects to specification types, is somewhat unique among OO and CBD methods.

An excellent treatise on the use of state charts as a primary specification vehicle with objects is [Cook94].

The subjective model of containment is unique to our approach. It is related to the idea of subject-oriented programming [Harrison93], which is focused more in composing OO implementations.

A discussion of types and classes in programming languages is in [D'Souza97]. Specification aspects of classes and types is discussed in [Wills95b] and [D'Souza96b].

Using type models to precisely document the interface between a superclass and its subclasses is explained in [D'Souza96a].

Chapter 4 Interaction Models: Use Cases, Actions, and Collaborations

The idea of a joint action as an abstraction of many possible interaction protocols was inspired by the work of Disco [Kurki-Suonio90]. We treat localized actions as a degenerate case of joint actions. Clarifying the meaning of *parameters* (and the corresponding rules and documentation for refinement) in the context of objects, and the treatment of use cases, we believe to be unique to our approach. Traditional use cases are described by Jacobsen in [Jacobsen92].

Specifying concurrent actions using rely and guarantee is based on the work of Cliff Jones [Jones93].

Practical guidelines on how to deal with concurrency in a design—covering issues of interference (safety) and cooperation (liveness)—and higher-level concurrency design constructs are included in [Lea96].

Treating collaborations as first-class design entities has been done by Rom Casselman and, more informally, in the work of [Wirfs-Brock90]. OORAM [Reenskaug95] was the first OO method to make this a central way of thinking. The OORAM method is based almost exclusively on the idea of role models; it is closely related to collaborations, with an added element of implied object-identity constraints on the diagrams. Using collaborations as a scoping construct for invariants, distinguishing the internal and external actions for each collaboration it plays a role in, is unique to our approach.

The Ph.D. thesis of D. Beringer [Beringer97] generalizes the concept of scenarios; in that context, it has a good discussion of refinement (action and object refinement), good ideas about the systematic depiction of scenario types, and a brief, insightful, discussion of the need for more-effective state abstraction.

Chapter 5 Effective Documentation

Our overall approach to clear interwoven documentation, rather than the production of either pictures or formal mathematics, is based on Knuth's ideas in *Literate Programming* [Knuth84] and on the style espoused by Z. Permitting multiple appearances of an element on multiple drawings, clearly separating models of business from software specifications and from their internal implementations, would appear to be plain common sense; yet is not spelled out often enough. The high value we place on clear vocabulary and precise definintions of terms comes from project experience. Structuring documentation around package structure, and the emphasis on small fragments of models interleaved with explanatory prose, has not been emphasized in popular methods.

Part III: Factoring Models and Designs

The importance of careful factoring and abstracting away details underlies all good OO programs. Popular methods do not provide well-defined support for composition, and for factoring medium- to large-grained units.

Chapter 6 Abstraction, Refinement, and Testing

Almost every method, from the most dense and formal ones to those based on the most loosely defined sketches and drawings, stakes a claim to being "abstract." The formal ones have a precise idea of what kinds of things are omitted in the abstraction and exactly what it means to correctly implement that abstraction; they also share our idea of a conformance relation with a justification. Refinement is a well-established technique in the formal methods community. The ideas of protocol refinement and data reification (model or state refinement) have been separately but quite thoroughly worked out elsewhere.

Model and operation refinement ideas go back to VDM and Z and to the work of Hoare and Dijkstra [Hoare94]. Joint actions and state chart refinement are addressed thoroughly in Disco and ADJ; Syntropy [Cook94] has an excellent account of a subtyping-based refinement with state charts. Using specifications for testing and debugging implementations has been long practiced by good software developers; [Meyer88] has a good account of it and a programming language that directly supports some of it.

Refinement and subtyping in OO implementations are discussed in several recent works, including those by [D'Souza96b], [Utting92], [Wills93, 96a, 97a], and [Mikhajlova97]. A highly formal approach, suitable for very high integrity systems, is described by Kevin Lano [Lano95].

Doug Lea defines interaction protocols for open object systems, and their refinement rules, in [Lea95].

Michael Jackson and Pamela Zave [Jackson96a] clearly point out the difference between effects and operations; their framework is not one of refinement. Rather, it is centered on how to structure and compose abstract state changing actions (using Z) and finer-grained protocols (using state machines or other mechanisms) to be confident about the results. Many of those ideas fit into the rules for documenting a refinement.

The co-author's book [Wills91] has a good account of refinement based on model and operation abstraction and subtyping. The need for collaborations as first-class entities and collaboration refinements beyond subtyping are described in [D'Souza96a].

We believe the basic forms of refinement to OO methods and the strong link to systematic testing are part of our contribution.

Chapter 7 Using Packages

Packages as groupings for pieces of models, designs, or specs, in the sense we use in Catalysis, go back to the idea of mathematical theories and are exemplified by the specification language of Larch [Guttag90] and Mural's specification tool [Mural91]. The idea of

extending type information in theory extensions comes from them, as does the definition of the language semantics from the ground up. [Wills91] describes how these theories, which he calls "capsules," can be used to separate specifications from code and to provide disciplined forms of class inheritance, method overrides, and code updates.

The more straightforward ideas of packages for grouping classes can be found in Eiffel's "clusters" [Meyer88], Booch's "categories" [Booch96], and Wirfs-Brock's "subsystems" [Wirfs-Brock90].

Treating all development artifacts as a part of a package would seem to be common sense, yet surprisingly few projects practice it and surprisingly few methods require it. Knuth's Web is based on strongly integrating code and documentation but does not address modeling or specification.

Java packages are clearly documented in [Gosling96]. Ways to use them to separate interfaces, collaborations, and classes, based on Catalysis techniques, are described in [D'Souza96a].

Our treatment of packages as units of version control, configuration management, and builds is similar to Sun's Forest project [Jordon97], which scales Java packages to very large-scale software development. The idea of being able to say more about any imported modeling element is supported in code in a limited way by the team-working product Envy for Smalltalk; it allows different aspects of the same implementation class to be defined in different "applications." It is commonly used in formal methods based on traits or theories [Guttag90]. The ideas behind this are discussed in [Wills91] and [Wills96b].

Chapter 8 Composing Models and Specifications

Composition is an age-old idea in software specification and development; any method must have clear rules about the outcome of combining designs, specifications, or code. Popular OO methods do not address this issue in a meaningful way. Exceptions are Disco's composition of entire models [Kurth90], Syntropy's subtyping and viewpoints [Cook94], and OORAM's role-model synthesis [Reenskang95]. Related work in non-OO approaches are plentiful: Z, Unity, and so on.

Our rules for intersecting types are based on well-known rules of conformance [Liskov91, Meyer88]; those for composing separate views in a nonmonotonic way were influenced by the work of D. Jackson [Jackson95] and Z [Spivey86].

Few OO approaches seriously support the specification of exceptions; yet exceptions often form the most complicated part of a complex specification and its implementation. Our approach to describing exceptions is based on the work on Assertion Definition Language [ADL], adapted to fit with our need to compose multiple specifications that can include exception conditions. Our integration of exceptions with use cases and action refinement we believe to be unique.

Chapter 9 Model Frameworks and Template Packages

The ideas of frameworks are akin to how Larch defines the importing of traits with renaming [Guttag90] and are related to the work in FOOPS [Goguen90]. Some of the earliest OO work related to our collaborations comes from Helm et al. [Helm90].

Our initial ideas of frameworks as generic collaborations in Catalysis were written up in [D'Souza95a]. Broader use of model templates using packages in Catalysis is described in [Wills96b].

The confusion between frameworks, generics, and subtypes is present even in the UML [UML].

The ideas of a framework at the level of OO implementation are well described in Taligent [Taligent94] and Johnson [Johnson92]. An excellent example of using the framework-like facility of C++ templates is in the Standard Template Library [Saini96]. At the level of design patterns, the historic GOF book [Gamma95] provides plenty of motivation for a specification construct like frameworks. Martin Fowler's book *Analysis Patterns* [Fowler97] elevates that need to a domain-specific level. Shlaer and Mellor [Shlaer92] use an idea akin to frameworks in their approach to recursive design, although they do not generalize it beyond that.

Framework provisions—constraints on properties an acceptable substitution must have when applying the framework—are used in Larch, FOOP, and even in well-documented implementations of the STL [STL96].

Using frameworks to define the meaning of the very modeling constructs themselves—and even to define and encapsulate known inference rules—is very similar to the approach in Larch [Guttag90]. Their application to Catalysis modeling constructs, UML stereotype-based extension, and new modeling constructs and notations is described in [D'Souza97a].

Frameworks seem quite related to some of the ideas of aspect-oriented programming [Gregor97], but we are not sure how. In particular, we would compose the models from multiple framework applications in which they would use an "aspect-weaver" program to combine implementation code.

A package might define a set of relevant object types with their states and interactions, but it also explicitly defines (or imports) the very rules that are used to make deductions about things. This arrangement can be useful in "agent-based" models; an economic simulation of a market might explicitly model the theory each company has of its competitors' actions, including the inference rules it uses. With good tool support, you might then do simulations and explore different hypotheses within a package.

Part IV: Implementation by Assembly

Many books have been written about good object-oriented design and implementation. Component-based development and assembly have only very recently become hot topics. Component and connector technology, effective pluggable design, and a coherent architecture together enable implementation by assembly.

Chapter 10 Components and Connectors

Szyperski [Szyperski97] has several clear definitions of component technology, and we adapted several definitions from that book.

Using components and connectors to define architectural styles was introduced in [Allen94] and elaborated in [Shaw96]. Component connectors appear in a number of patterns in [PLoP]. Our definition of them in terms of actions appeared in [Wills97a].

Among current component technologies, CORBA technology is described at the OMG Web site [CORBA]; COM at Microsoft's Web site [COM]; and JavaBeans at the JavaBeans Web site [EJB].

The approach to treating entire systems and subsystems as objects is something we have been practicing for many years.

Chapter 11 Reuse and Pluggable Design: Frameworks in Code

A book by [Wong93] gives a clear account of pluggable designs. The idea can be traced back through the development of the Smalltalk class library. [D'Souza96a] describes documenting the interface of up-calls and down-calls using type models. [Gamma95] has an excellent discussion of decoupling patterns, as does [PLoP].

Chapter 12 Architecture

Some of the early work on systematic description of software architecture was done by Shaw and Garlan [Shaw96b]. A good description of the breadth of issues involved in software architecture, which we have used and adapted, is in [Bass98]. Our informal definition of architecture was coined in frustration by D'Souza (we believe).

Architecture Description Languages (ADLs) have been receiving much study recently. Important work includes the ABLE project (with the language WRIGHT) [Able] and Rapide [Luckham95].

ROOM [Selic94] is one of the few OO methods that can be said to be based on an architecture definition (graphical) language. Shlaer and Mellor [Shlaer92] have long advocated a translation-based approach to architecture: You define the translation patterns and rules and generate the design from the analysis models.

Doug Lea's work [Lea95] addresses several architectural issues that arise in the building of object-oriented systems. And [Gamma95] is important in bringing pragmatic and proven design patterns to their rightful place in software architecture. Our use of frameworks for components, and connectors, and for other patterns and rules that describe architecture, provide additional expressiveness.

Part V: How to Apply Catalysis

Like the rest of this book, the processes and techniques have evolved from practical application and from the process definitions of many others. Catalysis emphasizes the systematic use of package structure, separation of package artifacts from the process and routes

that populate those artifacts, refinement from business models to code, frameworks at all levels, precise use cases accompanied by clear supporting type models, and multiple views that can be composed.

Chapter 13 Process Overview

Good general project guidelines can be found in [Cockburn98] and [Goldberg95]. RAD is discussed in [McConnell97] and DSDM (an excellent RAD approach) is discussed in [Stapleton97].

Chapter 14 How to Build a Business Model

Modeling of business processes is often based on a flow-like notation. Examples include the activity diagram of UML [UML], role-activity diagrams [Holt83], the process flow (and related relationship and organization structure) of Rummler and Brache [Rummler90], and functions and information in IDEF0 [IDEF093]. These would need a semantic connection to our other, more-precise modeling techniques.

Martin Fowler and Craig Larman have each written good books on patterns for modeling [Fowler97, Larman98].

Chapter 15 How to Specify a Component

The vanilla process is like that followed by many authors such as [Cook94] and [Colman94]. The use case focus is related to Jackson's [Jackson92], although our emphasis on refinement is different.

Chapter 16 How to Implement a Component

Some good guidelines for design are in [Reil97], [Gamma95], and [Taligent94].