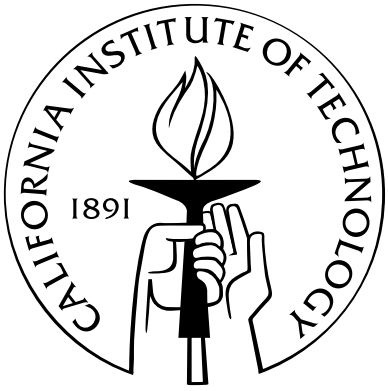
CS-TR-98-08

The Specification of Dynamic Distributed Component Systems

Thesis by
Joseph R. Kiniry

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science



California Institute of Technology
Pasadena, California

1998
(Submitted May 29, 1998)

Acknowledgements

Special thanks go to my wonderful advisor, Dr. K. Mani Chandy. Thanks also to Infospheres group members and fellow graduate students Dan M. Zimmerman, Adam Rifkin, and Roman Ginis, Compositional Computing group member Eve Schooler, and postdocs and past Comp group members Dr. Paolo Sivilotti, Dr. John Thornley, and Dr. Berna Massingill.

Dan Zimmerman was particularly helpful in his motivation to develop elegant and powerful software frameworks, Adam provided the inspiration of a Web-yogi, and Paul is the student-theoretician in whose footsteps I walk. Thanks also go to Diane Goodfellow for her administrative support.

Finally, I'd like to thank my long-term partner and best friend Mary Baxter — you'll always be an inspiration in my life.

This work was sponsored by the CISE directorate of the National Science Foundation under Problem Solving Environments grant CCR-9527130, the Center for Research in Parallel Computing under grant NSF CCR-9120008, and by Parasoft and Novell. The formal methods and adaptivity (reliability, mobility, security) parts of the project are sponsored by the Air Force Office of Scientific Research under grant AFOSR F49620-94-1-0244.

Abstract

Modern computing systems are terribly complicated — so complex that most system designers and developers can only hope to understand their small piece of the larger project. The primary *technologies* that help system builders manage this complexity are object-oriented and/or component-centric, and the primary *tools* are those that assist in system modeling and specification.

It is my belief that the next stage in managing system complexity comes in the form of *system specification through formal methods*. Only with precise, complete, and consistent descriptions of our systems and their components can we hope to understand the hyper-complex engineering that has become prevalent in computing today.

But, only through the introduction of some middle-ground, semi-formal technique can modeling and specification break through into the mainstream. Such a specification methodology can't be too hard to use, but need to be formal enough that it will help system designers and tools check the consistency and completeness of the system and its components.

This thesis is the first step on the road toward formal specification of dynamic, emergent, distributed component systems, and addresses all of the requirements mentioned above. I introduce DESML: a set of new modeling constructs which can be used as a thin layer on top of most modeling languages.

DESML is a variant of the Unified Modeling Language (UML), not an extension. I have redefined the the core metamodel, thus the new language is no longer compatible at the meta-level with UML. Note that such a modification is not necessary; it is only a convenience in the definition of our new language.

The reader should be familiar with the Unified Modeling Language and at least one formal specification language. Suggested references include [42] and [50, Chapter 2] for UML, and [20, Chapter 6] for a specification language (in this case, Z [152]).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern computing systems are terribly complicated — so complex that most system designers and developers can only hope to keep track of their small piece of the larger project. The primary *technologies* that help system builders manage this complexity are object-oriented and/or component-centric, and the primary *tools* are those that assist in system modeling and specification.

1.1 Motivation: Tackling Complexity

But, nearly 30 years after their introduction [12, 162, 126], these technological approaches only partially solve the complexity problem. They are not the final solution because large systems have hundreds, or even thousands of entities (classes, components, objects, files, computers, databases, etc.) and orders of magnitude more associations, dependencies, and relationships.

It is my belief that the next stage in managing system complexity comes in the form of *system specification through formal methods*. Only with precise, complete, and consistent descriptions of our systems and their components can we hope to understand the hyper-complex engineering that has become prevalent in computing today.

Although the use of development tools, from IDEs to expert systems, can help lower the usage cost of a specification methodology, this trend, that of *avoiding* complex specification methodologies to tackle complex systems design, cannot continue. Complex system development has reached the normal user's desktop — witness the Windows 98 operating system with its multi-tens of millions of lines of code.

Only through the introduction of some middle-ground, semi-formal technique can modeling and specification break through into the mainstream. Such a specification methodology can't be too hard to use, but need to be formal enough that it will help system designers and tools check the consistency and completeness of the system and its components.

Before we can explore the problem of system specification and modeling in more detail, we need to review our terminology so we have a common ontology for the rest of this thesis.

1.2 Terminology

We distinguish between a specification language, a model, a methodology, and a process.

A *language* is the means by which we describe a component or a system. A specification language can be realized as a well-defined natural language grammar, a mathematical notation, or can be a well-defined graphical language, either ideogrammic, logogrammatic, or pictographic. We use one or more languages to describe a system and its components.

A *model* is an abstraction, a system with base constructs, rules, and operators. Models, like formal theories, are focused on a small set of core entities. The Actor model has, of course, the actor construct, Z has its schemas, Demeter has its propagation patterns, predicate calculus has proof rules, etc.

A *methodology* is a language and model coupled with a set of rules, heuristics, suggestions, patterns, etc. A methodology is the full conceptual and descriptive package, but it is not complete without the behavioral description of *how* to use the methodology.

A *process* is just that description. It is the set of rules and heuristics that are provided as a guide for building the specification of the system with a methodology. Some processes are specific to a particular method, others are generic.

Thus, for a complete specification methodology, we need a formally defined *language* with well-defined semantics, a *model* that is focused on the proper abstraction(s), and a *methodology* coupled with a *process* that bring the language together with the model in a complete package.

1.3 Thesis Roadmap

This thesis is the first step on the road toward formal specification of dynamic, emergent, distributed component systems.

The reader should be familiar with the Unified Modeling Language and at least one formal specification language. Suggested references include [42] and [50, Chapter 2] for UML, and [20, Chapter 6] for a specification language (in this case, Z [152]).

In Chapter 2, I will discuss the general problem of systems and component specification in more detail. Next, in Chapter 3, I will describe some of our extensions to an existing system-level specification methodology. Then, in Chapter 4, I will describe the requirements, design, and implementation of the Infospheres Infrastructure 1.0, a dynamic, emergent, component-based framework. I will provide the specification of some of this framework in Chapter 4. In Chapter 5 I will review several of the distributed systems built with this infrastructure. I will then wrap up in the conclusion to the thesis in Chapter 6, summarizing the contributions of this work and its future directions.

Chapter 2

Specification

It is our belief that the next stage in managing system complexity comes in the form of *system specification through formal methods*. Only with precise, complete, and consistent descriptions of our systems and their components can we hope to understand the hyper-complex systems that are becoming prevalent in computing today.

In this chapter I will review the problem of system and component specification and discuss some of the challenges of specifying complex systems.

2.1 Formal Specification: A Brief History

Formal specification techniques have been used in computer science and other disciplines for over thirty years. Many of the abstract models have evolved from a formal grounding in mathematical [20, 67] and temporal logics [59].

Specification languages have been developed for a variety of *abstraction levels*, *domains*, and *goals*. These axes considerably influenced the design and evolution of specification languages and systems.

Abstraction Levels. System specification has many levels of granularity. At one end of the spectrum, what I'll call the "microscopic" level, there are specification languages for describing very static, precise systems, like the gates in a CPU (e.g. VDM [11]). At the "macroscopic" end of the scale, there are specification languages for describing entire huge, dynamic, and complex systems, like corporate entities (e.g. OOCL [161]).

This granularity of specification, the *abstraction level*, influences the kinds of things that can be described. A language that attempts to cover too many levels is likely to be overly ambiguous or complex. A language that is used for a very specific abstraction level will not be used very often and won't grow or be extended because of its restricted initial domain.

Domains. Most specification languages are developed for specific problem domains. For example, VDM is a specification language primarily used in designing digital circuits, predicate calculus is used for the specification and proof of computational algorithms, and Concurrent Sequential Processes (CSP) [82] and UNITY [30] were developed to assist in the specification and proof of concurrent systems. Thus, the systems for which specification languages are developed influence their application, flexibility, and usability.

Likewise, some languages are developed for a particular system paradigm. IBM's Object Constraint Language (OCL) was designed primarily for object-oriented systems, thus it has inherent support for object-oriented constructs like features, attributes, inheritance, etc. Most specification languages, (all mentioned thus far but for OCL), were not developed with object-oriented systems in mind. Therefore, while they can often be applied to such systems, the fit is imprecise and the resulting usage is strained and often *ad hoc*.

Goals. Finally, languages are often designed with their primary user in mind: either a human being or a computer. If a language is solely developed for the mathematician or computer scientist, encoding and manipulating it with tools such as theorem provers is often difficult or impossible. Conversely, some formal specification techniques are entirely designed for a computer. While efficiently encoded and eminently parsable, they are often nearly impossible for a human being to understand. For example, one might like to read a such a specification to help debug the tools that use the original information.

It is my belief that a successful specification technique must be usable by *both* the human designer and the computational support infrastructure.

While many specification languages are meant to help an algorithm or system designer prove the correctness of their artifact, many of the most popular languages today have nothing to do with formally assisting in the development of correct and complete systems. In fact, unsurprisingly, the widespread utilization of a specification language, much like a normal computer language, seems to be inversely proportional to the language's complexity — *i.e.*, the simpler the language, the more system builders will use it.

2.2 Current Leading Methodologies

There are several current leading methods for specifying systems and system components. These methodologies are either very formal, or fairly informal — there are few methods that ride the middle-ground between usability and formalism.

2.2.1 Informal System Specification Methodologies

First-Generation Methodologies. The leading first-generation informal and semi-informal system methodologies include Booch [15, 16], Coad-Yourdon [35, 36], Martin/Odell [118, 119], OOSE/-Objectory [93, 94], Rumbaugh [142, 143], Shlaer-Mellor [148, 149], Syntropy [39], and Wirfs-Brock [174]. These methodologies focused primarily on building small to medium scale (up to hundreds of classes) object-oriented systems. Some are not integrally tied to the object-oriented paradigm, they are either process-oriented or data-oriented. But, uniformly, all do not have the constructs necessary to describe very large and/or distributed systems.

Second-Generation Methodologies. The second-generation methodologies that are generally more formal and complete include Fusion [37], Meyer [121] and the Unified Modeling Language [17, 92, 144]. Designers of these new modeling techniques incorporated constructs that assist in the specification of large-scale and/or distributed systems. Additionally, these methods are capable of a

more formal specification of system components. Specifically, all encourage the use of pre-conditions and post-conditions in component specification, all propose the use of side-effect-less constraint specification, and all can specify loosely-coupled dynamic systems.

Third-Generation Methodologies. Finally, third-generation languages, now emerging from the modeling community, show further refinement. They incorporate ideas and terminology from cognitive science, knowledge engineering and representation, and information theory. These new techniques include Catalysis [47], Demeter [111], and Object-Oriented Change and Learning (OOCL) [161]. Due to the integration of ideas from these domains, these new techniques are very powerful and flexible.

Cataysis. Catalysis, from Desmond D'Souza and Alan Wills, is the most "normal" of the third-generation languages. It is an extension of UML that incorporates diagrams, techniques, and abstractions to specifically handle today's software architecture aspects such as components, patterns, and frameworks.

Demeter. The Demeter methodology from Northeastern University focuses on the problem of adaptive software. It is used, both for specification and implementation, at the meta-level more often than it used at the architecture level. A designer and programmer is left to deal with programming at a more semantic, higher schematic level. Predictably, this proves to be both difficult and powerful.

OOCL. Object-Oriented Change and Learning is a new methodology from Ed Swanstrom of Agilis Corporation. Swanstrom has incorporated numerous ideas from cognitive science and artificial intelligence (specifically, knowledge representation) to create a methodology that is more capable of modeling general organizations as well as software systems.

All of these specification methodologies are focused on three things: (1) the *language* used in the specification of a system, rather than system components, (2) the *process* of iteratively building the system model, and (3) *binding* the model to the implementation. What they do *not* help with (today) is any formal model- or system-correctness checking. That is the domain of the formal system specification languages.

2.2.2 Formal Component Specification Languages

Some of the leading formal models and languages for component specification include the Actor model [4, 3, 58, 80], Actor Algebras [63], the Larch language and system [73], OCL[88], Predicate Calculus [46], Process Algebras (CCS [123, 79]), CSP [82], SDL [25, 166], UNITY [30], VDM [97], and Z [90, 137, 152, 153, 178].

These models and their complementary languages are, for the most part, domain-specific. In particular Actors and UNITY have been proposed as a general purpose models for the specification and validation of concurrent and distributed systems. Dijkstra's predicate calculus is a self-contained theory of predicate transformer semantics, primarily used for the specification and development of general procedural programs. Similarly, Z (pronounced "zed") is a logic-centric model and language, also for the specification of procedural programs.

These languages are all equally powerful, permitting the user to completely specify system behavior and prove properties of the system. The two primary problems with these models and languages are:

- *They are not equally expressive.* Each specification language has a different abstraction level. Some, like OCL, can not denote side-effects. Predicate calculus and UNITY, in focusing on program semantics, deal with extremely low-level constructs like program statements. The Actor model's primary abstractions are single-threaded active processes that communicate using messages.

  All these specification languages are Turing-complete, thus they can all are equally powerful. But it is clear that the specification of a large message-passing system in a reasonable task in the Actor model or CSP, but would be horrendously complex in Z.

- *They ignore the models of modern system design and development.* None of these models take into account object-orientation, component-centric software, or meta-programming. While expressing higher abstraction levels with new constructs is possible, it leads to a model that is overly complex and focused on the wrong levels of abstraction and core entities. Examples of retrofitted or ill-targeted models include Object-Z [48], MooZ, OOZE, Z++, VDM++ [49], Seuss [125], and Actors.

These problems necessitate the creation of a new specification language and system. Such a system must inherit most of its formalism from the traditional languages like Z and others, but it must also be grounded in modern system design and development. I will discuss these requirements in more detail after providing an example of the range of formalism in use today.

2.3 Examples: From the Informal to the Formal

For an example, I'll consider the general specification technique *Use-Cases* [94] and the formal specification language *Z* [137]. Use-Cases is extremely simple to learn and use, thus it is widely adopted in industry. Z complex and formal, thus is rarely used anywhere, even in academic circles.

2.3.1 Use-Case Modeling

Use-Case modeling is a specification technique for describing a system at a very high abstraction level. Use cases are primarily utilized to informally document the interactions between a system and its users and to generalize a more detailed specification construct. Because of their simplicity and customer/end-user focus, use cases have received a great deal of attention in the development of second generation modeling techniques like the Unified Modeling Language [40, 41, 42].

The primary components of a use-case model are *use cases*, *actors*, and the *system* being modeled. The *system* is a black-box that implements some set of functions, each of which is represented by a *use case*. The external agent that is interacting with the system, whether it be a human being, a second part of the system, or another system altogether, is modeled as the *actor*.

An actor is represented by the classical "stick-figure", labeled with its role. The system is diagrammed as a simple box. Each use case is modeled as a labeled ellipse and is connected to all other related actors and use cases by a solid line.

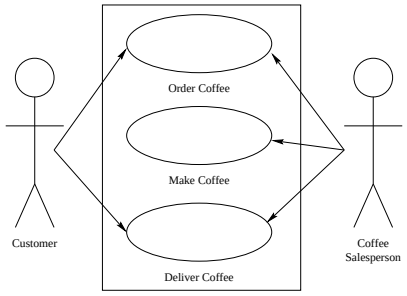


Figure 2.1: An example Use-Case diagram (a coffee-house)

For a simple example of a use-case diagram, see Figure 2.1. In this diagram, the actors are the "Customer" and the "Coffee Salesperson", the system is the box (the café), and the use cases are the three actions (ordering, making, and delivering coffee). Note that the "Customer" actor is not connected to the "Make Coffee" use case since the customer does not make his own coffee.

Use cases are primarily utilized as an informal means of documenting system functionality — a "leading abstraction" for other, more formal and complete, system specification methods. For example, in UML, a set of use-case diagrams can be linked to sequence, collaboration, and activity diagrams, three kinds of more detailed diagrams in UML. Likewise, in Catalysis, use cases represent an abstraction of a new modeling construct called a *collaboration* [47].

A use case must, by definition, deliver value to the actor. In Objectory and UML, all actors are classes, and most actors represent system elements like customers, users, and other real-world entities. Thus, these same entities (*i.e.* a paying client) see the real-world tangible benefits of developing and refining use cases in system modeling.

Use cases are a low-cost, low-complexity means of performing informal system description that keeps the customer happy because they can understand and affect the design process. Due to this positive benefit to complexity ratio, use-case diagrams are often seen in customer-focused system modeling more than any other diagram type.

A good summary of use-case modeling is found in [50].

2.3.2 The Z Specification Language

At the other side of the complexity and completeness spectrum we find the Z system specification language.

The Z language is a mathematical notation used to describe complex computing systems. It is based on set theory and logic, and its core constructs are independent of specification domain or implementation abstraction.

One goal of the Z designers is the ability to prove global properties of the system directly from the specification. Two of the most important system properties that can often be proven are the consistency (there are no contradictory specification elements) and the completeness of the system (the system model matches the real system being specified). Additionally, with a Z system specification, we can test the truth value of certain system-specific predicates like, "Can situation X ever arise?" (In general, this is not a decidable problem, but under the restrictive domain of system modeling, we can gain utility for such predicates.) Finally, sometimes we might want to be able to

Operator	Symbol	Example Usage
Powerset	$\mathbb{P}$	$\mathbb{P}(\mathcal{A})$
Universal Closure	^{c}	A^c
Unique, Counting, Summation Quantifiers	$\exists_1$, Ω, Σ	$\exists_1 x : S \bullet P(x)$
		$\Omega x : S \bullet P(x)$
		$\Sigma x : S \bullet P(x)$
Bag Collection Type	$[\![\ldots]\!]$	$[\![1 \ldots 100]\!]$
Sequence Collection Type	$\langle \ldots \rangle$	$\langle 1 \ldots 10 \rangle$
Sequence Concatenation	$\frown$	$A \frown B$
Naturals (starting with 1)	$\mathbb{N}_1$	$a \in \mathbb{N}_1$
Special Schema Symbols		
Incremental Inclusion	Δ	$\Delta message$
Identity Inclusion	Ξ	$\Xi message$
Free Type Definition	$::=$	$MESSAGE ::= yes \mid no$
Schema Define	$\widehat{=}$	$schema \mathrel{\widehat{=}} s \land t$
Schema Predicate	pred	pred $message$
Identifier Renaming	$/$	$NewName/OldName$
Identifier Hiding	$\setminus$	$\setminus OldName$
Pre-Condition	pre	pre $message \neq nil$
Post-Condition	post	post $message \neq nil$
Schema Compose	$\fatsemi$	$schema1 \fatsemi schema2$
Schema Pipe	$\gg$	$u = s \gg t$

Table 2.1: A partial summary of the Z language operators.

generate source code for the system directly from the specification.

A Z specification for a system is a set of units called *schemas*. Each schema has a declaration part, where a scope is defined, and a logical or predicate part, which specifies a set of predicates which must be true in the system.

Z's notation is a combination of operators from set theory, mathematical logic, functional programming collection types, predicate calculus, number theory, and some special symbols reminiscent of knowledge representation languages. See Table 2.1 for a partial list of some of Z's more interesting operators and constructs.

It is clear that, before one can use Z, one needs some formal training. A user must understand basic mathematical logic and set theory. Additionally, Z introduces new operators not found in the core mathematical training of most professionals which must be incorporated into a user's repertoire. These requirements impose a substantial learning curve on system designers that wish to use Z.

Z is formal for a very good reason. In essence, the use of the Z language is less *about* the specification than it is about the facts one can prove *from* the specification. This goal highlights the "tradition" of *specification towards proof*. There are several implications in this proof-oriented tradition.

Specification Towards Proof. First, formalism implies a certain degree of rigor in use. Many users do not have the discipline necessary to fully take advantage of such a formal system.

Second, as mentioned previously, there is a steeper learning curve in the use of Z than in a visual system methodology, even one as comprehensive as UML. (This fact might change when I consider some of the more formal, third-generation methods mentioned previously.) Therefore, users take

longer to learn the language and must work harder to take full advantage of the system and its tools. This ramp-up time and cost mean that often designers never get the chance to learn a system like Z because of their deadline-focused management.

Third, while many tools exist to assist in the validation of formal specifications and related proofs, most of the formal work at this level is performed by a human expert. Often, this human expert is the designer. Thus, not only must one be a good designer, but one must be a good mathematician to take advantage of the benefits of such a system.

These three implications mean that only some small portion of the designers capable of using an informal methodology have the resources to take full advantage of such a formal system. I'll consider the implications of this statement in the next section.

For the interested reader, an excellent concise overview of Z is available in [20, Chapter 6].

2.4 The Utility of Specification

Over the years, I have observed the genesis, evolution, and adoption of many specification and modeling languages. Based upon this experience, I have come up with the following set of "principles of specification". These principles have had an enormous impact on my work (in the form of this thesis, related papers, tool building, etc.).

- *Language does matter.* While I find the development and use of "research" and out-of-mainstream languages interesting and challenging, I will not pursue the creation or use of such a language.

  I consider the following interesting research languages: Beta [106, 113], CLOS [14, 60, 101, 102, 131] and [155, Chapter 28], Dylan [9], ML [38, 124, 134], the Pascal/Modula family [127, 128, 175, 176, 173], Oberon [141, 177], Obliq [23, 24], Self [2, 167, 168], Simula [12, 45], and Squeak [91]. The following are the out-of-mainstream production languages I believe are worthwhile: Eiffel [121, 122], Objective-C [43, 44] (the best short introduction can be found in [108], the best book on the language is [136]), Smalltalk [65, 66], and Python [112, 170, 171].

  The adoption of a more "main-line" programming language that fulfills most of our needs will result in a body of work that can potentially influence a large number of practicianers. I chose to use the Java language [10, 68] and XML representation format [19] for exactly these reasons.

- *The support of modeling tools is very important.* Different languages have different levels of tool support. Some (e.g., Actors) have no tool support; they are used as "by-hand" mathematical languages, useful for specification and proof. Others (e.g. UNITY, predicate calculus, OOCL, and OCL) have partial tool support; a verifier, like UV, the UNITY Verifier [98], or theorem prover or proof-checker might be available to assist an expert in system specification (e.g. Isabelle [133], LOTOS [169]).

  Finally, some (UML, Demeter, the VDM family, the Z family, and the Larch family) have excellent tool support. They include generators, parsers, reusable libraries, compilers, proof-checkers, etc., and thus are excellent specification languages from a tools-oriented point of view. More users will be inclined to try out a specification language if it includes a tool that helps them get their job done. (Examples include: Rational Rose [139]; the Demeter Tools [85];

IFAD VDM-SL Toolbox [89]; Cogito [13]; LP, the Larch Prover [62], LSL, the Larch Shared Language [73], and LCLint, a tools used for statically checking annotated C programs [51] built with Larch).

Moreover, if one can insinuate the use of a formal specification language via a commonly used tool *without the user knowing (a priori) that they are using such a system*, we will gain accidental "converts". Once users realize the impact that such a system can have on their design and programming efficiency and their system's reliability, we speculate that they will become true evangelists for the system.

- *Inclusion of innovative "boundary" research areas is necessary, but not sufficient, for a method's adoption.* Some methods have focused on specific "boundary" research areas, thinking that their innovation will be enough to make an impact. This highly-focused work, while important within the specific area of research, is often not communicated fully to, and appreciated by, a wider audience. It is very difficult to present a tightly focused, highly-specialized body of work to a general audience.

  Consider the Demeter method (a focused summary of Demeter's core concepts can be found in [111, Chapter 15], an annotated version of [132]). Its focus is exclusively on adaptive object-oriented software through the use of propagation patterns. Because of this tight focus, and the fact that the method uses metalevel architectures, it has not had much impact in the modeling and design communities.

  To quote Ralph Johnson, "In my opinion, the entire vocabulary of the reflective programming community is an unmitigated disaster. That community has succeeded in taking an extremely important topic and making it so hard to understand that it is ignored by most of the people who need to know it [96]."

  This "generated complexity" that is prevelant in many of the fields that this work is based upon, (metaobject protocols, knowledge representation, reflection, semantics representation, etc.), I hope to avoid though a conscious effort to keep our vocabulary and concepts clear and to the point.

- *The proper ground technologies must be chosen.* A proper foundation for such a complex and comprehensive body of work must be carefully chosen. This will help reduce the amount of work, provide a solid base upon which to build, and help regulate and order the development of new constructs and tools.

  I have chosen UML as the base system modeling language upon which to build, and XML as our ground representation language. UML is the *de facto* standard for system modeling and has a fairly well-defined (but not well-tested) mechanism and process for extending the core language. XML is the emerging *de facto* representation format for arbitrary data and has many constructs (namespaces [18], linking [116], and pointing [117]) and instantiations (RDF [109], CKML [34], and OML [130]) and tools [53] which lend themselves to our use.

To summarize, I believe that next-generation work in formal specification must use the right language(s), provide tools, include interesting and relevant "boundary" research ideas, and use the proper ground technologies and standards. I aim to fulfill all of these requirements in this body of work.

I will next consider in more detail some of the new challenges in system's design and architectures that influence this work.

2.5 New Challenges in Specification and Modeling

The last ingredient necessary for a successful specification methodology is the ability of the method to handle the challenges, techniques, and abstractions in modern software architectures. I believe that the most important aspects that must be captured by the specification methodology are:

Dynamism. Modern system components change over time; they are replaced, they change behavior, they change location, etc. Additionally, the system changes over time as it responds to the changing demands. The forces behind the dynamism, much like Wegner's interactive systems' stimuli [172], are placed upon it by other systems, users, customers, etc.

Emergence. Systems not only change over time, but can exhibit behaviors that were not originally part of the system's original design. Such *emergent* behavior is unsurprising in some systems. Corporations, intentional non-deterministic or random systems, and artificially designed, evolvable simulations, like Tom Ray's Tierra [140] commonly exhibit surprising behavior [99, 100]. But a system like the World Wide Web, very simple at its core but complex in its scope, application, and degree of growth, is not expected to evidence such evolving behavior.

Meta-Levels. "Meta" levels of system design and development are gaining attention for their expressive and descriptive power. Such constructs and theories include metaobject protocols [102], aspects [103], subject-oriented programming [77], and reflection [114, 151]. A complete methodology must be able to capture all of the meta levels of the system being modeled.

Components. Component architectures are at the root of most software development today. The component, as a base level of abstraction, is the appropriate core construct for the language, model, methodology, process, and complementary theory. No existing methodology and theory has the component as its core construct, a serious deficiency from our standpoint.

Knowledge Representation. Specification is all about capturing information and knowledge about a system and its components. Constructs and theory from knowledge representation can be applied to the problem of software system specification. A methodology must incorporate these ideas and tools.

As mentioned previously, the goals of this work are more than just the evolution of an existing methodology or the creation of a new specification language. We must also provide the tools, process, model, and theory to support this broad collection of work. Therefore, such a framework will unify the generalism, breadth, and informalism of methodologies like UML, *and* take advantage of the formalism of a specification language like Z or Larch, *and* package it all in a cohesive body of work.

To realize these ideas and goals, an implementation framework for dynamic active objects needed to be created. I was a key figure in the creation of just such a framework — the Infospheres Infrastructure, version 1.0. A next-generation framework (version 2.0) is currently under development. It incorporates even more innovative ideas from the object and Web communities.

Additionally, the system-level and component-level keystones of this work, the DESML and CDL specification languages, are being developed.

2.6 Framework for Experimentation: Infospheres 1.0

The Infospheres Infrastructure, version 1.0, (I will use II for short), is a first-pass design and implementation of a component-based, dynamic, active object framework. It was built to (1) explore the use of the Java language in building such a system, and (2) to facilitate the development of high-level systems and tools to reify the theoretical and modeling work of the Infospheres group at Caltech. I will describe this framework in more detail in Chapter 4.

2.7 The Dynamic Emergent System Modeling Language

DESML, the Dynamic Emergent System Modeling Language, is a formal variant of UML for dynamic, emergent distributed systems. It is specified with the standard extension mechanisms of UML: Stereotypes, Tagged Values, and Constraints and a slight modification to the UML metamodel. It provides several new graphical constructs to extend the "visual" language of UML. DESML also incorporates many of the ideas that are part of the third-generation modeling languages, particularly OOCL and Catalysis.

Some of the foundation constructs in DESML are described in Chapter 3 of this document.

2.8 The Component Description Language.

The other fundamental half of the specification architecture is CDL, the Component Description Language. CDL is a new specification language that incorporates ideas from several of the other specification models and languages previously mentioned (OCL, Z and VDM variants, Larch, UNITY, process algebras, and predicate calculus, in particular). CDL's core model and complementary theory is component-centric and based on object calculus [1]. CDL will be described in full detail as part of my PhD dissertation. It is not described here.

2.9 Examples

A number of distributed systems have been built with the II. Several of them will be described in Chapter 5.

Chapter 3

Modeling Dynamic Distributed Systems

This chapter presents the core of this thesis, a set of modeling constructs that are the foundation of DESML. These constructs fulfill the challenges and requirements discussed in Section 2.5 of the last chapter.

3.1 Background

I will provide the reader with some background in the domain of system modeling so that they have sufficient context to understand the contributions of these new modeling constructs.

System modeling languages with visual constructs have been used for decades. The field gained attention in the 80s with the emergence of object-oriented languages and systems into the mainstream.

The original system modeling languages were not object-oriented, they were either *data-oriented* or *behavior-oriented*. Languages from both domains evolved and were incorporated into object-oriented modeling languages. Understandably, the bias of their origins can be seen in this transition.

3.2 Historical Opposites: Booch and Shlaer-Mellor

The Booch [15] and Shlaer-Mellor [148] object-oriented methodologies are the historic archetypes of behavior and data-oriented methodologies. Booch is a behavior-oriented model, Shlaer-Mellor is a data-oriented model.

Data-oriented techniques focus on describing the data that is at the core of all computational systems; behavior-oriented techniques concentrate on describing the activities of a system, independent of the data representation. Data-oriented modeling techniques primarily originate in the database and information processing communities. Behavior-oriented techniques, on the other hand, originated with the "traditional" computational-oriented practicianers.

In some sense, both data-oriented and behavior-oriented languages are the "right" source from which to evolve an object-oriented language. Object-orientation is about about understanding the behavior of a system as applied to the data; thus, the unification of both domains is appropriate.

Thus, both Booch and Shlaer-Mellor significantly influenced the development of today's leading modeling languages. Booch (and other leading behavior-oriented methodologies of the late 80s) provided many of the behavior-focused constructs, Shlaer-Mellor (and the other data-oriented methodologies) influences the data-oriented aspects.

3.3 Current Modeling Languages

The bulk of today's modeling community has settled down on a single syntax and semantics for a code modeling language. This core language is called the Unified Modeling Language (UML) [40, 41, 42].

UML was jointly developed by many companies, but leading the effort are three of the major researchers in the modeling community: Grady Booch, James Rumbaugh, and Ivar Jacobson. Each of these individuals brought their own perspective and modeling language and process to the table. Booch developed the previously mentioned Booch method [15], Rumbaugh's language is called the Object Modeling Technique (OMT) [143], and Jacobson's language and process are called OOSE/Objectory [93].

Additionally, several other companies, individuals, and methods influenced the development of UML. One of the most influential methods is called Catalysis, and is considered one of the leading-edge methods available today[1].

Finally, there exist some new methods that are outside the mainstream. While they often use UML as a syntactic base, they introduce new concepts, constructs, and techniques to system modeling. One of these new-generation languages is OOCL [161].

We'll briefly discuss these three leading methods, highlighting their strengths and weaknesses. Then I will introduce a set of new modeling constructs and diagrams which can be added to any modeling language to improve its expressiveness and capability.

For the reader interested in an overview, history, description, and comparison of the primary methodologies of the late eighties and early nineties, see [86, 87, 119].

3.3.1 UML: The Unified Modeling Language

As previously mentioned, UML primarily originates from the work of Booch, Rumbaugh, and Jacobson. Booch and Rumbaugh began working on UML together in 1994 when Rumbaugh joined Booch's company, Rational Software Corporation. Their goal was to unify the Booch and OMT methods, and thus unify their user bases. Jacobson later joined in 1995 when Rational purchased his company, Objective Systems. At this point in time, the three realized that they were doing more than unifying their own methods, they were creating a standard modeling language for the entire modeling community.

As they worked on UML, Rational provided preliminary specifications via the Web. Feedback from the modeling community provided many ideas and suggestions for incorporation into the language. Simultaneously, the OMG issued a CFP for a standard modeling language for object systems. The UML developers were instrumental in the issuance of this RFP and realized the value in making UML an industry-wide standard with the OMG's stamp of approval.

[1] In fact, this author was the reviewer of the new Addison-Wesley Catalysis book: *Objects, Components, and Frameworks with UML - The Catalysis Approach* by Desmond D'Souza and Alan Wills [47].

Even though the core of UML comes from Booch, OMT, and OOSE/Objectory, ideas were incorporated from several other modeling languages as well. For example, Harel's work on state charts [74, 75, 76], Coleman *et al* work on numbering operations in Fusion [37], and Gamma *et al* work on documenting patterns [61] were all rolled into the UML 1.0 final release.

UML is a modeling language designed by a committee. Thus, it cannot be expected to solve the challenges presented in the last chapter. In particular, UML does not handle emergent systems and knowledge representation at all. Its support for dynamism is limited to the standard set of diagrams used to describe dynamic systems.

The UML Diagrams UML is defined by a metamodel. This metamodel consists of a set of standard constructs and defines a number of diagrams used to describe the static, dynamic, and interactive aspects of a system. The eight diagrams defined by the UML standard are grouped into four classes:

- *System Interaction.*
  - *Use-Case Diagram.* I have already discussed *use-case diagrams* in Section 2.3. Briefly, a use-case diagram shows the relationship among actors (active entities including software elements and human users) and use cases (usage scenarios) in a system. An example use-case diagram was presented as Figure 2.1[2]
- *Static System Structure.*
  - *Class Diagram.* A *class diagram* is a graph of classifier elements connected by their various static relationships. A classifier is one of many static system constructs such as classes, interfaces, packages, relationships, and even instances of objects and links. An example class diagram can be seen in Figure 3.1.

    An *object diagram* is an instance of a class diagram indicating the static relationship between object instances, frozen in time.

    The stereotypes (see below) *type* and *implementation class* can be used to indicate that classifier elements are either object types (see Section 3.3.2 for the definition of type in this context) or implementations. The previously mentioned class diagram uses several stereotypes as an example.
- *System Behavior Diagrams.*
  - *Statechart Diagram.* A *statechart diagram* shows the sequences of states that an object or interaction goes through during its life-cycle in response to stimuli. It also documents the object's responses and actions. The semantics and notation of statechart diagrams is basically identical to David Harel's statechart diagrams [76], as mentioned previously. A simple example statechart is provided in Figure 3.2. A more complex statechart can be seen in Figure 3.3

[2] Several figures in this document are adopted from, or modifications of, a variety of sources including [163, 88, 41, 40, 47].

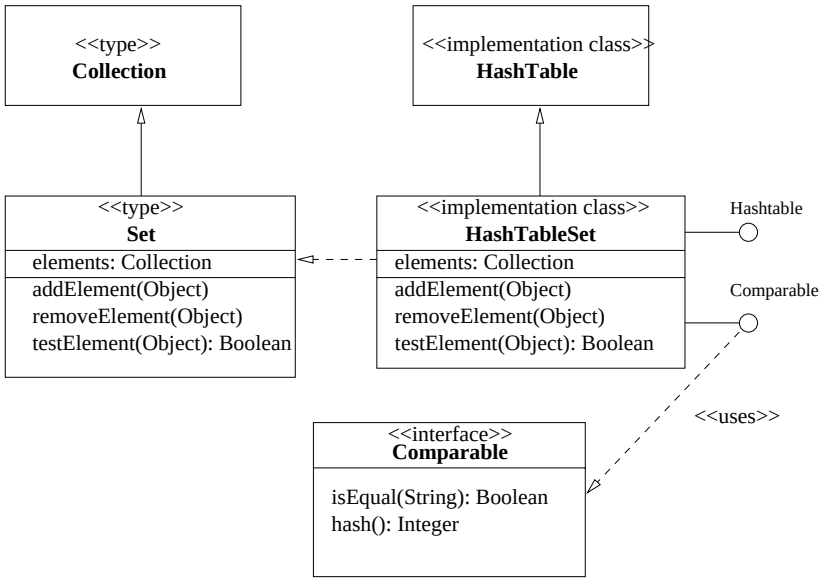


Figure 3.1: An example class diagram

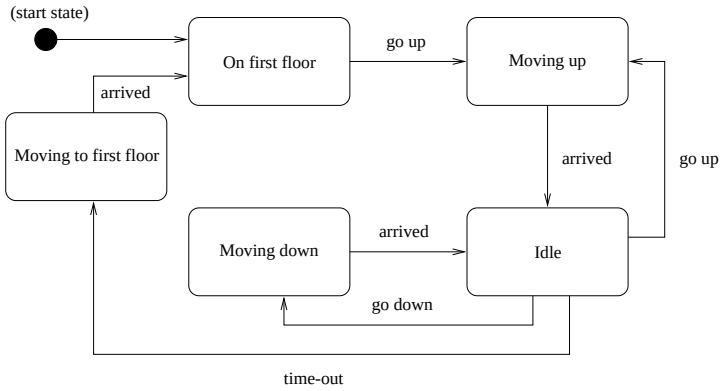


Figure 3.2: An example statechart diagram

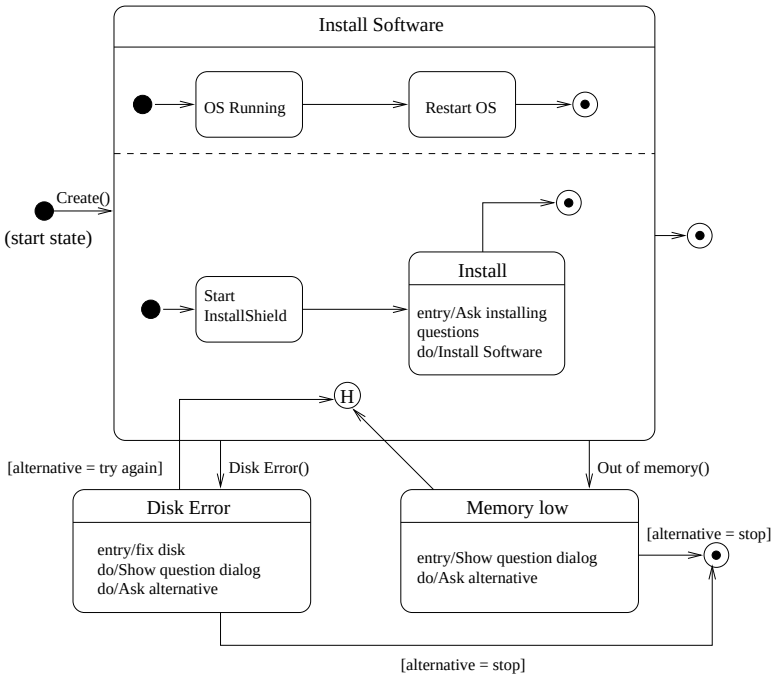


Figure 3.3: A more complex statechart diagram

– *Activity Diagram.* An *activity diagram* is a variation of a state machine in which the states are "activities" representing the execution of operations and transitions are triggered by the completion of operations. Figure 3.4 is an example of an activity diagram.

– *Interaction Diagrams.* A pattern of interaction among objects is shown on an *interaction diagram.* Interaction diagrams come in two flavors:

  * *Sequence Diagram.* A *sequence diagram* represents an interaction in which messages are exchanged among a set of objects to effect a desired result. An example sequence diagram is included as Figure 3.5.

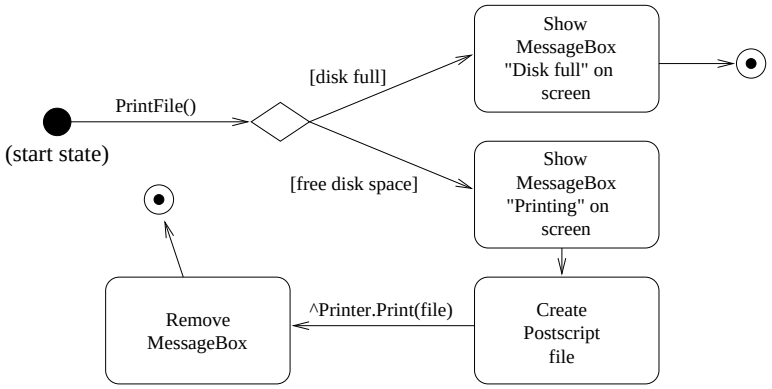


Figure 3.4: An example activity diagram: a print server

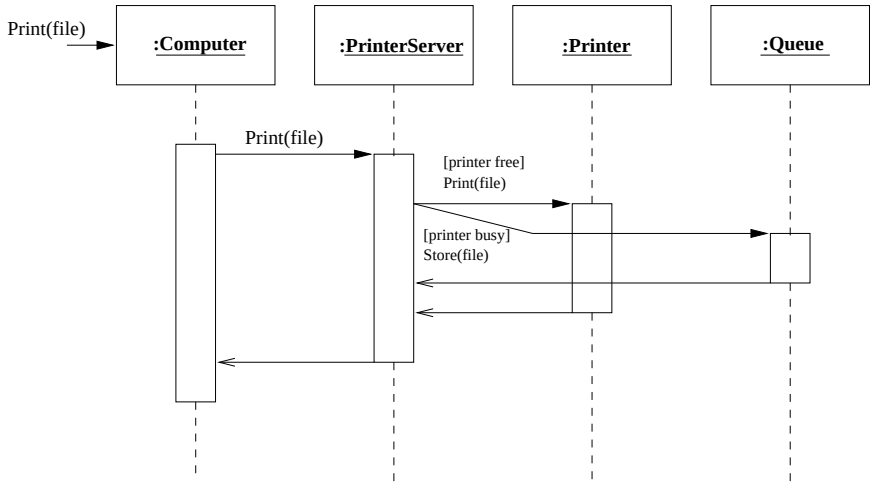


Figure 3.5: An example sequence diagram: a print server

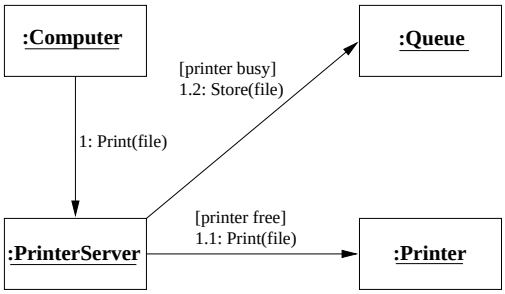


Figure 3.6: An example collaboration diagram: a print server

  * *Collaboration Diagram.* A *collaboration diagram* shows interaction among objects as well as their relationship with each other. Unlike a sequence diagram, time is not documented in a collaboration diagram, so the sequence of messages and concurrency must be indicated with sequence numbers. Figure 3.6 is an example of a collaboration diagram.

- *Implementation Diagrams.*

  – *Component Diagram. Component diagrams* are meant to document the structure of the code itself. Note that the notion of a component in UML has little to do with *component software*, one of the primary challenges presented in Section 2.5. An example component diagram is provided as Figure 3.7.

  – *Deployment Diagram. Deployment diagrams* show the configuration of the real processing systems on which a system is to run. Elements included in deployment diagrams include computers, network elements, processes, objects, networks, etc. The example deployment diagram, shown in Figure 3.8, documents the Infospheres lab here at Caltech.

Extending UML UML also provides a number of general mechanisms for annotating and extending system specifications. These constructs include:

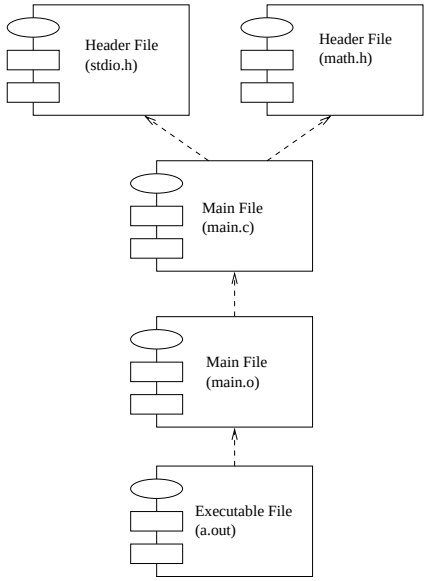


Figure 3.7: An example component diagram

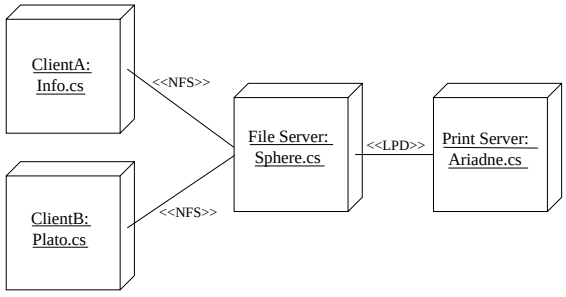


Figure 3.8: An example deployment diagram: a portion of the Infospheres lab

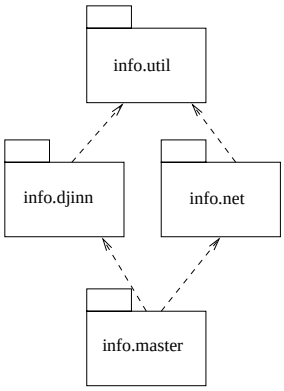


Figure 3.9: An example package diagram: the Infospheres core packages

- *Packages.* A *package* is a collection of model elements. Packages can be nested and inherited and are used to highlight high-level system dependencies. The semantics of a package are only one of collection. E.g. There is no necessary relationship between the UML package element and the *package* construct of Java or a *module* from Ada or Modula-3, though this would be the reasonable binding. An example of a package diagram can be seen in Figure 3.9.

- *Stereotypes.* A *stereotype* is a new class of modeling element that is introduced at modeling time. Stereotypes are introduced by specializing existing modeling elements. Generally, a stereotype indicates a usage or semantic extension.

  Stereotypes are represented by attaching a keyword string to an existing model element within guillemets, as in the following examples.

  $\ll abstract \gg$ $\ll actor \gg$ $\ll djinn \gg$

- *Constraints.* A *constraint* is a semantic relationship among model elements that specifies conditions and propositions that are invariant. Constraints are specified by including text in braces next to model elements. For precision, the text is usually some formal specification language like OCL or Z, or a mathematical expression.

  Several examples of constraints are provided as follows:

  $\{message.oclIsTypeOf(SummonRequest)\}$
  $\{\forall\, n : \mathbb{N} \bullet n + n \in even.\}$

- *Properties and Tagged Values.* Any value attached to a model element (attributes, associations, tagged values, etc.) is a *property.* A *tagged value* is a keyword-value pair that can be attached to any model element. Tagged values permit arbitrary annotation of models and model elements. Such annotation is considered an extension when the tagged values are precisely defined. Properties are specified as a comma-delimited sequence of specifications in braces, much like constraints.

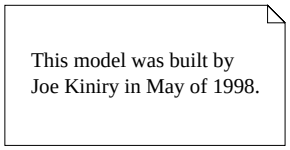


Figure 3.10: An example note

Here is an example of several properties.

```
{ message.priority = HIGH }
{ self.ref = other.ref }
{ Djinn.allInstances -> forAll(p1, p2 | p1 <> p2 -> p1.name <> p2.name) }
```

- *Notes.* A note is a graphical symbol containing information, normally textual but possibly graphical or programmatic. It is the notation used for rendering various kinds of textual information for a model or metamodel, such as constraints, comments, program code, or tagged values. An example note is included in Figure 3.10.

UML Limitations and State UML does begin to handle component software with component diagrams, insofar as components are objects and code. UML has no notation for representing the intricate relationships that components can have with one another, neither can it truly represent a component's inbound *and* outbound interface properly. I will present extensions to handle just these problems in the next section.

UML has an extensible metamodel and thus can describe the meta-aspects of a system, but only as long as such aspects are representable in the core modeling language. Realistically, most of the aspects of meta systems are not captured by UML at the non-meta level, thus they certainly cannot be captured at the metalevel. In particular, aspects, reflective components, and metaobject protocols are poorly handled by UML.

UML is now at revision 1.1 and its core specification is publicly available on the Web at `http://www.rational.com/uml/`. A brief introduction to the language is provided in [42] and [50, Chapter 2].

3.3.2 Catalysis

Catalysis is a modeling language and process that takes the next step in scalable, rigorous, component-based development. As D'Souza and Wills put it, "(Catalysis is) A next-generation standards-aligned method for open distributed object systems (that is constructed) from components and frameworks that reflect and support and adaptive enterprise." [47, Section 1.8].

The main "constructive" elements to system modeling that Catalysis contributes are called *types*, *collaborations*, *refinements*, and *frameworks*. In short, *types* describe the external behavior of an object, *collaborations* specify behavior of a group of objects, *refinements* relate different levels of description of behavior and map from realization to abstraction, and *frameworks* provide specializable, reusable, recurring patterns of collaborations, types, designs, etc.

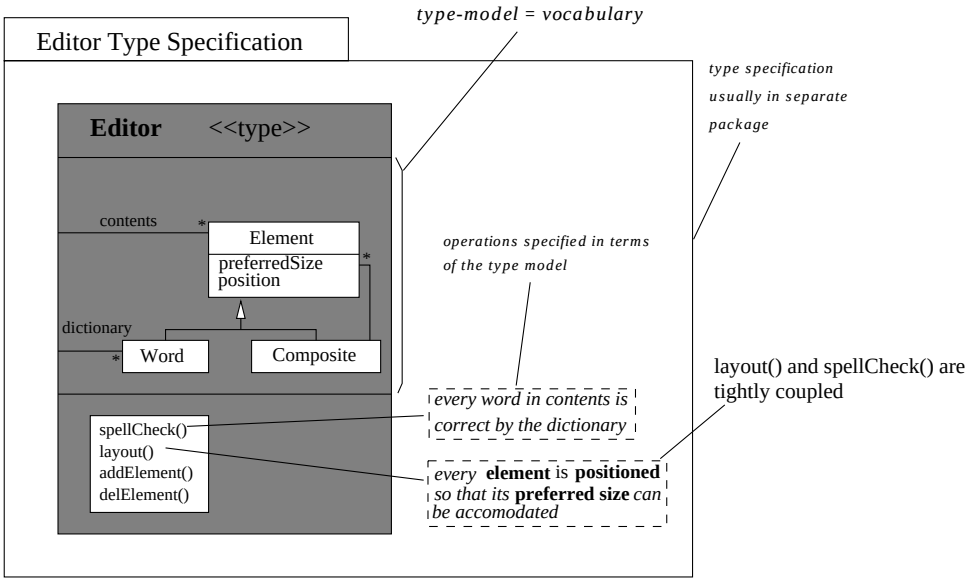


Figure 3.11: An example type: a portion of an editor with a dictionary

More specifically:

- *Types.* Catalysis's choice of terminology with respect to types is unfortunate. A Catalysis *type* is a a description of the (inbound) interface of an object; a list of its operations with their complementary safety conditions (preconditions and post-conditions). A Catalysis *type model* is the dictionary used to specify a type. An example type and type model are provided in Figure 3.11.

- *Collaborations.* A collection of components that satisfies a type specification is called a *collaboration*. A collaboration is essentially a well-defined composition of several types. An example collaboration can be seen in Figure 3.12.

- *Refinements.* Refinements provide a variety of means by which one can abstract specification. For example, types are refined to classes in Catalysis. Refinements in Catalysis differ from specialization and generalization in other modeling techniques because refinements can be applied to more than just system elements like classes and interfaces. E.g. Refinements can also be applied to business processes and specifications. See [47, Section 7.3] for a detailed refinement example.

- *Frameworks.* Almost all modeling reveals reccuring patterns of types, classes, attributes, operations, and refinements. Many such patterns are captured by existing modeling constructs like associations, constraints, generalization, etc.

  However, there are many patterns, usually at a higher level of abstraction, which are not captured by these built in constructs. Catalysis *frameworks* provide a means by which to document these general reccuring patterns. The notation provides a means by which an abstract framework can be instantiated to a particular instance, complete with parameterization of subelements. An example framework specification is provided in Figure 3.13.

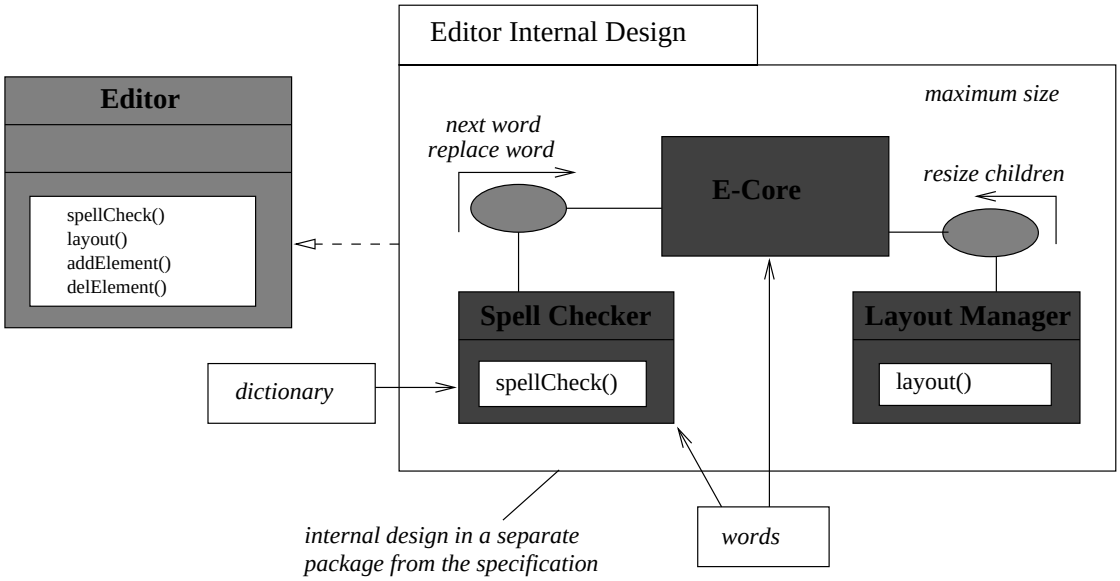


Figure 3.12: An example collaboration

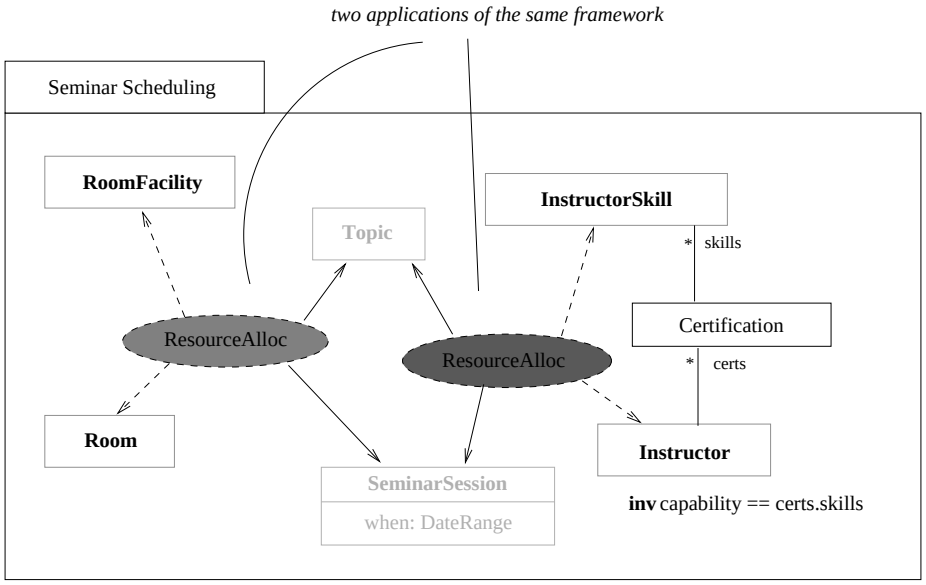


Figure 3.13: An example framework: applying a framework twice

The Unified Modeling Language and metamodel have adopted significant modeling constructs from Catalysis, including types, behavior specification, refinements, collaborations, and frameworks. Since Catalysis is the primary motivator in the introduction of types, refinements and collaborations to the UML core model, the same comments on the capability of the language hold for Catalysis as for UML. Meaning, Catalysis takes the first steps in the representation of components and component groups, but does not provide enough constructs to be as expressive as necessary in complex systems.

3.3.3 OOCL: Object-Oriented Change and Learning

Object-Oriented Change and Learning (OOCL) is a modeling language and process designed for "people-based" systems. Such systems are "organic", in the sense that grow, develop, evolve, mutate, learn, metamorphose, and adapt. As mentioned previously, these types of systems often defy traditional modeling and measurement techniques. OOCL, in its focus on modeling, understanding, managing, and accelerating the growth and learning capabilities of "people-based" business systems, might potentially be the leading methodology in tackling the problems mentioned in Section 2.5.

OOCL is primarily used for modeling business processes, strategies, complex adaptive systems, and system dynamics. I.e. OOCL is not used to design just software systems; it is used to design and understand business systems holistically — software is only one component of a complex business system.

OOCL as a methodology has extended the object-oriented approach to support Jay Forrester's Systems Dynamics modeling approach [54, 55, 56, 57] as well as the Systems Thinking approach developed by MIT. The newest version of OOCL (version 2.0), is a major method update with new features including the ability to model self-organizing, complex adaptive systems (CAS). For more information on the work underlying the OOCL method, see [83, 84, 146, 147, 161]. For a quick summary of OOCL, see [5, 160, 163].

OOCL introduces a number of new modeling constructs to help specify such dynamic system. These elements include:

- *Agent.* An *agent* is an entity responsible for a set of tasks.
- *Organization.* An *organization* is a group of agents working toward a common goal.
- *Constellation.* A *constellation* is a group of organizations working toward a common goal.
- *Resource.* Agents, organizations, and constellations consume, use, and create *resources.* Resources are things like money, materials, information, etc.
- *Process.* A *process* is a set of agents and resources interacting to deliver a particular service.

Each of these new modeling constructs introduces new symbols to the base modeling language. These elements are summarized in Figure 3.14.

OOCL combines techniques from complex adaptive system studies of Santa Fe Institute, cognitive science, organizational learning, object technology, business process engineering, agile manufacturing, and others. Essentially, it attempts to unify those practices and techniques that are concerned with organizational agility.

OOCL adds several diagrams and graphs to a UML base as well as introduces new steps to the standard software engineering process. Each of these diagrams or graphs introduces a number of

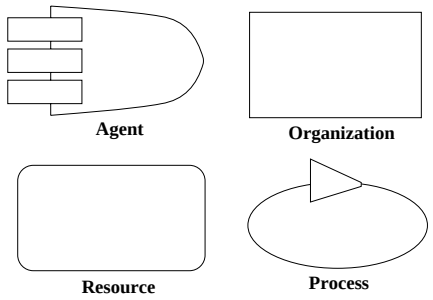


Figure 3.14: OOCL elements

new constructs to the language. The diagrams and graphs are organized below by the stages of the OOCL process.

- *Define Boundaries.* During this first phase of the OOCL process, the rules, assumptions, and presuppositions that influence the model are defined. This information is summarized in a *Microworlds Diagram.* Microworlds have two main hierarchies, one based upon generalization and the other upon meta-levels.

- *Scanning the Environment and Envisioning.* In this phase, the system modeler attempts to capture the vision and goals of the organizational components of the microworld. This information is captured in several diagrams:

  - *Vision Diagram.* A *vision diagram* documents the shared vision, goals, and missions of agents across groups, organizations, and constellations. This shared vision is what motivates agents within these contexts.
  - *Concept Mapping Diagram. Concept maps* are used to identify and capture organizational concepts from discussions, documents, etc.
  - *Metaphor Diagram.* A *metaphor* is a specific story that can be generalized to fit many situations.
  - *Stakeholder Diagram.* A *stakeholder diagram* documents how different stakeholders will utilize the business.

- *Business Analysis.* This phase presents an external view of the business system perspective. The emphasis is more on *what* the business will do than *how* the business does it. There is no focus on implementation at this stage. This stage builds a *Business Interface Model* using the following diagrams:

  - *Business Scenario Diagram.* The *business scenario diagram* captures the interactions between actors and the business system from a high-level (black box) point of view.
  - *Organization Interface Diagram.* This model shows the input and output of the business system.
  - *Business Schemata Model.* The *business schemata model* describes the semantics of each business stimulus in the business interface.

  - *Business Class Diagram.* A *business class diagram* documents the business objects that provide the logic for the business rules of an organization. Business objects are reifications of the concepts described in the concept mapping diagrams of a system.

- *Business Design.* During this stage, several diagrams are developed to refine the business class model of the last stage. Design is concerned with assigning operations to the classes defined in the business class model, determining how objects communicate, and what the inheritance relationships are between these classes.

  Several graphs and diagrams are developed at this stage:

  - *Business Object Interaction Graph.* The *business object interaction graph* describes the dynamic business behavior of an organizational unit.
  - *Activity Graph.* Each *activity graph* shows the activities that take place inside each process step in the business object interaction graphs.
  - *Business Class Descriptions.* These descriptions describe the precise specifications of the business classes.

- *Business Transformation.* During this phase, all the models created during the previous design phases are implemented in simulation, walk-throughs, or actual changes to the business structure. Two diagrams support this stage:

  - *State Diagram. State diagrams* are used to document the simulations necessary to transform the business structure.
  - *Project Diagram.* Finally, *project diagrams* are used to track and test processes during the development of the business.

- *Continuous Change and Learning.* During and after the initial phase of the OOCL process, a continuous cycle begins where observations and insights are learned and incorporated into the model. Lessons learned are encoded and stored in the level two (metalevel) knowledge repository. There are also level three and four knowledge and processes related to improving the team, division, or organization.

Providing examples of each of the above diagrams and graphs would take up too much space for this body of work. We suggest [5, 163] for summaries and examples of all of the diagrams.

OOCL also evolves and introduces new concepts to system modeling. The primary "evolved" traditional concepts at the heart of OOCL is the business object dictionary, an evolution of the data and class dictionaries of other methods. The primary new concept central to OOCL is the so-called "Corporate DNA". The Corporate DNA is the specification of the resources, motivations, goals, and aspirations of an organizational unit, primarily a corporation or project.

OOCL is focused primarily on describing complex systems that change over time, thus introduces a number of important concepts that help solve some of the challenges mentioned in the previous section. In particular, OOCL incorporates metamodels, knowledge representation, and dynamism as first-class entities for representation. Thus, OOCL is the only modeling language that potentially deals with these three challenges.

On the other hand, OOCL does not extend the modeling language with respect to core software engineering problems; that of component representation and interaction. It exhibits the same limitations as Catalysis and UML.

OOCL is not yet available in full for public review; [161] is still under development. When the full language definition becomes available, I will evaluate the capability and appropriateness of OOCL as a ground modeling language. I will come back to this point at the end of this chapter.

3.4 Modeling Extensions for Dynamic Systems

This section will introduce a small set of modeling extensions that help solve the challenges discussed in Section 2.5.

Several modeling problems are discussed. Each problem is briefly introduced, often with an example. Then the corresponding construct is presented that assists in solving the problem. The construct's syntax, both graphical and symbolic, is then explained. Finally, its semantics are fully specified.

Briefly, the problems discussed here are as follows:

1. **Core Interface Behavior Representation.** How can a component's interface be specified and how can we unify these interface specifications into a single model?
2. **Core Component Representation.** How can real software components be completely and properly specified?
3. **Partial Component Interface Specification.** What are the problems inherent in object and component specification and how can objects and components be partially and totally specified?
4. **Component Associations.** What kinds of associations exist between components and how can these associations be modeled?
5. **Dynamic and Emergent Structures of Components.** How can dynamic and emergent structures of components be modeled? What are the core representational elements?
6. **Tying Knowledge/Semantics to Components.** How can we relate information (knowledge and semantics) to component specification?

3.4.1 The Interface Behavioral Element

Problem One: Core Interface Behavior Representation. UML has several core constructs that explicitly and implicitly specify the interface of an object. Specifically, the (explicit) *Operation* and related metaclasses are independent in the UML metamodel, all specializing from *Behavioral Feature*. *Event*, the implicit metaclass, is currently not associated with *Behavioral Feature*.

I introduce two new specification elements, *Channel* and *Tuple*, which will be used to specify alternate interfaces for components. Additionally, I believe that the *Event* metaclass should also be used associated with *Operation* so that it can be used to specify the interface of a component.

The core metaclass backbone for UML is shown in Figure 3.15. Our modifications result in the new structure seen in Figure 3.16. Only the relevant part of the model are shown.

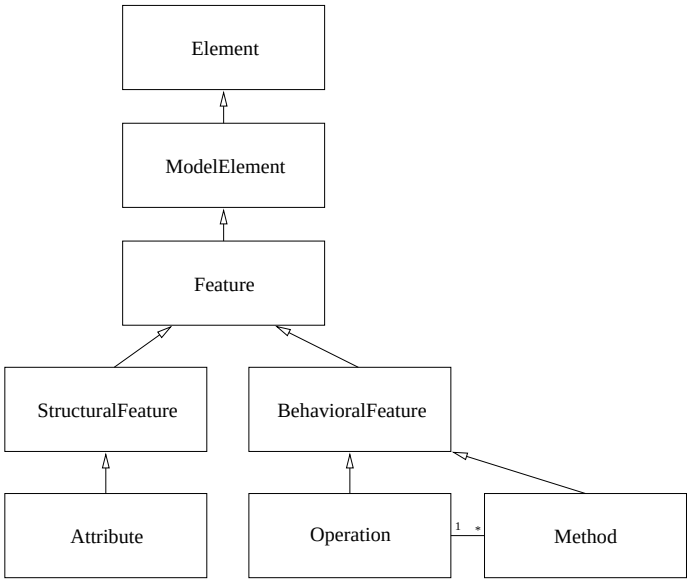


Figure 3.15: Current UML core metaclass backbone

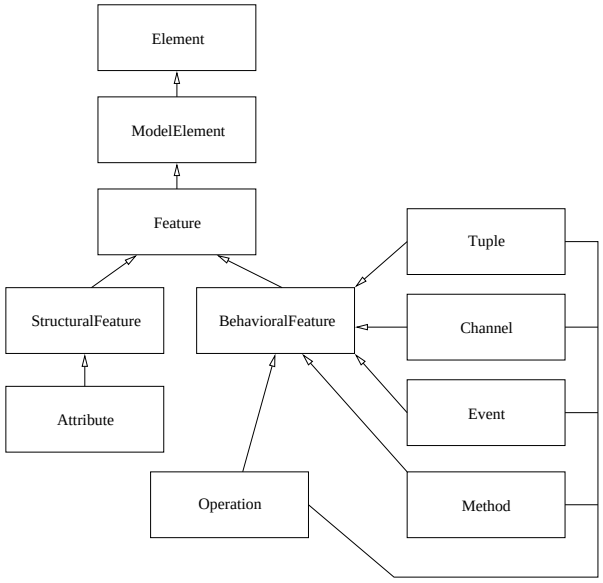


Figure 3.16: New UML core metaclass backbone

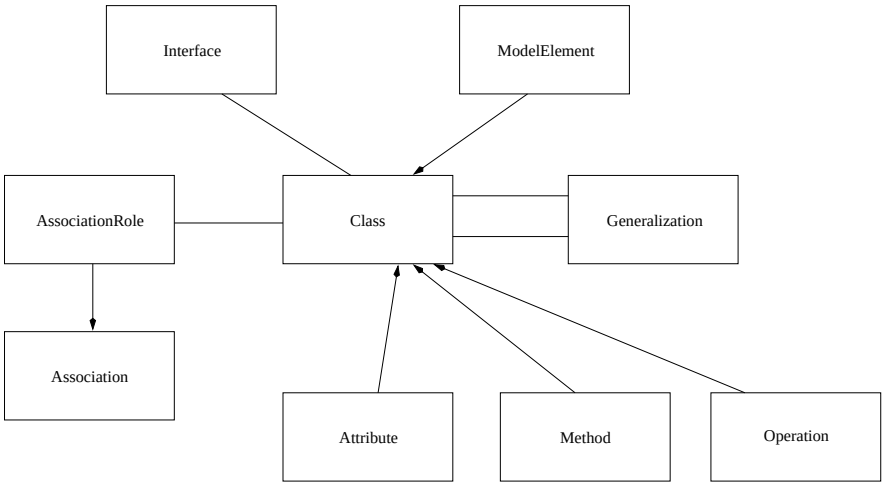


Figure 3.17: Old class-centric metamodel

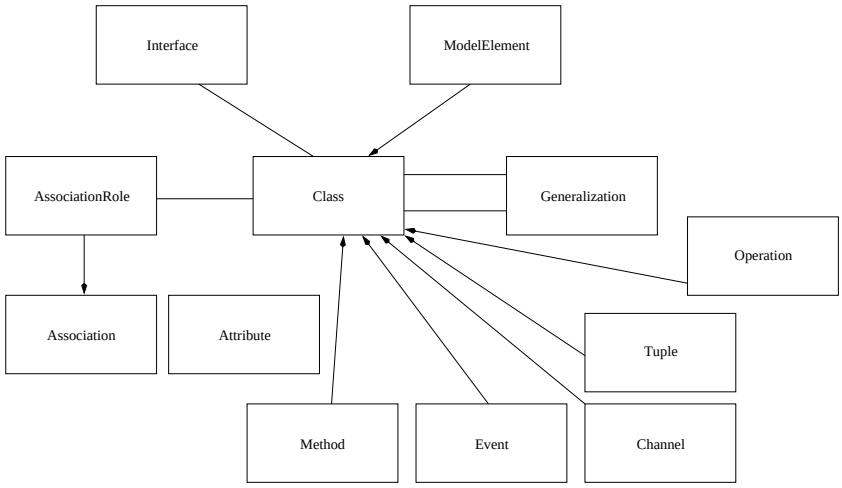


Figure 3.18: New class-centric metamodel

This change impacts several aspects of the UML metamodel:

- First, I have introduced two new metaclasses, *Channel* and *Tuple*. These metaclasses must be fully defined, as per the UML metamodel definition. See the semantics definition provided in OCL later in this section.

- Second, the *Event* metaclass used to be a part of the *State Machines* package which is within the *Behavioral Elements* package. Since the *Behavioral Elements* package was dependent upon the *Foundation* package (in which all core elements reside), but not vice-versa, the move of *Event* from *Behavioral Elements* to *Foundation* will not adversely affect the metamodel.

Thus, the part of the metamodel related to the *Class* metaclass can now be rewritten from its existing structure as see in Figure 3.17 to the new structure, as seen in Figure 3.18.

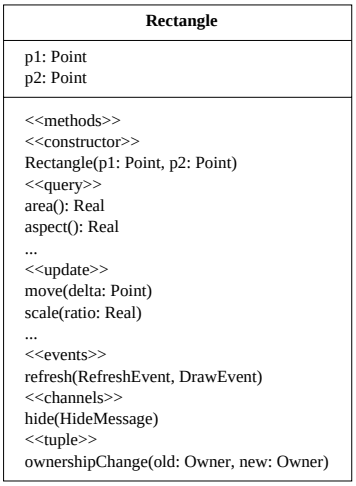


Figure 3.19: The specification of a class that has all four kinds of *BehavioralElements*

Operations and Interface Specifications. Note that *Event*, *Tuple*, and *Channel* are all still related to *Operation* in exactly the same manner that *Method* was. More precisely, all are an implementation of an *Operation*, specifying the algorithm or procedure that effects the results of an operation. Each has a *body* attribute that is the implementation of the element, represented as a *ProceduralExpression*. Additionally, each has a *specification* association that designates an operation that the element implements.

When I say an element implements an *Operation*, I mean that an instance receiving a message of the type of the *BehavioralElement* will cause the implementation to fire. For the various *BehavioralElement* types, this implementation is interpreted in different ways. For a *Method*, a method is invoked on the object in question. For an *Event*, either a method is invoked (in most event models) or a signal is raised (in older systems). For a *Channel*, either a message is inserted into a queue or a *read* action completes. Finally, for a *Tuple* element, either an object is inserted into the tuplespace or a *read* action on the space completes.

Representation of Behavioral Elements. Each of these behavioral elements should be represented in the same manner, since they are all analogous, in a model diagram. The standard means for specifying methods on an object is simply listing them in the operation part of the class element, as in our previous examples of class diagrams.

Each of our new *BehavioralElements* also has a signature. In each case, there is a destination-aspect of the message and a set of types of messages that can be accepted. I will designate this with a signature similar to that of a *Method*, but lacking in the implicit order found in the parameters of a *Method*. Additionally, I suggest specifying *Methods* in the same manner, with the implicit assumption that the parameters are ordered and necessary, unless otherwise specified. More on this point in Sections 3.4.3 and 3.4.6.

An example generalized specification of a class that accepts all four kinds of messages can be seen in Figure 3.19. Note that I have used stereotypes to designate the type of each set of *BehavioralElements*.

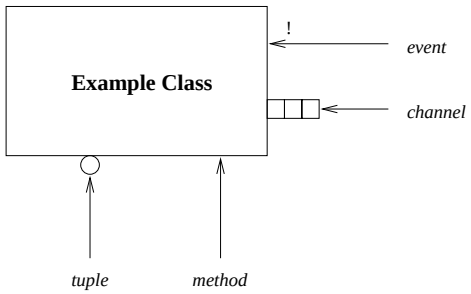


Figure 3.20: New *BehavioralElement* syntax

While this unification of syntax is elegant, it does not provide a representational advance. In fact, such a specification is longer and less clear than the original diagram. So, an alternative specification syntax is necessary for each of the four *BehavioralElement* types. Our suggested syntax can be seen in Figure 3.20.

Finally, I must specify the specific semantics of the new metaclasses. This only requires us to widen the scope of the current semantic definition for *Method* to include *Event*, *Tuple*, and *Channel* as follows (in OCL):

Channel, Event, Method, Tuple

```
self.specification -> exists(op | op.isQuery) implies self.isQuery
self.specification -> forAll(op | self.hasSameSignature(op))
self.specification -> forAll(op | self.visibility = op.visibility)
```

This completes the summary of our modifications to the UML metamodel to support new behavioral interface specifications.

3.4.2 The Component Element

Problem Two: Core Component Representation. Components are fundamentally different than the standard classes and objects of object-oriented and object-based programming. Surprisingly, no existing modeling language provides the appropriate core construct for describing the software engineering component.

In fact, the *Component* element of UML has very weak semantics when it comes to specifying component-based software. More specifically, a *Component* metaclass in UML is defined as " ... a reusable part that provides the physical packaging of model elements." This means that the UML designers, in trying to create a compositional metalevel architecture (one that identifies "things" and how they can be "plugged together"), they ignore component architectures at the base (non-meta) level.

3.4.2.1 Missing Features in Component Specification

The primary features necessary in modeling components missing in current modeling languages are:

- *Outbound Interface.* All object-oriented modeling languages fully support the specification of the inbound interface of an object, sometimes called the "has" side of the interface. Surpris-

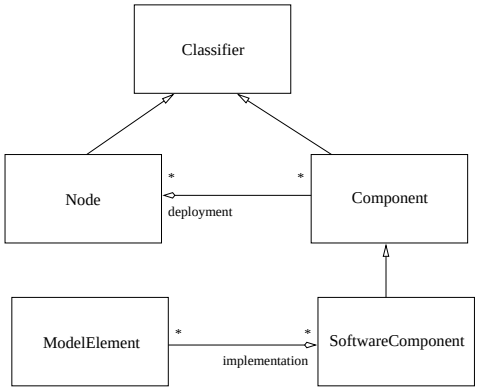


Figure 3.21: Extending the UML metamodel with the SoftwareComponent element

ingly, no language permits the user to specify the opposite, *outbound*, interface, sometimes called the "needs" interface.

Only Meyer's *Design by Contract* [121] method and process support this two-way specification at its core. UML can be extended to support such a description, and I will provide just such an extension.

- *Properties and Attributes.* Components that are objects have attributes that have special semantic value. These attributes are often called *properties*. Examples of properties and their related semantics can be seen in the JavaBeans [159] component model. UML does have properties defined as base constructs; they are used to extend the language.

  These properties can be mapped to the properties of component software, but lack the additional semantics that standard properties have in component software. I will define these additional semantics and apply them appropriately.

- *Events and Methods.* Component architectures use primarily use events to connect components together. Events are simply specially named methods in the JavaBeans model, but because they are events, they have extra semantics that methods do not have. I will make the corrections to the representational model's semantics to account for events in component models.

- *Dependencies and Associations.* Components can have subtle dependencies and associations not captured by the standard generalization and association constructs of UML. I will document some of these subtleties and provide the corresponding extensions to UML, primarily in Section 3.4.4.

I will represent a component, in the sense of a software component from a compositional software architecture, by extending the existing UML metamodel element *Component*. I will specialize the *Component* metaclass to a new *SoftwareComponent* metaclass that has additional semantics in order to deal with the issues mentioned above.

Before I can specify the additional semantics of *SoftwareComponent*, I must discuss how to visually represent this new construct.

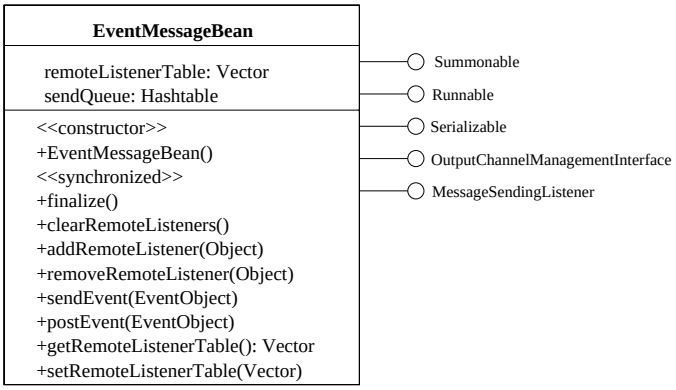


Figure 3.22: The EventMessageBean, specified in UML 1.1

3.4.2.2 Visual Representation of *SoftwareComponent*

The new *SoftwareComponent* metaclass has, in addition to the standard set of attributes and methods, a set of "outbound" behavioral elements. These are the behavioral elements on which the component *depends* in order to operate properly.

There are two parts to the specification of the "needs" elements of a component K. The first part is the macro-level specification, a list of the components $C = \{c_1, c_2, \ldots, c_k\}$ upon which K depends. The second part is the micro-level specification, a list $B = \{b_1, b_2, \ldots, b_k\}$ where $\forall\, b_i \in B : \exists! j \ni b_i \in c_j$ of the specific behavioral elements of the components upon which C depends.

These new "needs" elements are specified in one of three ways. Most simply, two new lower divisions of the standard *Class* "box" can be used. Optionally, the new elements can be specified through the use of a new stereotype. The synonymous $\ll$*outbound*$\gg$, $\ll$*needs*$\gg$, or $\ll$*dependson*$\gg$ are suggested. Choosing between the alternatives depends upon your project's terminology. Finally, dependency associations can be denoted with real association links. This last option is the most illustrative but depends upon extensions to UML presented in the next subsection, so I will defer the example until then.

Properties are attributes with special semantic value, depending upon your component architecture. I suggest using the same syntax for specifying properties as one denotes attributes, only changing the font's style or weight. I use **bold** text to denote properties that are read-only, *italic* text to denote properties that are write-only, and ***bold-italic*** text to denote properties that are read-write.

3.4.2.3 An Example: The EventMessageBean

I will present a bean called the *EventMessageBean* as an example of these modeling techniques. First, the standard UML diagram for *EventMessageBean* can be seen in Figure 3.22.

In the next illustration, Figure 3.23, the same bean is presented, now with new class boxes for outbound interface dependencies and specification as well as annotated properties. Note that the class names specified in the "needs" divisions of the illustration are abbreviated. If there are duplicate class names within a development scope, full class names would be used instead.

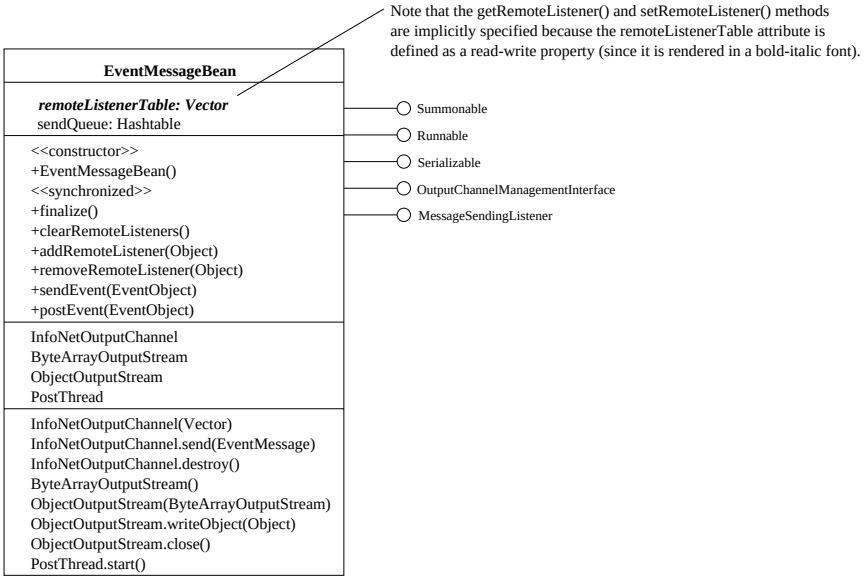


Figure 3.23: Using font changes to denote properties

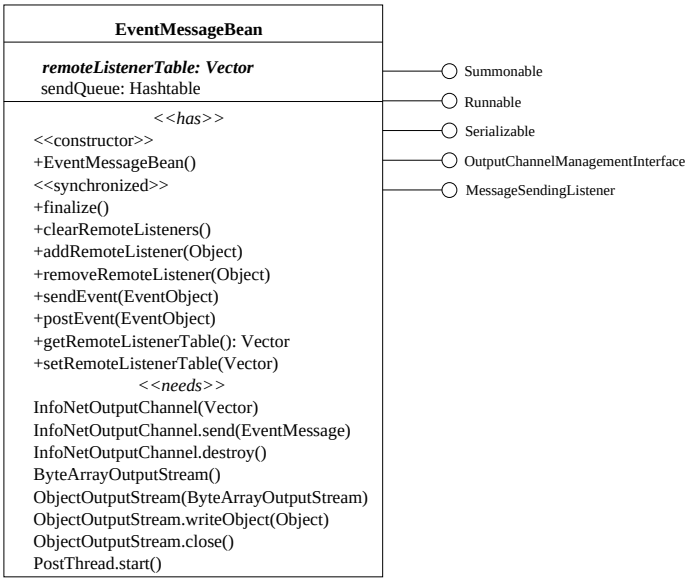


Figure 3.24: An example of using stereotypes to denote the full interface of a component

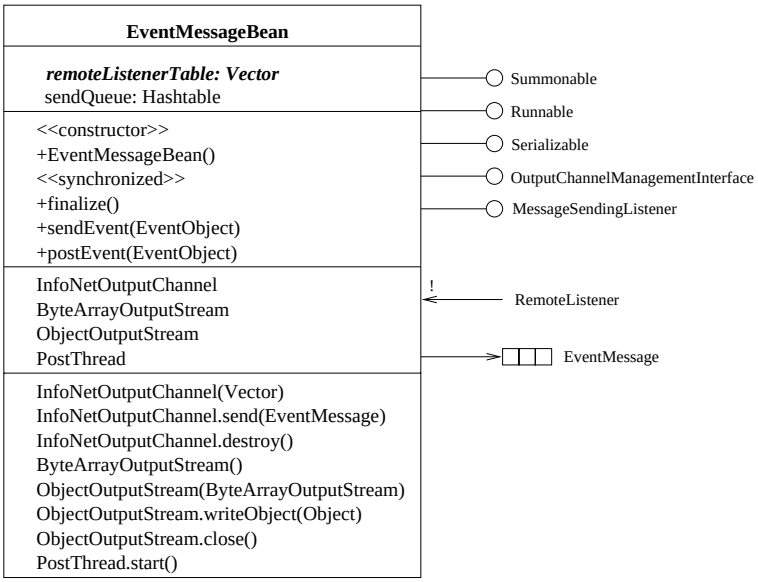


Figure 3.25: Using event behavioral elements to denote the full interface of a component

Next, in Figure 3.24, the same component will be specified with the use of stereotypes. Note how the component's presentation is more cluttered.

As mentioned above, events are used in many component models as the primary compositional mechanism. I have already discussed how to represent events with the *Event* element. Using the graphical event-based syntax as discussed above, this same component would be specified as in Figure 3.25.

Component Associations. Finally, components exhibit subtle dependencies and associations not captured with normal association models. UML's metamodel provides an *Association* metaclass which is specified as "defining a semantic relationship between classifiers". An *Association* has at least two *AssociationEnds* and an instance of an *Association* is a *Link*. Additionally, an *AssociationClass* metaclass is defined that is a multiple specialization of both an *Association* and a *Class*.

I can create new association types for our purposes with a stereotype applied to the *Link* instance of an *Association* or the instance of an *AssociationClass*. I prefer the second option because it means that associations, the relationships between components, are now first-class entities.

Our new association types will be fully described below in Section 3.4.5.

3.4.3 Partial-Interface Specification Stereotype

Problem Three: Partial Component Interface Specification. A component K depends upon a (potentially empty) set of other components $C = \{C_1, C_2, \ldots, C_n\}$. K depends upon, in turn, only *some* of the interface of each of these C_i components. I wish to describe and exploit this partial-dependency in component specification.

More specifically, each component C_i that K depends upon, has an interface specification I. I is composed of a set of behavioral elements $I = \{B_1, B_2, \ldots, B_{k_i}\}$. But, within the specification and

implementation of K, only some subset $I' \subseteq I$ is utilized. Only I' is necessary and sufficient for K to operate safely and correctly. Thus, we need a mechanism for specifying the partial-interfaces of components.

UML provides no such explicit mechanism. What's worse, UML provides no means by which one can specify whether a given construct is complete or not. Meaning, just because a class diagram of class k shows methods $\{m_1, \ldots, m_n\}$ does *not* mean that k has only those methods defined upon it. So, while we can perform partial specification in UML (indeed, it is done all the time), the semantic ground of such a specification is incorrect.

Catalysis provides *types*, as previously discussed, which go a long way toward solving this problem. *Types* permit the intentional partial specification of an interface because types are only a representational abstraction, *i.e.* they are independent of architecture and engineering constraints.

It must be noted that UML has incorporated the *type* construct, (through the use of a $\ll type \gg$ stereotype), but the semantics of this stereotype are ill-defined. This is likely the case because Catalysis was still in its infancy when UML was defined, and thus the formal definition of Catalysis-derived elements in UML are as of yet ill-defined.

Catalysis's type diagrams provide a means of specifying object state snapshots. Meaning, a type diagram specifies a potential object instance, its potential legal and illegal states, and its relationships with other objects. These diagrams are the first step toward the object network diagrams which I will discuss in Section 3.4.5.

What is still missing from type diagrams is completeness semantics. The completeness of the specification of an object in a type diagram is as ill-defined as in UML. This convenience leads to sloppy specification and the potential for large errors in design due to association and state oversight.

3.4.3.1 The Complete, Partial, and Sufficient Stereotypes

I introduce three new stereotypes, $\ll complete \gg$, $\ll partial \gg$, and $\ll sufficient \gg$, which will help specify partial interfaces for components. Before defining these new elements, I must modify the semantics of the base metaclass *Classifier*. See [41, pg. 29] for the full details of the semantics of *Classifier* upon which the following discussion is based.

The metaclass *Classifier* has four associations, *feature*, *participant*, *realization*, and *specification*. Each of these associations refers to a set of model elements, some of which are realized in the graphical representation of the *Classifier* specializations *Class*, *Interface*, and others, including our *SoftwareComponent*.

A number of metaoperations are defined on *Classifier* which provide access to these lists of related metaclasses. Examples include *allFeatures()*, *allOperations*, and *allMethods()*. I will use the metaoperations to define our two new stereotypes.

Classifier's semantics must be modified by conjoining a new predicate which describes the default state of the specification S of a *Classifier* with respect to our new stereotypes.

3.4.3.2 Default Completeness of Specification

The default state of any description of a classifier's associations is *complete*. More precisely, let $S = \{s_1, s_2, \ldots, s_n\}$ is a specification for classifier C. Each s_i is a specification of a single behavioral

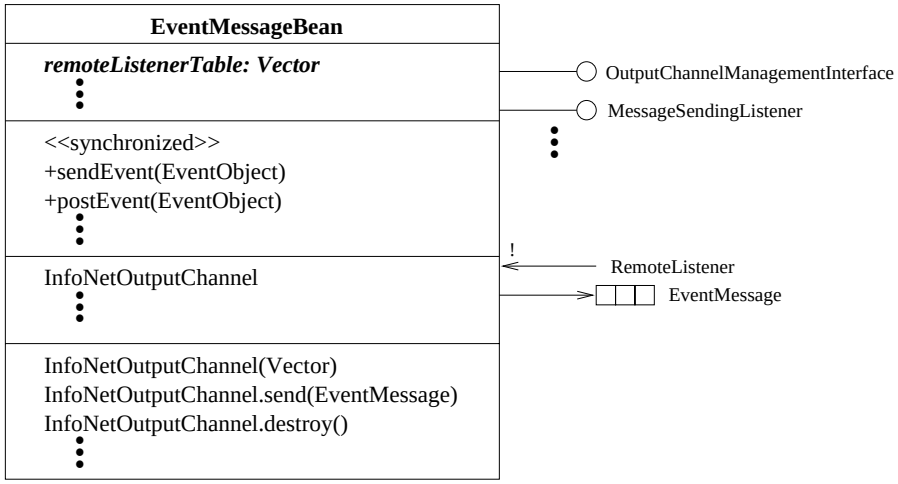


Figure 3.26: An example partial class specification using ellipses

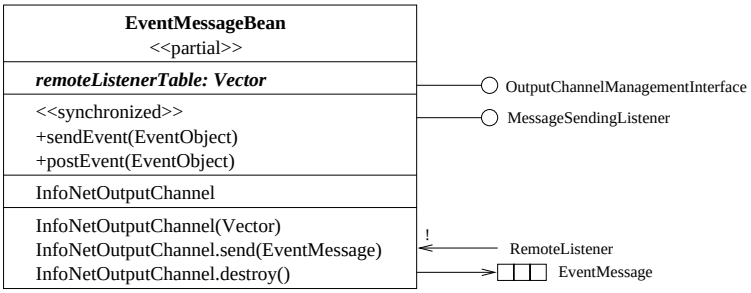


Figure 3.27: An example partial class specification using the $\ll partial \gg$ stereotype

element from one of the four association sets of C (*feature*, *participant*, *realization*, and *specification*, mentioned above).

If an association set A has *at least one* behavioral element specified, then by default, *all* behavioral elements of A must be specified within S. The default $\ll complete \gg$ stereotype is implicitly applied to the specification. It can be explicitly presented on a specification for clarity, but it is unnecessary.

3.4.3.3 Partial and Sufficient Specification

If a specification fails to fulfill this requirement, the it falls into one of two categories: it is either intentionally *partial* (e.g. an example), or is a *sufficient* specification.

If a specification is intentionally incomplete, it should be tagged with the $\ll partial \gg$ stereotype, or should use some other means of indicating that behavioral elements are missing (e.g. ellipses or the like can be used). Two examples of such a specification can be seen in Figures 3.26 and 3.27

A specification that is *sufficient* is one that is intentionally documenting *exactly those behavioral elements that are sufficient in the specification context.* Such a specification should be tagged with the $\ll sufficient \gg$ stereotype. An example of a partial specification can be seen in Figure 3.28. Note that in the left diagram, the *Hashtable* class depends upon the *Comparable* interface, whereas in the right diagram it depends only upon the partial specification of *Element*, namely the *compareTo()*

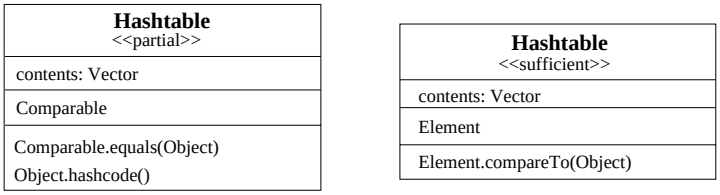


Figure 3.28: Two example class diagrams featuring the «*sufficient*» stereotype

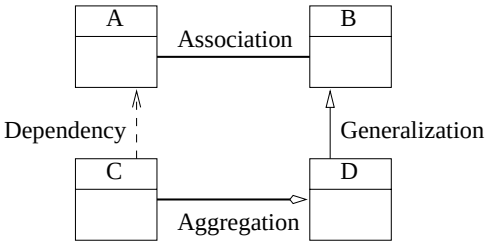


Figure 3.29: The basic forms of association

method.

Through the rigorous use of these new stereotypes, we can now define partial dependencies between components. Such stereotypes can be applied not only to all instances of the metaclass *Classifier*, to all *ModelElements* within UML. The same application holds true for Catalysis and OOCL.

3.4.4 Refinements for Associations

Problem Four: Component Associations. The association-related metaclasses in the UML metamodel support the specification of several different kinds of relationships between model elements. The two base classes of associations are *Association* and *AssociationClass*.

An association is defined as "...a semantic relationship between classifiers...". The most commonly used associations (see Figure 3.29) are references, where a class A has an instance variable that refers to class B, and aggregations, where a class C contains and a class D. Several extra relationship types are also shown. In particular, C depends on A and B is a generalization of D.

All of these associations and relationships have refinements. There are several refined versions of aggregation in UML including *shared aggregation*, where elements can be members of several containers, and *compositional aggregation* in which containers "own" their parts. Generalizations have default constraints defined so that overlapping, disjoint, complete, and incomplete generalizations can be represented.

All of these association and generalization refinements are defined through the use of stereotypes and constraints. Most of the basic relationships for object-oriented systems are covered by the core-standard elements of UML 1.1. On the other hand, some of the more component and network-oriented relationships that exist in component architectures, especially distributed ones, can not be modeled in UML.

3.4.4.1 Relationships in Distributed Component Architectures

I will identify several kinds of relationships exhibited in distributed component architectures. Each will be denoted with the use of a stereotype when applied to an association link. Alternatively, an association link's graphical representation could be modified (line thickness, color, style, etc.). I only suggest using this approach with object associations within specific diagrams because I believe that there are already enough such styles in the core of UML. Adding many more would only increase the possibility for diagram misinterpretation and steepen the UML learning-curve.

The following is a list of all the association types I have identified:

- *Standard local reference.* (no stereotype necessary)

  A standard local reference is the normal reference type as defined in UML. E.g. a pointer in C or a reference in Java are legitimate standard local references.

- *Garbage Collector Reference Types* (e.g. Java 1.2)[3]

  - *Guarded reference.* (≪*guarded*≫)

    A guarded reference is the strongest type of reference object. If the garbage collector determines at a certain point in time that the referent of a registered guarded reference is no longer strongly reachable, then at some later time the collector will enqueue the guarded reference.

  - *Weak reference.* (≪*weak*≫)

    A weak reference is a reference object that does not prevent its referent from being made finalizable, finalized, and then reclaimed. If the garbage collector determines at a certain point in time that an object is no longer strongly reachable and has no guarded references, then at that time it will clear all weak references to the object, simultaneously and atomically from the standpoint of the program.

  - *Phantom reference.* (≪*phantom*≫)

    A phantom reference is a reference object that remains valid after the collector determines that its referent is otherwise eligible for reclamation. If the garbage collector determines at a certain point in time that the referent of a registered phantom reference is no longer strongly reachable, has no guarded or weak references, and has been finalized, then at some later time the collector will enqueue the reference.

  - *Soft reference* (≪*soft*≫)

    As the amount of memory available to the Java heap decreases, an instance of this class may be cleared automatically if its referent is reachable only via soft references and, perhaps, via some guarded, weak, or phantom references. Soft references are cleared in approximate least-recently-used order. A best effort is made to clear all soft references before the virtual machine throws an `OutOfMemoryError`.

- *Indirect association.* (≪*indirect*≫)

[3] All of the following definitions are taken directly from the JDK1.2beta3 javadocs for the `java.lang.ref` package. See the documentation on `java.lang.ref.Reference` for the definition of *strongly reachable*. The definitions are Copyright © Sun Microsystems, Inc.

An *indirect* association is exactly that; a association which is obtainable via one or more levels of indirection. Typical examples of indirect associations include (a) the standard pointer-to-pointer or ref-to-ref in C and C++ respectively, (b) a reference to an entry in a data-store of some kind (a database, a Web page element, etc.), (c) a reference to an active object which, when queried, will respond with the association in question. Indirect associations are usually only documented if the obtainable resource is reachable with a single level of indirection and is directly relevant to the construct being specified (i.e. not extraneous).

- *Renewable association.* (≪*renewable*≫)

  A *renewable* association is a weak reference "by name" to an entity which may not be immediately accessible. Such associations are used for cached objects and services, transient network links, and the like. The semantics of the "renewal" operation are such that the renewable entity can be retrieved in finite time by some system service. Examples of renewable associations include DNS names, URLs, URNs, etc.

- *Mobile association.* (≪*mobile*≫)

  A *mobile* association is a reference to a mobile entity. Such an entity might be a software agent, a mobile phone, an automobile, or even a battleship. Such associations are often *renewable* as well.

- *Constant association.* (≪*const*≫)

  A *constant* association is a reference which is immutable.

- *Channel association.* (≪*channel*≫)

  A *channel* association is a association between two components through a channel behavioral element. See Section 3.4.1 for more information on channels. An example of a typical channel association is any middleware layer that uses a connection-oriented mechanism.

- *Event association.* (≪*event*≫)

  A *event* association is a association between two components through a event behavioral element. See Section 3.4.1 for more information on events. An example of a typical event association are the associations connecting composed JavaBeans.

- *Method association.* (≪*method*≫)

  A *method* association is a association between two components through a method behavioral element. See Section 3.4.1 for more information on methods. Note that some object-oriented languages (e.g. Java) do not have method references. Others, e.g. C++ and CLOS, use them frequently.

- *Tuple association.* (≪*tuple*≫)

  A *tuple* association is a association between two components through a tuple behavioral element. See Section 3.4.1 for more information on tuples.

- *Reflective association.* (≪*reflective*≫)

A *reflective* association is a association between two entities obtained at runtime via a reflection mechanism. Relationships between JavaBean composition tools and the beans that they contain are reflective associations. Examples of reflective associations are those used throughout the CLOS runtime[102].

- *Meta association.* (≪*meta*≫)

  A *meta* association is a association obtained via one or more operations at a meta-layer of an architecture. Many of the associations in the implementation of CLOS are meta associations.

- *Semantic association.* (≪*semantic*≫)

  A *semantic* association is an association between entities that is obtained and supported by some kind of semantic information. I will discuss such associations in a bit more detail in Section 3.4.6. An example of such an association is two objects sharing knowledge which is utilized to realize a communication mechanism, such as Sims' SDOs [150].

- *Persistent association.* (≪*persistent*≫)

  The *persistent* association is an association which lasts across life-cycles of the participating entities.

Note that if an object A has any local association with another object B, it implicitly has a *standard local reference*. (Thus the syntax of the language — a standard association is visually represented by a solid line.)

3.4.5 Object Network Diagrams

Problem Five: Dynamic and Emergent Structures of Components. Components are connected together in a large number of ways, as discussed in the previous section. Collaborative collections of components, when viewed at a high level, have inherent structure; physical and virtual patterns, (in a visual sense), are exhibited by complex systems.

I will call these patterns *object* or *component networks*. I will describe these object networks in a new diagram type called an *Object Network Diagram*.

To efficiently describe these object networks, we need new representational constructs for components and associations. Existing constructs, that of the class boxes for classes, objects, and components, and the annotated directed line segments for associations, are too large and unwieldy for the large-scale structures we wish to describe.

The constructs that I am going to describe are primarily inspired by the modeling constructs used by chemists and biologists to describe molecular structures and reactions[100].

3.4.5.1 New Modeling Constructs for Object Network Diagrams

My new constructs for the entities that make up an object network are summarized in Figure 3.30. The novel constructs are defined as follows:

- *Agent.* An agent is any instance that has an autonomous thread of control and is acting toward the accomplishment of a particular goal. An entity that contains a thread, as in the agent example, means that the entity has its own thread of control (i.e. is not solely reactive).

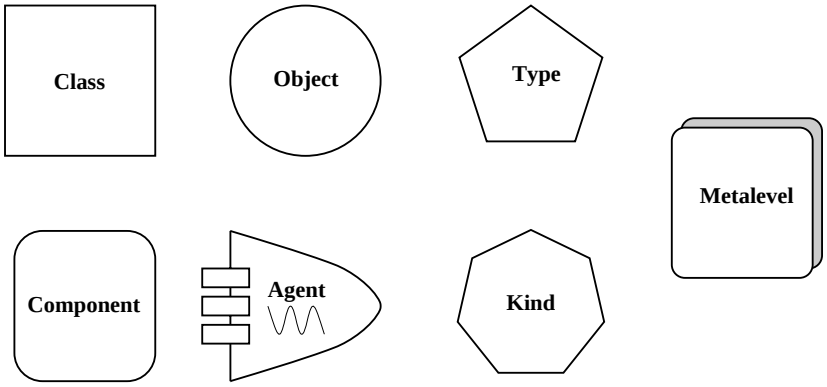


Figure 3.30: Summary of new representational entities for object network diagrams

- *Type.* A *type* is a metaclassifier for classes and objects (as per type theory; see [7, 8, 64, 72, 157, 165]). In some languages, like C++, type and class are equivalent.
- *Kind.* A *kind* is a semantic metaclassifier for types. See Section 3.4.6 for more information on *kinds*.
- *Metalevels.* Metalevel specification are denoted with the "ghosting" annotation as seen in Figure 3.30. The number of "ghosts" indicate the metalevel of the construct.

The new constructs for associations are based upon the association types described in Section 3.4.4. Because I want these diagrams to be as compact and informative as possible, I will denote the types of associations with sets of line segments (or curves) or varying thickness and color. These new relationship/association constructs are shown in Figures 3.31 and 3.32. Note that the constructs shown in Figure 3.32 are annotations that can be applied to all of the associations defined in the modeling language.

All of these association types were summarized in Section 3.4.4.

3.4.5.2 Object Network Diagram Semantics

An *Object Network Diagram* documents the associations between collections of components, primarily in distributed component architectures.

The compositional units in an object network diagram are recursive. *i.e.* components are potentially composed entities with aggregations, collections, etc. Normally, these compositions would be hidden with encapsulation, simplifying the model and the resulting design. When rendering an object network diagram these reusable subunits are often rendered in full detail. Maintaining this level of detail permits the designer to represent and recognize potential reuse within the system.

Typical *subunits* in standard systems are subsystems of reusable packages like collections, algorithms, and data-structures. Subunits in distributed systems come in many forms: system services, UI/client components, mid-tier business objects, data-stores, etc.

Each object network diagram has a *focus*. The focus of an object network diagram are those objects that are the primary objects being described in the diagram. The only elements that are *necessarily* represented in an object network diagram are only *exactly those objects directly associated*

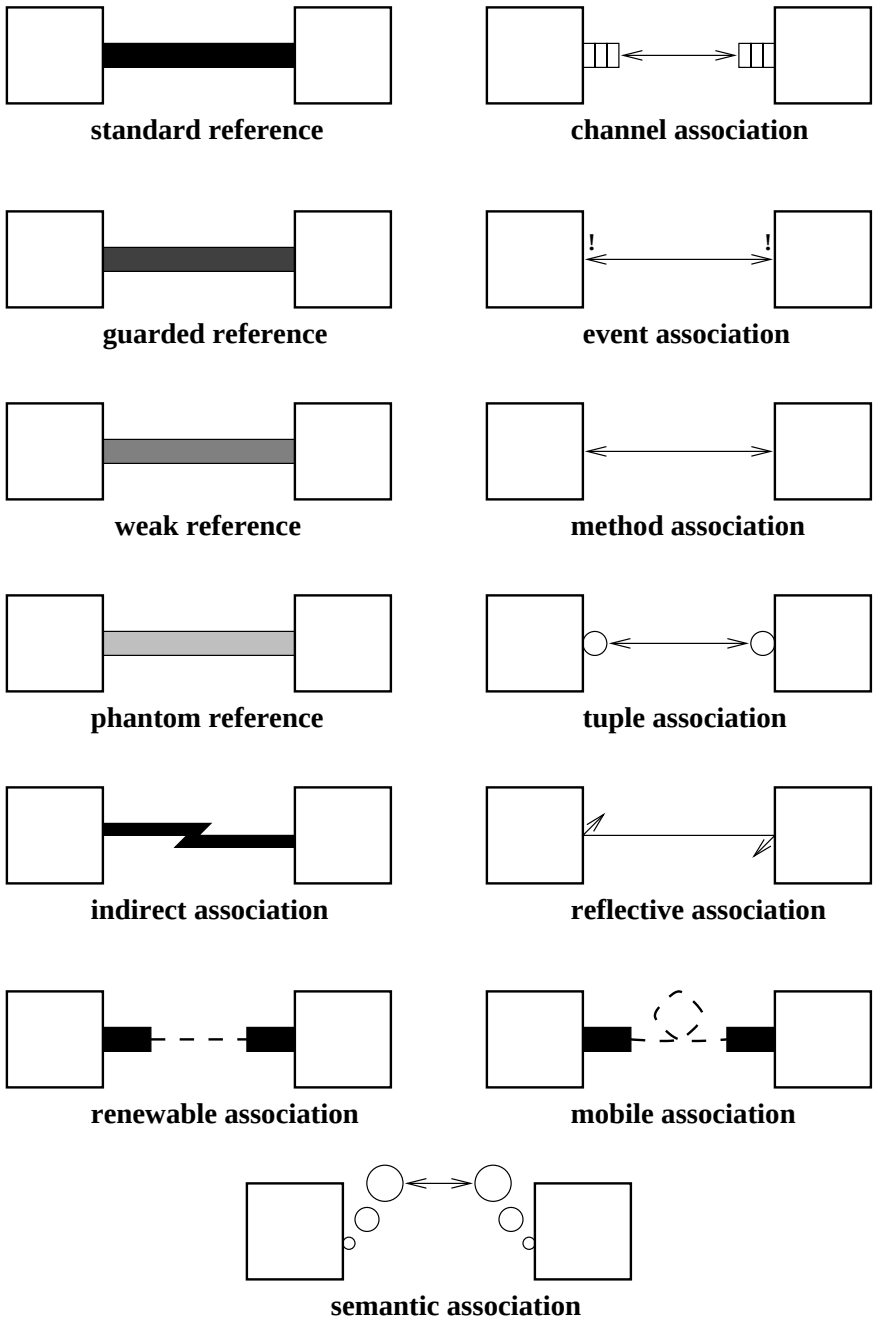


Figure 3.31: New representational entities for object network associations and relationships

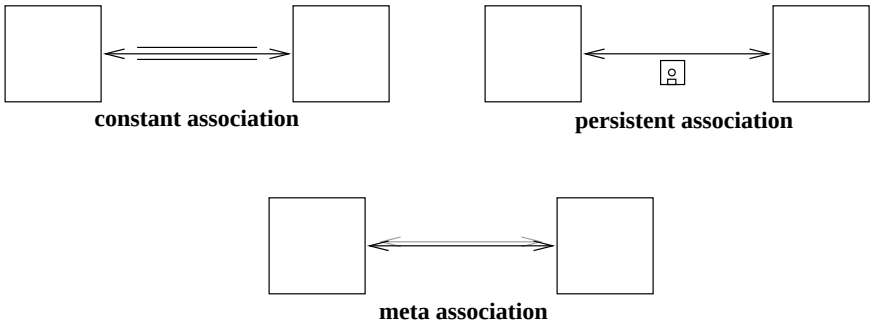


Figure 3.32: Three new annotation constructs for associations

with the focus objects. All other objects are extraneous and, while not necessary, might be useful in recognizing reuse, patterns, etc.

Entities within object network diagrams can also be annotated with *component valences.* The *valency* of a component is the number of associations unspecified in a given model. The valency of a component is indicated by annotating the entity name or construct with a small $+k$ where k is the valency of the entity. $+$ is shorthand for $+1$.

3.4.5.3 Example Object Network Diagram: The Web Architecture

An example object network diagram describing a typical Web client/server scenario is seen in Figure 3.33. The focus objects are the Web browser and the web server, "browser core" and "server core", respectively.

Through the rigorous use of object network diagrams and their related constructs, the specification of dynamic (distributed component) systems can be more efficient, clear, comprehensive, and flexible.

3.4.6 The Kind Stereotype and Role

Problem Six: Tying Knowledge/Semantics to Components. One of the next major steps in our research agenda is developing the theory and means by which formal semantics can be related to a knowledge representation and software instantiation of a system and its elements.

The *semantic* association type mentioned previously, and the hints at future work in Sections 2.8 and 6.2, begin to touch on the issues surrounding the need to attach precise and extensible semantics to knowledge-based component software.

The core element in our investigations in this domain is a new metaclass called *Kind.* A *kind* is the next level of abstraction above *type* — two levels above *class.* Two components are of the same *kind* if they are semantically compatible given a semantic context. This topic is within our primary research direction for the next several years.

In the short term, I will define a new stereotype $\ll kind \gg$ that can be applied to arbitrary modeling elements to denote semantic compatibility or interoperability. I will not provide any examples of such usage because the formal underpinnings of *kinds* are not yet fully realized and it is premature to provide concrete examples.

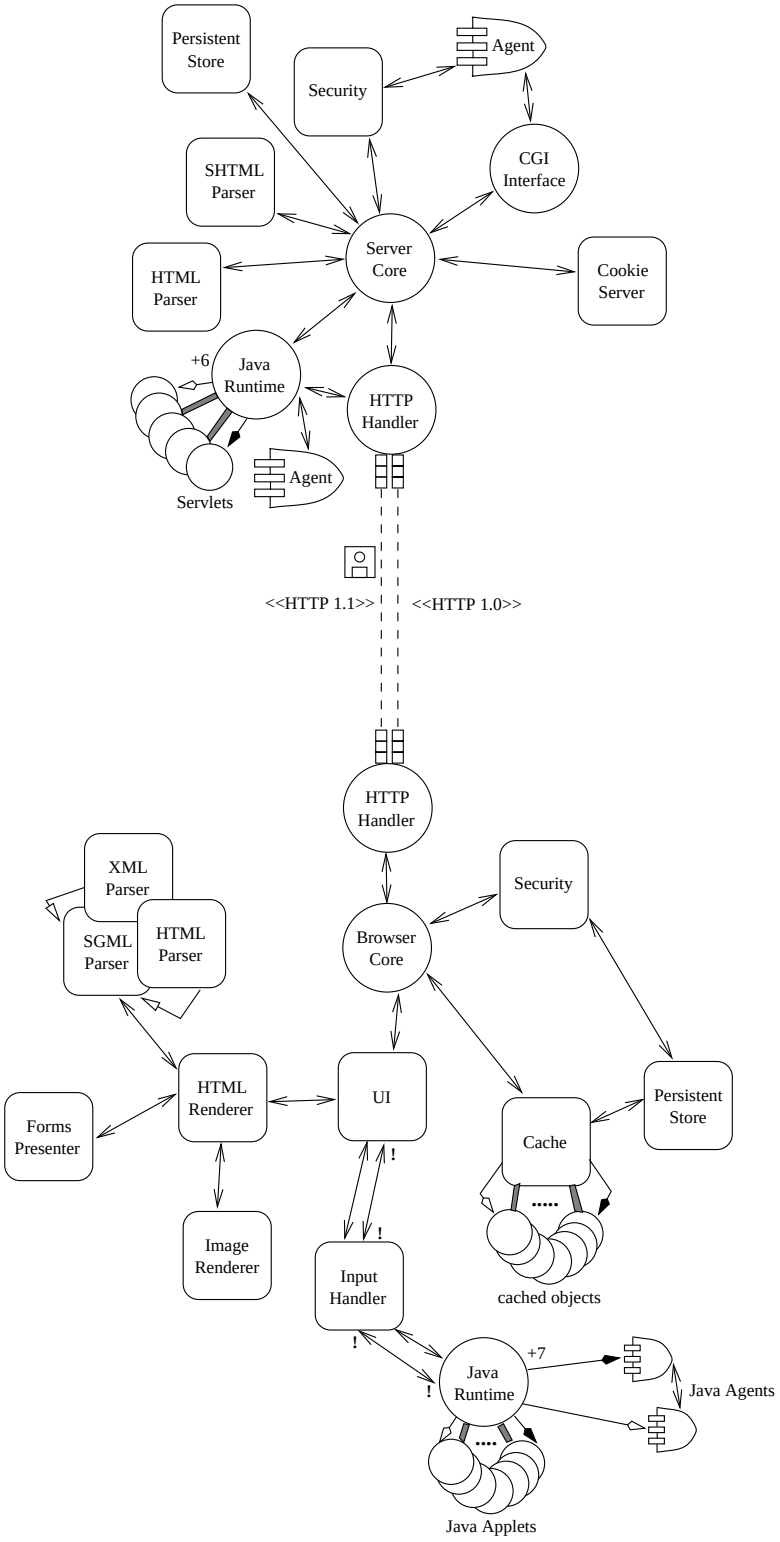


Figure 3.33: An example object network: a standard Web client/server system

This concludes our first set of extensions to current modeling languages that constitutes the base extension elements of DESML. In the next chapter I will describe the Infospheres Infrastructure, a framework we built to demonstrate component-based, dynamic, autonomous object systems.

Chapter 4

The Infospheres Infrastructure

The first version of the Infospheres Infrastructure, (II for short), is a first-pass design and implementation of a component-based, dynamic, autonomous/active object infrastructure. The II provides a framework for building dynamic distributed systems composed of active Java 1.0 objects that communicate using messages.

The II 1.0 is an extensive framework. Since this document is meant to focus specifically on modeling dynamic distributed systems, I will only briefly describe the core and relevant aspects of II 1.0 here. The II 1.0 is documented in full in the Infospheres Infrastructure User's Guide [69] and is available for download via the group's Web site at `http://www.infospheres.caltech.edu/`[1].

4.1 Infospheres Infrastructure History

The II was designed and built in late 1996, thus exclusively used the Java 1.0 language [68] and technologies. Its initial version (the alpha release) was built by a group of undergraduates managed by myself. The infrastructure was then redesigned and completely rewritten, but for the `info.net` package, by this author. Other groups members also contributed a great deal of help with late design work, some implementation, and bug-tracking and fixing.

A new infrastructure, II 2.0, is being designed and built at this time (mid-1998). We have taken what we have learned in building II 1.0, added in new features found in Java 1.1 [10] and 1.2 [10, Appendix C], and incorporated a new object model that synthesizes the standard dynamic distributed system's object model with the Web object model. More information on II 2.0 can be found in [70] and [71].

4.2 The Infospheres Infrastructure: Requirements Analysis

Our research goals dictate an infrastructure that must support the composition of distributed persistent opaque components with dynamic interfaces. Additionally, these components must be able to participate in both synchronous and asynchronous collaborations. I will briefly discuss these requirements here, then describe the system design.

[1] Portions of this chapter might be taken from previously published material including [26, 27, 28, 33, 69, 71].

4.2.1 Opaque Distributed Software Components

The only visible aspects of an opaque component are (i) its external interface, so that other components can connect to it, and (ii) a specification of the component. In a distributed system, component interfaces are specified in one of four ways:

Remote Invocation. The most prevalent interface specification method is procedure or method-based. Remote procedure call [156] (RPCs) have been used as a simple interface specification technique for many years. Remote method invocation [129, 179] (RMI — essentially, object-technology-centric RPCs) as a object specification technique has gained popularity with the rise in use of object-oriented languages, especially Java [158].

Events. Events are messages with extra-system semantic meaning. Events are gaining popularity as a general-purpose communication framework, especially in publish-subscribe and push technologies [32, 78, 105, 110, 145, 154].

Message-Passing. Specification with respect to a component's sending and receiving messages is an alternative technique [31, 33, 52, 81].

State-Space Operations. A more unusual but equivalent specification technique is to describe components with respect to the ways in which they can access and modify their environment's state-space, perhaps via Z coupled with Linda as in [22]. Most non-object-oriented specification languages fall into this category because encapsulation is not a ground concept of the specification language.

Each approach has advantages and disadvantages, but the specific form of the interface is less important than the fact that the component implementation is hidden. The infrastructure must support at least one of these methods of interface specification.

4.2.2 Dynamic Interfaces and Interactions

A component must be able to adapt to changing conditions in a computation. These include the addition of new components to the computation, temporary unavailability of communications resources, and other common situations which arise in Internet-based distributed systems. One way to deal with the dynamic environment is to allow a component to change its interface and connections to other components during the course of a computation, so we require that the infrastructure allow component interfaces and interconnections to be completely dynamic.

4.2.3 Modes of Collaboration

All components participating in a synchronous collaboration must be active concurrently. By contrast, components participating in an asynchronous collaboration need not be active concurrently; any given component may be quiescent and show activity only when necessary (e.g. a message arrives for it). The advantage of asynchronous collaboration is that the participating components need not hold resources concurrently, since they use resources only when they are computing. The disadvantage is that handling an incoming communication can be expensive, because the communication must be handled by a daemon that activates the quiescent component and then forwards

the communication. Because of this tradeoff, we require the infrastructure to support both synchronous and asynchronous interactions, allowing individual component application developers to choose whichever mode is appropriate for their application.

4.2.4 Persistence

Components must be persistent, because a collaboration involving a set of components may last for years. Rather than requiring a component to stay active for the life of its collaborations, it is advantageous to design the component system such that the life-cycle of a component is a sequence of active phases separated by quiescent phases. In such a system, when a component is quiescent, its state is serialized (and can, for example, be stored in a file) and the component uses no computing resources.

When a component is active, it executes in a process slot or thread and listens for communications. Components designed in this way are often quiescent for most of their lifetimes, so the fact that quiescent components use no computing resources allows many more components to exist in the network and system simultaneously than would otherwise be possible. The infrastructure must support the storage of persistent state information by individual components. In addition, it is desirable for it to provide some method of efficiently updating persistent state information, such as by saving only incremental changes.

4.3 The Infospheres Infrastructure: Design

In this section, I will briefly describe the Infospheres Infrastructure, version 1.0 [26, 33, 69], and show how it satisfies the requirements identified in section 4.2.

4.3.1 Infospheres Framework

The II framework employs three structuring mechanisms: *personal networks* enable long-term collaborations between people or groups; *sessions* provide a mechanism for carrying out the short-term tasks necessary within personal networks; and *infospheres* allow for the customization of processes and personal networks.

As an illustration of these structuring mechanisms, consider a consortium of research institutions working on a common problem. This consortium has a personal network composed of processes that belong to the infospheres of the consortium members. This personal network provides a structured way to manage the collection of resources, communication channels, and processes used in distributed tasks such as determining meeting times and querying distributed databases. Each session of this personal network handles the acquisition, use, and release of resources, processes, and channels for the life of one specific task.

Infospheres are discussed in detail as part of the user's guide to our framework [69]. Here, I will focus on the conceptual models for processes, personal networks, and sessions.

4.3.2 Conceptual Model: Processes

Processes are the persistent communicating components which manage interfaces and devices. In our framework, we call these processes *djinns*.

4.3.2.1 Process States.

A given process can be in one of three states: *active*, *waiting*, and *frozen*. An active process has at least one executing thread; it can change its state and perform any tasks it has pending, including communications. A waiting process has no executing threads; its state remains unchanged while it is waiting, and it remains in the waiting state until one of a specified set of input ports becomes nonempty, at which point it becomes active and resumes execution. Active and waiting processes are collectively referred to as *ready* processes.

Ready processes occupy either (i) thread groups within a Java virtual machine or (ii) process slots in the OS process table. Both can make use of other resources provided by the operating system. By contrast, processes in the frozen state do not occupy any active system resources and cannot make use of any other resources provided by the operating system. The only resource used by a frozen process is the storage space, such as a small file or a database entry, which holds process state information.

4.3.2.2 Freezing, Summoning, and Thawing Processes.

Each process has a *freeze* method, which saves the state of the process to a persistent store, and a *thaw* method, which restores the process state from the store. A typical process remains in the frozen state nearly all the time, and therefore consumes minimal system resources. In our framework, only waiting processes can be frozen, and they can be frozen only at process-specified points. Except for its persistent store, all system resources held by a process are yielded when its freeze method is invoked.

A ready process can *summon* another process. If a process is frozen when it is summoned, the summons instantiates the frozen process, causes its thaw method to be invoked, and initiates a transition to the ready state. If a process is ready when it is summoned, it remains ready. In either case, a summoned process remains ready until either it receives at least one message from its summoner or a specified timeout interval elapses.

4.3.3 Conceptual Model: Personal Networks

A personal network consists of an arrangement of processes and a set of directed, typed, secure communication channels connecting process output ports to process input ports. Its topology can be represented by a labeled directed graph, where each node is a process and each edge is a communication channel labeled with its type and the input and output ports connected by that channel. Since processes can freely create input ports, output ports, and channels, the topology of a personal network is completely dynamic.

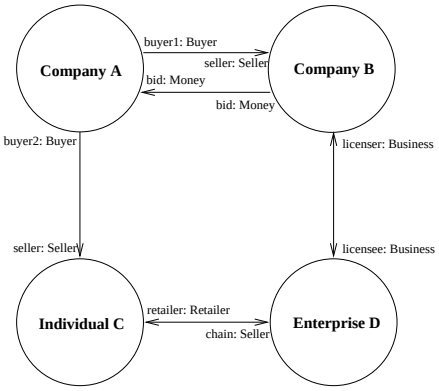


Figure 4.1: An example personal network

4.3.3.1 Communication Structures.

Processes communicate with each other by passing messages. Each process has a set of *inboxes* and *outboxes*, collectively called *mailboxes*. Every mailbox has an associated type and access control list, both of which are used to enforce personal network structure and security.

A connection is a first-in-first-out, directed, secure, error-free broadcast channel from the outbox to each connected inbox. In our framework, connections are asymmetric: a process can construct a connection from any of its outboxes to any set of inboxes for which it has references, but cannot construct a connection from an outbox belonging to another process to any of its inboxes.

Components also communicate with each other by sending and receiving *service requests*. Each component has the option of implementing a *service channel*. A service channel is a single point of contact for the component. We expect that the one-service-per-component model we have witnessed in component software to date will continue.

Example services include a word check for a spelling checker component, a date query for a clock component, a database lookup for a query component attached to a database, etc.

4.3.3.2 Message Delivery.

Our framework's communication layer, called *info.net* works by removing the message at the head of a nonempty outbox and appending a copy to each connected inbox. If the communication layer cannot deliver a message, it raises an exception in the sender containing the message, the destination inbox, and the specific error condition. The system uses a sliding window protocol to manage the messages in transit [135].

The communication layer eventually handles every message at the head of an outbox. The conceptual model uses asynchronous messages rather than remote procedure calls, because the range of message latencies across the Internet makes message passing with synchronous remote procedure calls impractical. However, the structure of our communication layer allows us to consider message delivery from an outbox to inboxes as a simple synchronous operation even though the actual implementation is complex and asynchronous.

4.3.3.3 Dynamic Structures.

A process can create, delete, and change its mailboxes, in addition to (as mentioned above) being able to create and delete connections between its outboxes and other processes' inboxes. The operation of creating a mailbox returns a global reference to that mailbox that can then be passed in messages to other processes. Since a process can change its connections and mailboxes, the topology of a personal network can evolve over time as required to perform new tasks.

When a process is frozen, all references to its mailboxes become invalid. This invalidation of mailbox references allows frozen processes to move and then be thawed, at which point the references to its mailboxes can be refreshed via a summons.

4.3.4 Conceptual Model: Sessions

A session encapsulates a task carried out by (the processes in) a personal network [**?**]. It is *initiated* by some process in the personal network, and is *completed* when the task has been accomplished. A later session with the same processes may carry out another task. Thus, a personal network consists of a group of processes in a specified topology, interacting in sessions to perform tasks.

4.3.4.1 The Session Constraint.

We adopt the convention that every session must satisfy the two part *session constraint*:

1. As long as any process within the session holds a reference to a mailbox belonging to another process within the session, that reference must remain valid.

2. A mailbox's access control list cannot be constricted as long as any other process in the session holds a reference to that mailbox.

The session constraint ensures that, during a session, information flows correctly between processes. An important corollary to the session constraint is that, because no valid references to their mailboxes exist, frozen processes cannot participate in sessions.

A session is usually started by the process initially charged with accomplishing a task. This process, referred to as the *initiator*, creates a session by summoning the processes that will initially participate. It then obtains references to their mailboxes, passes these references to the other processes, and makes the appropriate connections between its outboxes and the inboxes of the participating processes.

There are many ways of satisfying the session constraint. One simple way is to ensure that every process participating in a given session remains ready until that session terminates, and that once a process sends a given mailbox reference to another process in the session it leaves that mailbox unchanged for the duration of the session. Another approach is to have the initiating process detect the completion of the task using a diffusing computation or other common termination detection algorithm, after which it can inform the other session members that the session can safely be disbanded.

4.3.4.2 Example of a Session.

An example of a session is the task of determining an acceptable meeting time and place for a quorum of committee members. Each committee member has an infosphere containing a calendar process that manages his or her appointments. A personal network describes the topology of these calendar processes. A session initiator sets up the network connections in this personal network. The processes negotiate to find an acceptable meeting time or to determine that no suitable time exists. The task completes, the session ends, and the processes freeze. Note that the framework does not *require* that processes freeze when the session terminates, but that this will usually be the case.

4.3.4.3 Communication Within Sessions.

During a session, it is vital that the processes receive the quality of service required to accomplish their task. Therefore, communication is routed directly from process to process, rather than through object request brokers or intermediate processes as in client-server systems. Once a session is constructed, our framework's only communication role is to choose the appropriate protocols and channels. A session can negotiate with the underlying communication layer to determine the most appropriate process-to-process mechanism. While the current framework supports only UDP and native Java messaging layers (like RMI), the incorporation of alternative communication layers, like Übernet [180], iBus [115], Java ACE [95], or JSDA [21] is straightforward.

4.4 The Infospheres Infrastructure: Specification

I will provide a detailed specification of only one core subsystem (`info.djinn`) of II 1.0 here due to space considerations. I will informally describe several of the other system components of II to provide the reader with sufficient context for the details in Section 4.4.3.

4.4.1 The Messaging Subsystem

The messaging subsystem is contained in the package `info.net`. It provides a asynchronous, mailbox-based messaging system, much like the CSP [82] model, built on top of UDP.

The use-case and class diagrams for `info.net` are provided. The use-case diagrams are Figures 4.2 - 4.6

The partial class diagrams for `info.net` are found in Figures 4.7, 4.8, and 4.9. I only include a partial class diagram because the `info.net` package has thirty-eight classes. All core classes are included in these figures. Also, rather than show all the associations between the classes of `info.net`, I am only presenting the generalization relationships and full object specifications.

4.4.1.1 Main Classes

`info.net` has nine main classes:

1. *Broadcastbox.* `Broadcastbox` is a single `Mailbox` that maintains a set of destination places (see the definition of `Place` below). The user can add or remove places, and send messages to all of the places at once.

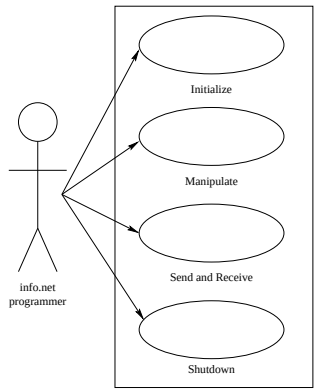


Figure 4.2: `info.net` use-case diagram — top level

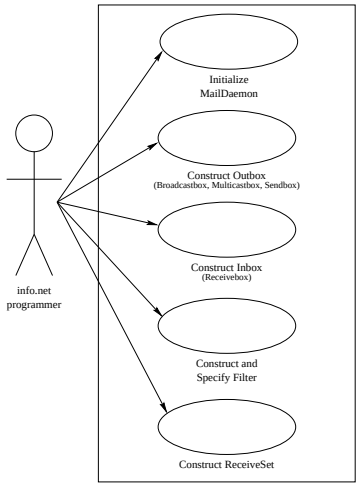


Figure 4.3: `info.net` use-case diagram — initialization

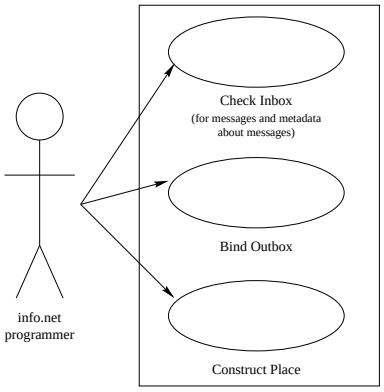


Figure 4.4: `info.net` use-case diagram — manipulation

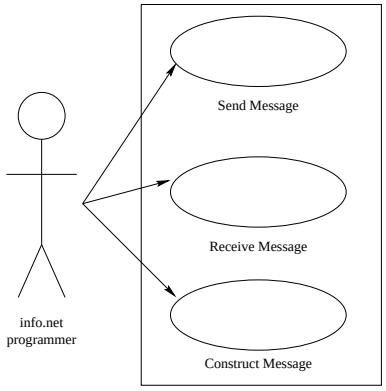


Figure 4.5: `info.net` use-case diagram — sending and receiving

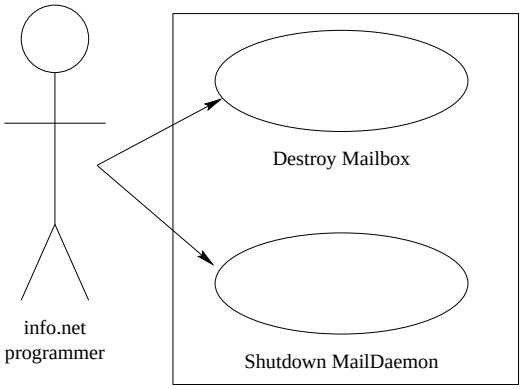


Figure 4.6: `info.net` use-case diagram — shutdown

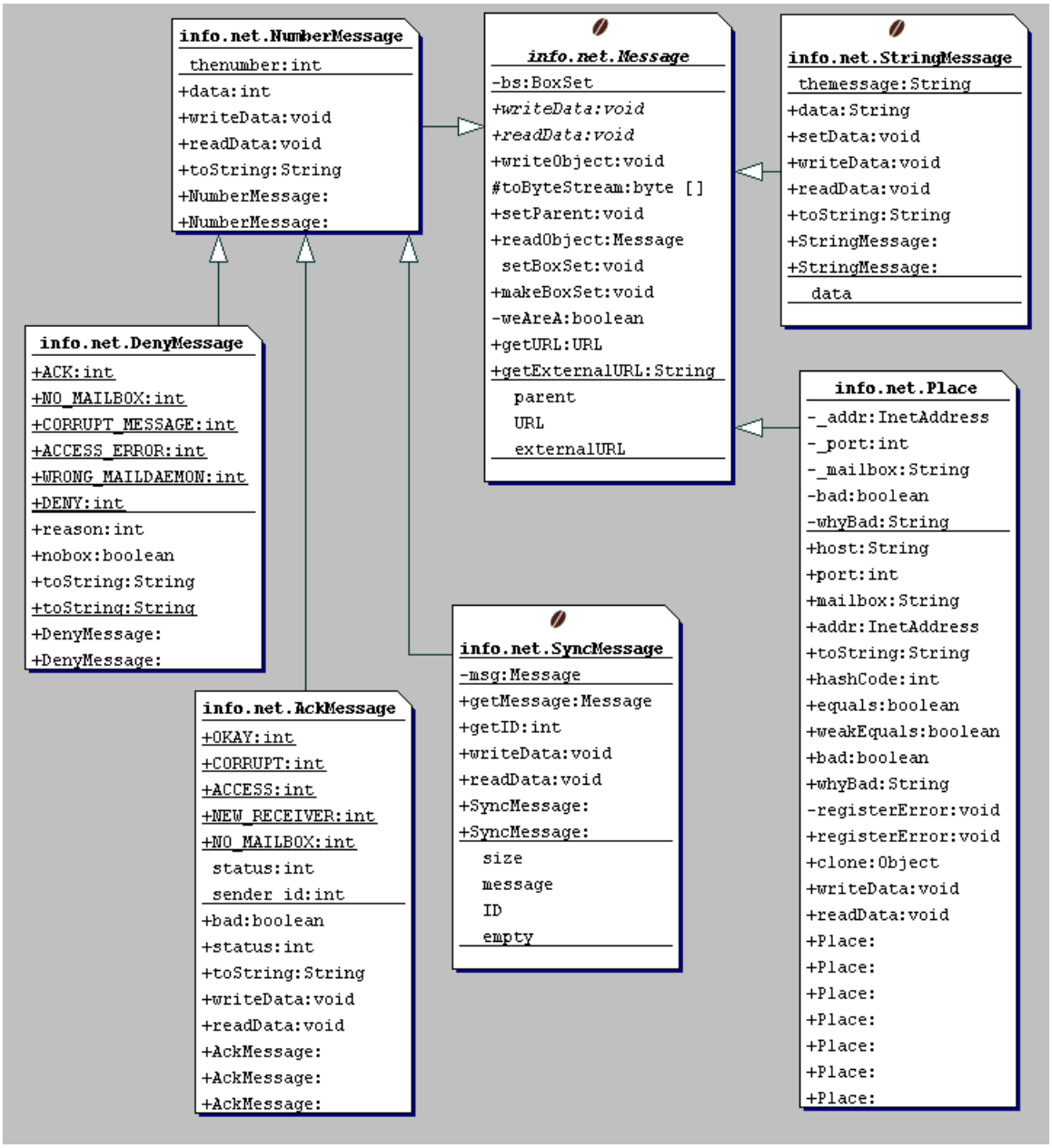


Figure 4.7: `info.net` class diagram (messages)

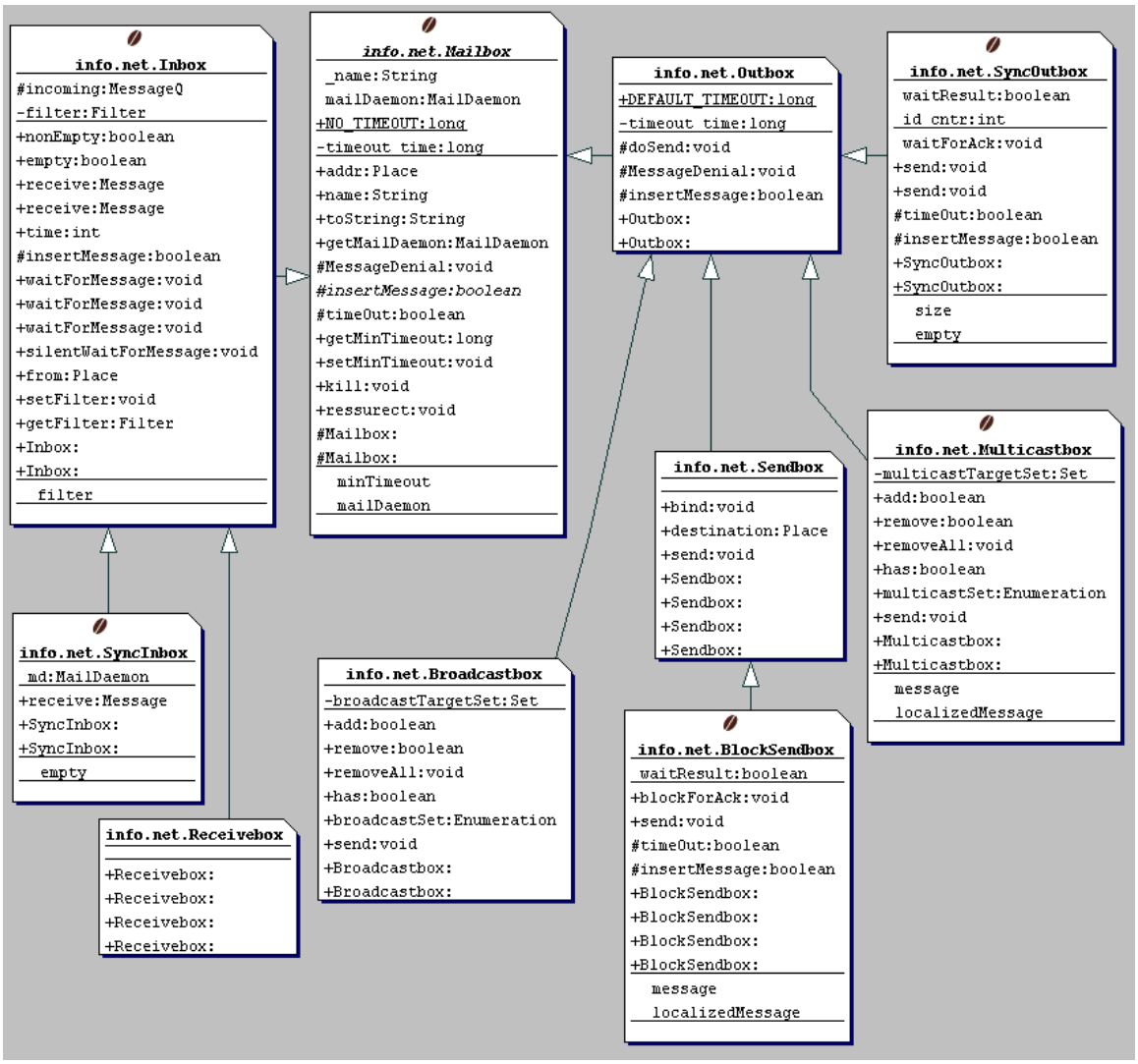


Figure 4.8: `info.net` class diagram (mailboxes)

Figure 4.9: `info.net` class diagram (core classes)

`Broadcastbox` ensures that no `Place` is duplicated in the set. Repeated adds will silently fail. On a send command, each `Place` in the set receives exactly one copy of the message.

The difference between a `Broadcastbox` and a `Multicastbox` (see below) is the protocol used for message transfer: unicast for the former, multicast for the latter.

2. *Filter.* The `Filter` class is used by inboxes to check messages before accepting them into the queue. Filters can be used in many ways. For example, one can ensure there are only `NumberMessages` in the queue or one might make the queue only accept messages from a specific `Place`. To install a `Filter` in an inbox, use its `setFilter()` method, and pass to it the `Filter` you want to use.

   The `CheckMessage()` method does the actual filtering. It takes a `Message` and the place that sent it, decides whether to accept or deny it, and returns a boolean based upon that decision. Thus, to install new behavior, a developer overrides the `CheckMessage` method. By default, `CheckMessage` calls `SendDenial` to send a `DenyMessage` to the sending mailbox of any message does not pass the `Filter`.

   Each `Filter` can be associated with one, and only one, `Inbox`. If you attempt to use the same `Filter` on more than one `Inbox` (either by using `setFilter()` or by construction), all `DenyMessages` generated by the `Filter` will have the most recent `Inbox` to which the `Filter` has been assigned as a source.

3. *MailDaemon.* The `MailDaemon` is the object that manages the mailboxes of a djinn. It performs reliable, ordered message passing and mailbox management on behalf of the djinn.

   Mailboxes in a djinn are registered with its maildaemon with mailbox constructors. The maildaemon then listens on a port for messages addressed to its incoming mailboxes, and processes those messages by putting them in the inbound queues of the appropriate incoming mailboxes. It also delivers messages from the outbound queues of outgoing mailboxes to the appropriate destination mailboxes which are often on a different host.

4. *Message.* `Message` is the abstract class that is the parent of all objects sent through the mailboxes. To create a new message class, a developer inherits from this base class to create a new public child class. The public methods `writeData()` and `readData()` must be overridden so that the new message is serialized and deserialized properly. In addition, the class needs a public default constructor that takes no arguments.

   Any message class can have a URL associated with it. These URLs are sent along with the messages to serve as a globally unique ID. This ensures that if a message arrives at a location it is either recognized correctly or is recognized as being an unknown message type.

   If a message of unknown type arrives, a remote class loader is invoked by the maildaemon and the class associated with the message and URL is loaded remotely, if possible. Before such drastic actions are taken, however, the URL is checked with the `Message` class's static `URLStringSecurity` object.

5. *Multicastbox.* `Multicastbox` is a single `Mailbox` that maintains a set of destination places. The user can add or remove places, and send messages to all of the places at once. `Multicastbox`

ensures that no `Place` is duplicated in the set. Repeated adds will silently fail. On invocation (send), each place in the set receives exactly one copy of the message.

As mentioned above, the difference between a `Broadcastbox` and a `Multicastbox` is the protocol used for message transfer: unicast for the former, multicast for the latter.

6. *Place.* The `Place` object serves as a unique name for mailboxes of all sorts. Its main use is to uniquely name inboxes: a given mailbox has a unique `Place`.

   A `Place` has three components: a machine address (i.e. an `InetAddress` object like "frankie.cs.caltech.edu"), a port number, and a mailbox name (a `String` object). There are many different constructors for a `Place` given the variety of contexts in which it will be used. A `Place` object is immutable.

7. *Receivebox.* `Receivebox` receives messages from a maildaemon and inserts them in a queue. Messages can be removed from the inbound queue using the `receive()` method. If there are no messages in the queue, `receive` blocks until there are. There are other methods that allow greater control over the receivebox, such as the `empty()` method that checks whether the queue is currently empty, and the `waitForMessage()` method that waits for a message to arrive in the queue.

   In addition, two other variables are associated each `Message`: the local time stamp (which allows for complete ordering of all messages received locally by time of arrival), and the return address (which is the address of the mailbox that sent the message). Use the `time()` and `from()` methods, which act on the oldest message in the queue, to access these two variables.

8. *ReceiveSet.* `ReceiveSet` collects multiple inboxes in a set. One can wait for a new message on the entire set, or get the oldest message in the set of queues associated with the inboxes in the receiveset. Mailboxes can be added and removed from this set, allowing for a dynamic collection of receive mailboxes.

9. *Sendbox.* A sendbox class is an outbound channel endpoint, exclusively supporting point-to-point communication. Once connected to a receivebox using the `bind()` method, a sendbox can send messages to the bound receivebox. At any point, a program can dynamically re-bind to another receivebox using the `bind` method. Any messages sent from that point onward will be sent to the newly bound inbox. Bindings are accomplished by passing the `bind` method a `Place` object that has the address of the desired inbox.

In summary, `info.net` permits our components to exhibit dynamic communication interfaces (the Mailboxes) which can appear, disappear, and change their behavior over time. Note that modeling the semantics of mailboxes (and arbitrary messaging layers) requires the use of the new component interface specification stereotype, as described in Section 3.4.3.

For the reader interested in more information about the `info.net` package, please see [69, Chapter 3].

4.4.2 The Djinn Master

The *Djinn Master* is the personal mini-ORB that is at the core of the II run-time. It is a djinn (like any other component in the II) and is responsible for several services:

- *Instantiation.* When djinn A wishes to communicate with djinn B, it sends a summons message to the Djinn Master DM_B responsible for B. DM_B is responsible for determining if the summons is valid, performing a lookup on the summoned djinn's implementation and metadata, instantiating, initializing, and starting the djinn.

  The metadata associated with B consists of (1) its implementation repository location, (2) its implementation name, (3) its run-time instantiation mode, and (4) its run-time configuration mode.

  B's run-time instantiation mode has two possible states. B can run either (1) as a set of threads, organized in a Java `ThreadGroup` within DM_B (called *thread* mode), or (2) it can run as an independent process within the host machine's operating system (appropriately called *process* mode). Because all communication between A, B, and DM_B is accomplished with messages, (there are no shared memory segments, references, etc.), these two execution models are indistinguishable at the code level.

  B's run-time configuration mode can be in one of three states:

  1. *Instance Mode.* B can run in *instance* mode. In *instance* mode, each and every summons message arriving at DM_B causes DM_B to instantiate a new instance of B. This is implemented in the natural manner (if the djinn is in *thread* mode a new threadgroup and set of threads are constructed and started, otherwise a new process is executed by DM_B).
  2. *Server Mode.* The second mode is called *server* mode. In *server* mode, new summons arriving at DM_B for djinns of the same name as B causes DM_B to "virtually" instantiate a new thread of control *within* B.

     This "virtual" instantiation is accomplished by either forwarding the summons message to B, whether it is running in *thread* or *process* mode. The `info.djinn` package handles this forwarded message and instantiates a new thread within the context of B to handle the invocation.

     Server mode is primarily used for those objects which provide a specific (and usually singular) *service* to one or more other objects. Its threads of control *can* share state and resources because they are all within the same threadgroup.
  3. *Mutual-Exclusion Mode.* This mode (which we call *MUXL* for short) provides a distributed monitor-like capability for our djinns. If a MUXL djinn B is running when a summons message arrives for it (either via DM_B or directly to B) it is rejected with a "djinn is busy/locked" message.

  With these six degrees of freedom for the life-cycle and base behavior of the djinn, we have been able to implement a tremendous variety of distributed objects and services.

- *Persistence.* When a djinn becomes quiescent (its threads of control go idle or exit), the djinn is automatically moved to persistent store. The Djinn Master is not responsible for initiating this freeze, but is responsible for identifying the store to the djinn when it is instantiated and knowing when a djinn is frozen or ready.

- *Message Forwarding.* As mention previously, there are instances when messages come in for a djinn B via its Djinn Master DM_B. Summons and service (see the next item) messages

need not have their destination djinn fully specified. A partial specification is valid only if it uniquely determines a djinn within the context of the receiving Djinn Master.

More precisely, a djinn's identity (independent of instantiation) is designated by a unique `DjinnTrueName`, and each instance is uniquely named by a `DjinnName`.

A `DjinnTrueName` DTN is a tuple

$$DTN = \langle DTN_E, DTN_N, DTN_O, DTN_D, DTN_U, DTN_d, DTN_l, DTN_V, DTN_v \rangle$$

with the following elements:

- DTN_E – *Author Email Address.* DTN_E specifies the contact address for the author(s).
- DTN_N – *Author Name.* DTN_N specifies the name of the author.
- DTN_O – *Author Organization.* DTN_O denotes the organization that the author(s) worked for when they wrote this djinn. Example organizations include: schools, research groups, companies, divisions, private foundations, self, etc.
- DTN_D – *DjinnName.* DTN_D is the name by which the organization and authors know the djinn. This name does not have to be associated with the actually name of the djinn's class. The djinnname is usually a unique name within the organization (a product name) and is slightly self-descriptive.
- DTN_U – *Djinn URL.* DTN_U lists the URL associated with this particular djinn, author, or organization, whatever is more appropriate.
- DTN_d – *Release Date.* DTN_d is the date that this djinn is released for use.
- DTN_l – *Release Level.* DTN_l is the release level of this djinn. A release level is usually a suffix that indicates, above and beyond the version number, exactly how stable this component is considered. Example release levels include: alpha, beta, gamma, final, fcs, a1, b2, etc.
- DTN_V – *Major Version.* The major version number for this djinn is stored in DTN_V.
- DTN_v – *Minor Version.* The minor version number for this djinn is stored in DTN_v. An example version number for a djinn would be *1.2*, where $DTN_V = 1$ and $DTN_v = 2$.

A DjinnName DN is a tuple

$$DN = \langle DN_d, DN_{DTN}, DN_l, DN_m, DN_n, DN_o \rangle.$$

with the following elements:

- DN_d – *Summoning Date.* DN_d is the date and time at which this particular instance of the djinn was initially instantiated.
- DN_{DTN} – *DjinnTrueName.* DN_{DTN} is a reference to the full `DjinnTrueName` for this instance.
- DN_l – *Instance Location.* DN_l refers to a `Place` on which this djinn is running.

- DN_m – *Master Location.* A djinn is associated with exactly one Djinn Master (the one that instantiated the djinn). DN_m holds the location of that Djinn Master's instance location so the child djinn can contact its parent master.
- DN_n – *Instance Name.* Each djinn is named locally by the user that installs and uses the djinn. These local, generally agreed-upon names help summons succeed from sources that have little or no information about the djinns that they are summoning.

  For example, a user might be running as their scheduling djinn a product called "MyScheduler, version 2.2" from the company "ScheduleIt!". Objects attempting to contact this scheduler need not know its version number, product name, etc. in order to communicate with it. E.g., each user in the network locally names the object that is responsible for scheduling appointments "Scheduler" so that communicating objects need not have *a priori* knowledge in order to interoperate.
- DN_o – *Instance Owner.* DN_o is the username of the owner of the djinn.

If a djinn A attempts to summon a djinn B with DjinnName DN_B and DjinnTrueName DTN_B, but only specifies an incomplete DjinnName (DN) and/or DjinnTrueName (DTN) as part of the summons (S), the following conditions are checked:

If

1. $((DN_n = DN_{B_n}) \wedge (DN_m = DN_{B_m}) \wedge (DN_o = DN_{B_o}))$
   (*i.e.* the instance name, master location, and instance owner of the summons is equal to the same of a running djinn), AND
2. $((DN_{DTN_V} \leq DTN_V) \wedge (DN_{DTN_v} \leq DTN_v))$
   (*i.e.* the version of the component in the repository is more recent than the one being requested), AND
3. the pairwise comparison between the E, N, O, D, U, d, and l elements of DN_{DTN} and DTN_B are sufficient to uniquely identify the same instance B, THEN

then

B is considered to be the djinn that A was attempting to contact with incomplete information.

Likewise, summoning relies on a similar "weak-lookup" specification so as to provide a kind of simple trader within each Djinn Master.

- *Service Invocation.* Service requests destined for a djinn but delivered to its Djinn Master are also resolved with the above algorithm. If a match is determined, the request is forwarded on to the running djinn.

Since the Djinn Master is itself a dynamic persistent active object, it can be summoned and manipulated like any other djinn. The partial class diagram for the `info.master` package is included in Figure 4.10.

For the reader who is interested in more information, please see [69, Chapters 4 and 5].

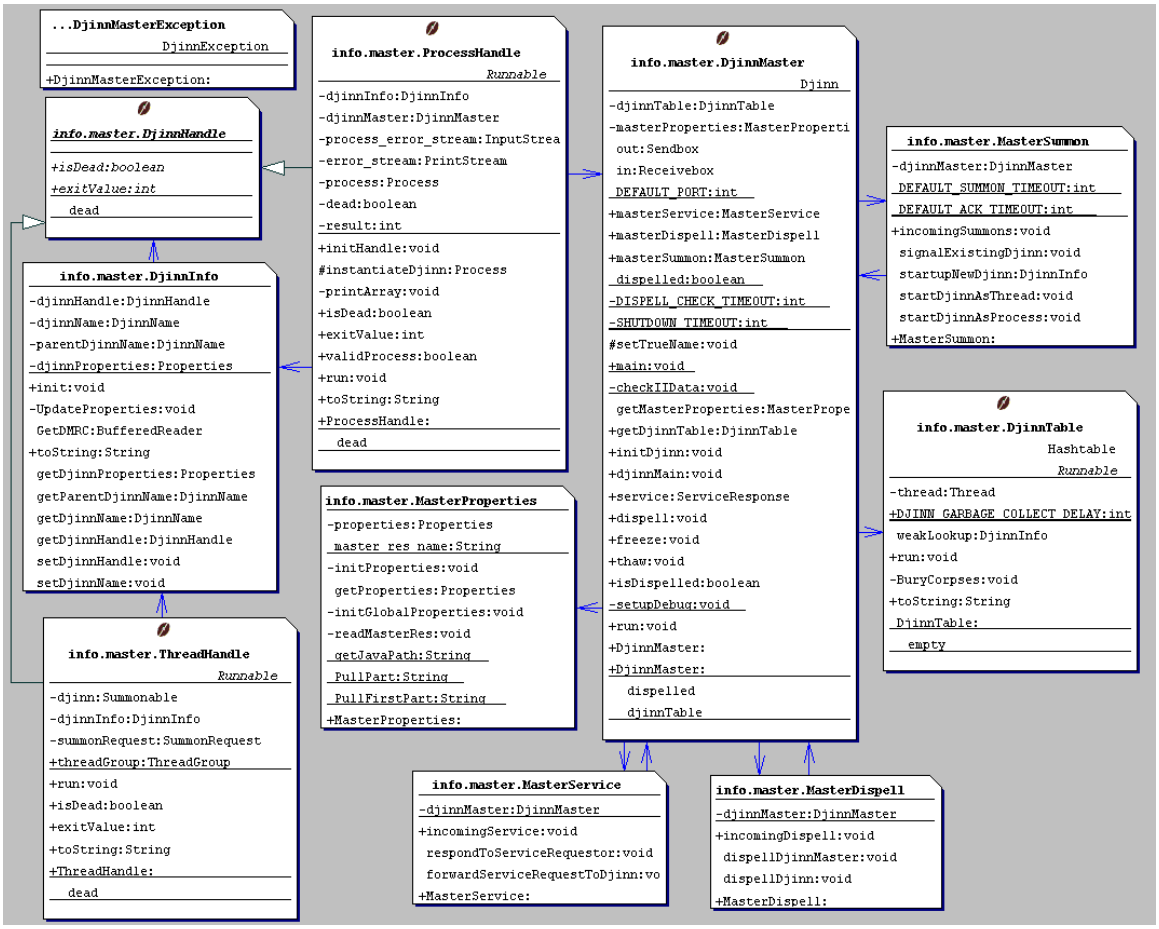


Figure 4.10: `info.master` class diagram

4.4.3 The Core Persistent Communicating Component: The Djinn

I will describe the core subsystem, that of `info.djinn` in greater detail. In particular, I will use DESML (our extension to UML as described in Chapter 3 to specify the high-level architecture of this package, given its complexity. I will not provide a detailed component-level specification (e.g. an Object-Z specification) beyond providing class and object diagrams, because of space restrictions.

4.4.3.1 `info.djinn` Overview

The full (ten thousand foot) class diagram for the `info.djinn` package is provided in Figure 4.11. It is evident that the package is tightly coupled and attempting to describe the myriad of interdependencies is not useful. This figure is has intentionally blank class boxes because the snapshot is taken at such a high level.

A closer view of portions of the package is available in Figures 4.12 - 4.18. Note that two classes are too large to fit on a single page in a readable font, so I have chosen to crop the lower part of the specification which shows less useful information (in particular, the class's parent's methods). The small "bean" image indicates that the annotated class has been identified as a potential JavaBean via reflection.

Several base classes for the *Summonable* interface are provided: *Djinn*, *DjinnBean*, *DjinnServlet*, and *DjinnApplet*. Each provides a base class from which a djinn developer should subclass, depending upon the demands of their application. Note that *Djinn* specializes *Message*, thus some groundwork for adding mobility to the II is already in place.

There are several classes which inherit from `info.net`'s `Message` base class, and they are organized in three groups: *Summon*, *Service*, and *Dispell*. Each group contains the messages (and exceptions) which are used to implement the group's functionality, as well as the core class which contains the logic to implement the functionality of the subsystem (named, appropriately, *Summon*, *Service*, and *Dispell*).

As mentioned previously, there are two classes which provide the naming service for the II, *DjinnName* and *DjinnTrueName*. Note that both are specialized from *Message*.

Finally, there are several service subsystems, each encapsulated in a few interrelated classes. These subsystems include *Persistence* (made up of *AbstractPersistence*, *AppletPersistence*, *Persistence*, *PersistenceException*), *Djinn Services* (consisting of *Service*, *ServiceException*, *BackgroundThread*), *User Interface* (including *AWTThread*, *IIWelcomeDialog*), *Session Management* (made up of *OutboxAssociation*, *OutboxAssociationTable*), and *Core Djinn Functionality* (consisting of *BackgroundThread*, *DjinnThread*, *Lamp*, *LampThread*) which provide the core of the framework.

4.4.3.2 The Persistence Subsystem

The *Persistence* service subsystem is responsible for helping save the state of a djinn to persistent store. This subsystem is made up of four classes: *AbstractPersistence*, *AppletPersistence*, *Persistence*, and *PersistenceException*.

I have begun to add support for persistent djinns that are both normal objects (*Djinns* and *DjinnBeans*) as well as objects that have special security models (*DjinnServlets* and *DjinnApplets*). This distinction inspired the creation of the abstract class *AbstractPersistence* that provides the

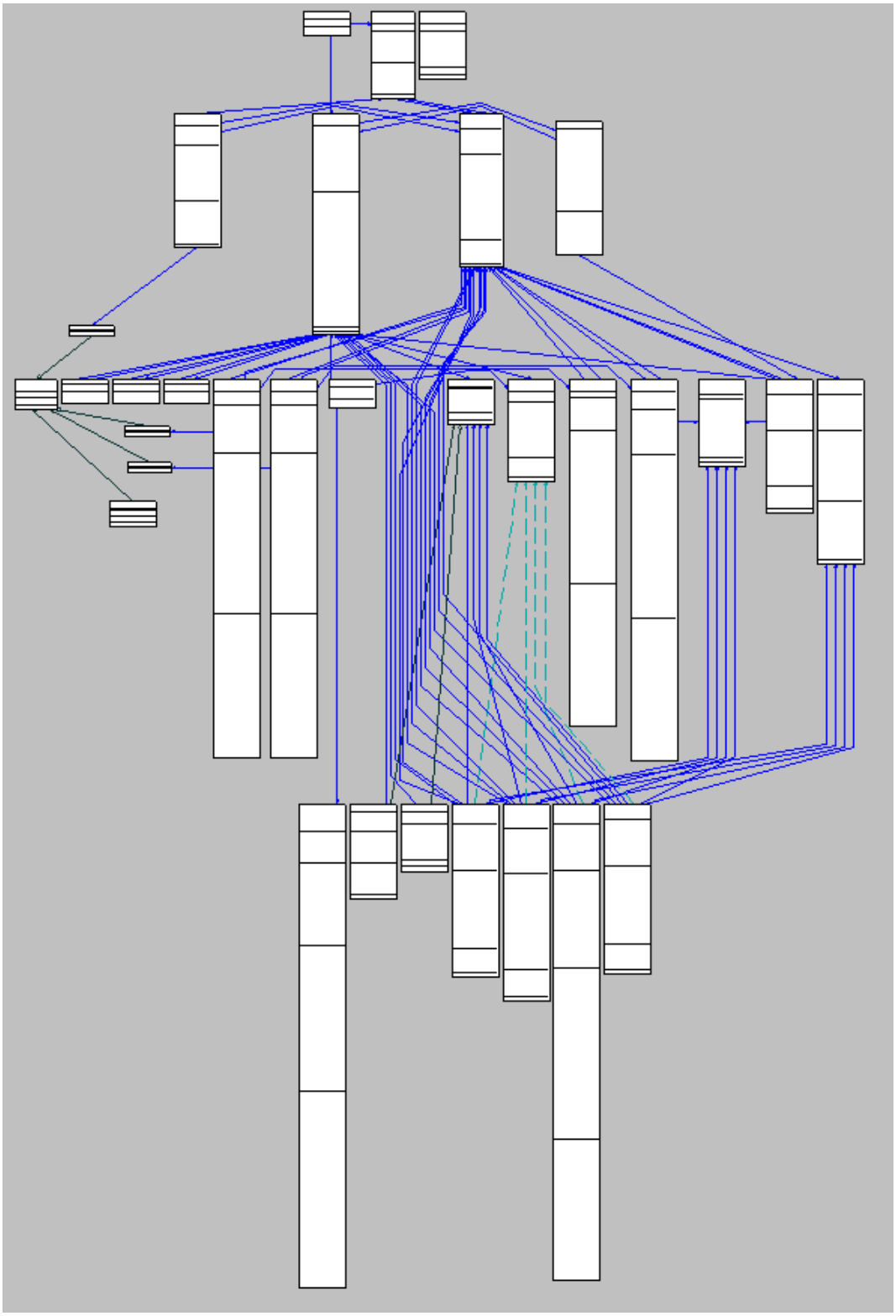


Figure 4.11: High-level `info.djinn` class diagram

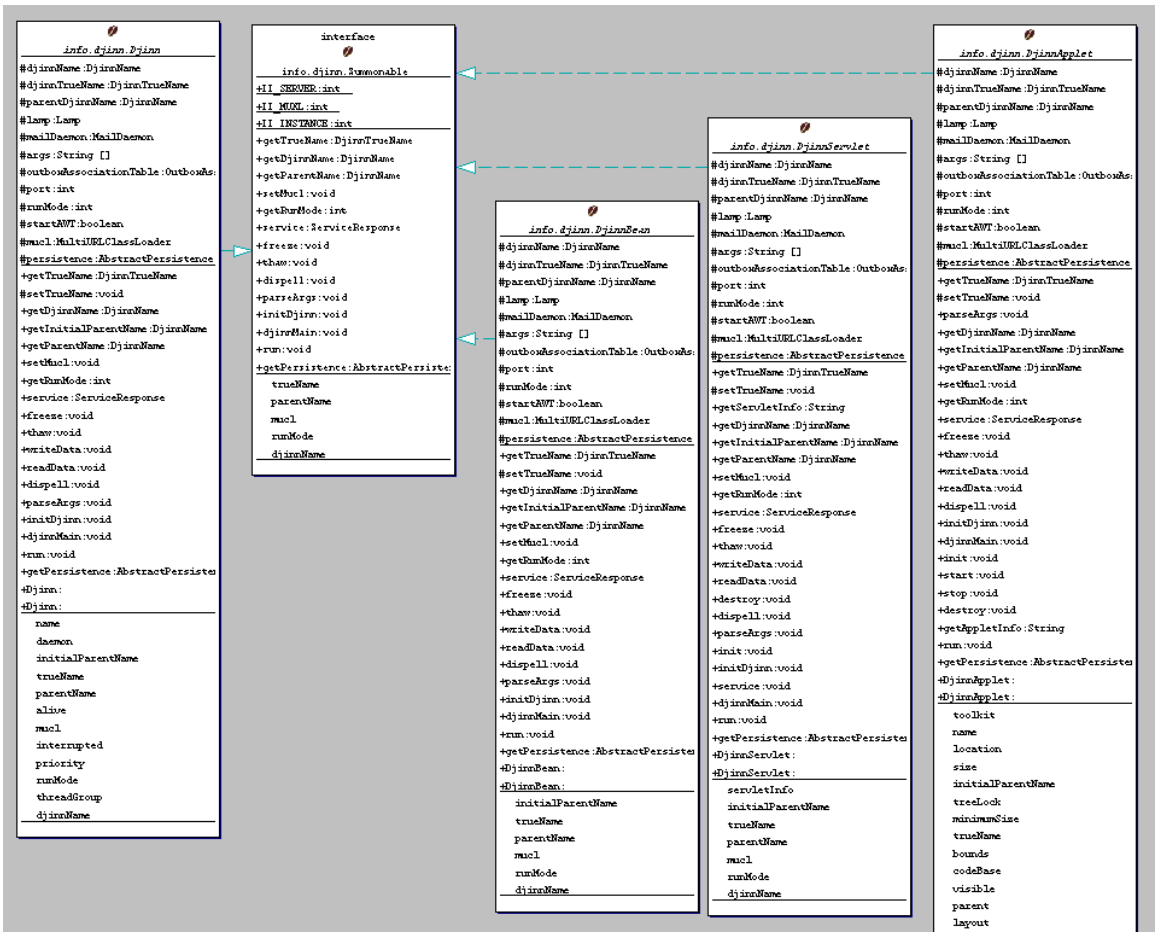


Figure 4.12: `info.djinn` class diagram (djinn base classes)

Figure 4.13: `info.djinn` class diagram (service subsystems)

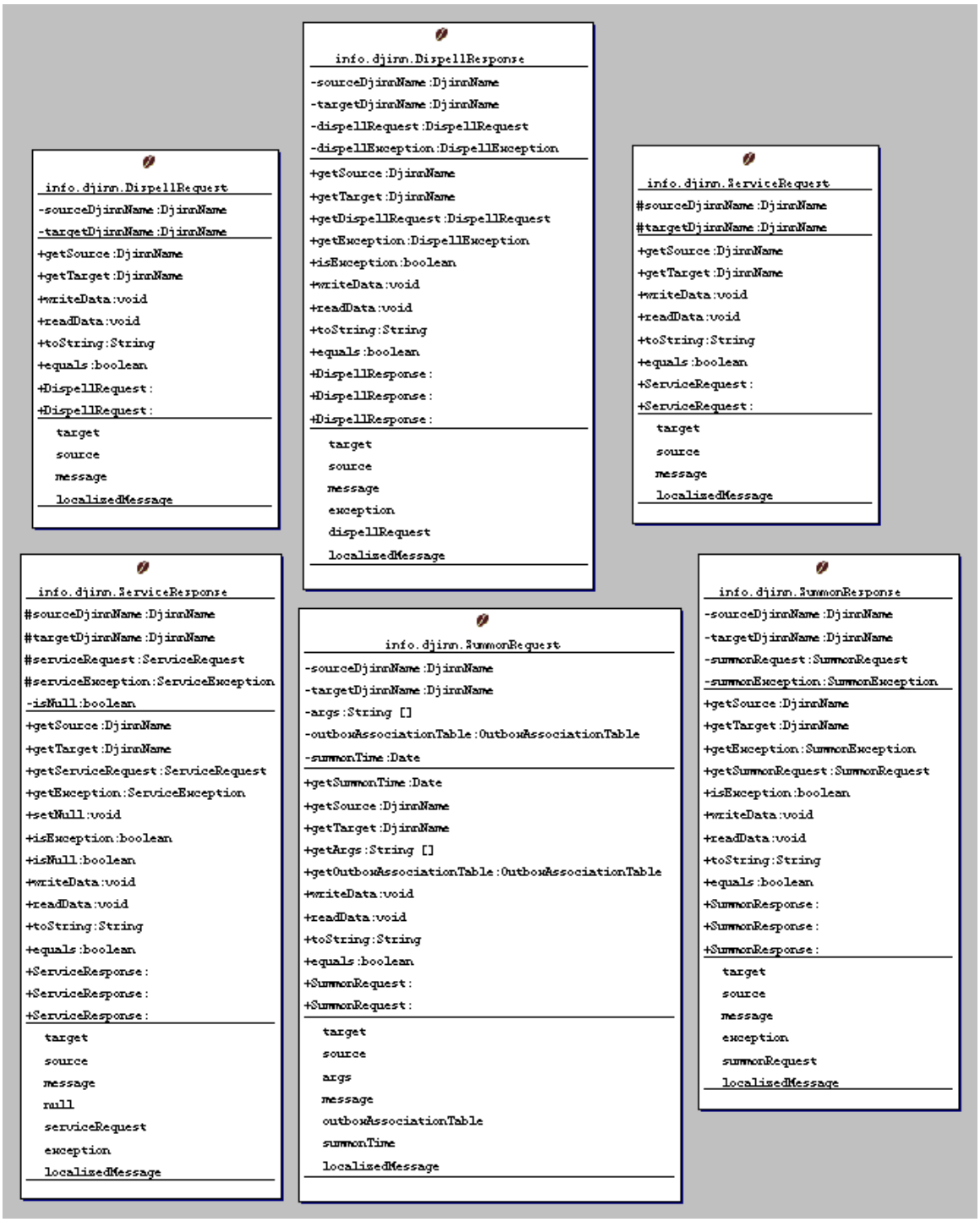


Figure 4.14: `info.djinn` class diagram (messages)

Figure 4.15: `info.djinn` class diagram (names)

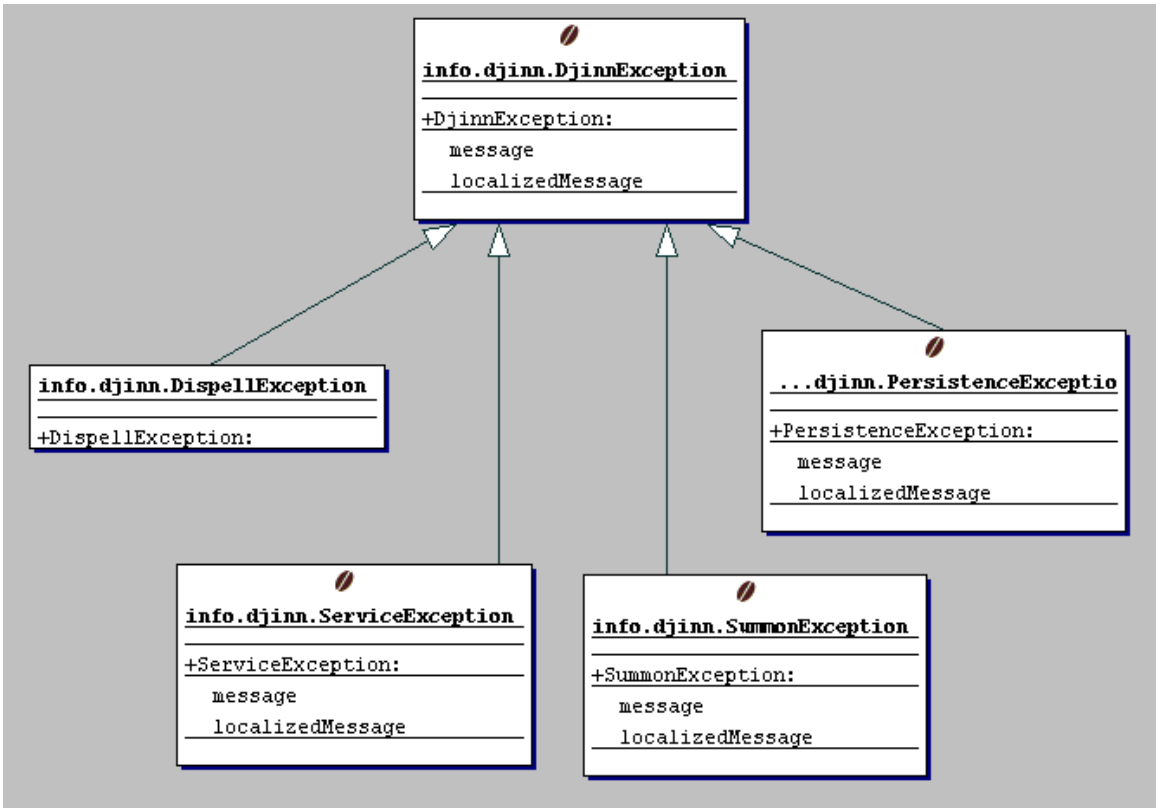


Figure 4.16: `info.djinn` class diagram (exceptions)

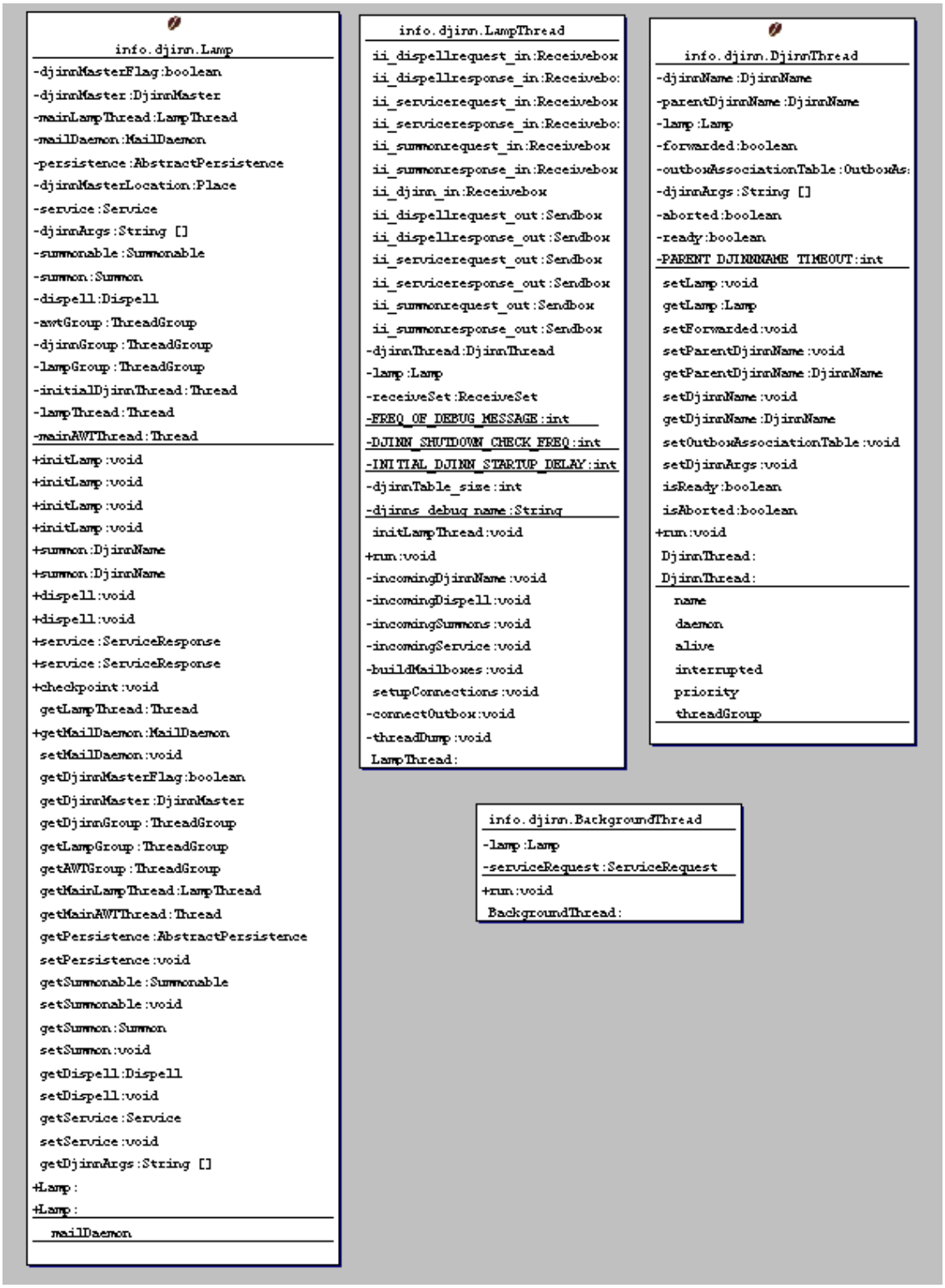


Figure 4.17: `info.djinn` class diagram (core)

Figure 4.18: `info.djinn` class diagram (UI and session management)

base class for the specific persistence mechanisms for the subclasses of *Djinn* that we do support. All base classes but *DjinnServlet* are either completed or under development.

Aside: Abstract Classes vs. Interfaces We chose to make *AbstractPersistence* an abstract base class because we wanted it to have non-public methods. This is a typical trade-off we repeatedly had to choose because of Java's restrictions of what types of features can go in interfaces. Almost all of the abstract base classes of the II do not contain base code, the primary reason in using an abstract class instead of an interface in almost all other object-oriented languages.

AppletPersistence and *Persistence* both specialize from *AbstractPersistence*. As their names would indicate, *AppletPersistence* provides persistence services for *DjinnApplets* and *Persistence* provides persistence for *Djinns* and *DjinnBeans* (and *DjinnServlets* with the proper security access).

Persistence. Each djinn instance is given a unique ID within the persistent file store. When a djinn is constructed and initialized for the first time, its corresponding *Persistence* object is constructed and initialized via the *initPersistence()* method. The *Persistence* object determines the djinn's unique ID at this time.

From that point onward, any time the djinn wishes to store its persistent state, it simply calls the *checkpoint()* method. *checkpoint* performs some safety checks on the data-store and, if appropriate, calls the djinn's *freeze()* method, piping the djinn's frozen state through an output stream and into the data-store.

When a djinn shuts down or is garbage collected, its *Persistence* object is finalized. Thus, the last snapshot of the djinn is complete and correct and the data-store is closed properly.

AppletPersistence. The protocol between *AppletPersistence* and *DjinnApplet* is identical to that of the standard *Persistence* object, as described above.

A *DjinnApplet* obtains its persistent store via the Web server from which its code was downloaded. The stream to which a *DjinnApplet's* state is sent is a `PUT` method on the originating Web server at the proper unique ID, as above. Likewise, obtaining persistent state for the *DjinnApplet* is accomplished via a `GET` method applied to the same server.

PersistenceException is the exception thrown by the persistence subsystem if there is an error during the initialization or action of the subsystem. In many cases a djinns can continue to operate if its data-store is temporarily unavailable, but such decisions are left up to the djinn's designer.

4.4.3.3 The Summoning/Dispelling Subsystems

A djinn A can cause a second djinn B to be instantiated via a remote Djinn Master M by *summoning* B. The *summonDjinn()* method of the *Summon* class is used to instantiate and obtain a reference to the summoned djinn. If the summoned djinn B is already running and in server mode (described previously), then the summon operation will simply return a new reference to B.

Likewise, when A is done interacting with B, A should *dispell* B. The semantics of the dispell operation are undefined — it is up to the designer of each djinn to decide what a dispell message means to that djinn. These semantics should be documented in the specification of the djinn. For A to dispell B, it should invoke the *dispellDjinn()* method of the *Dispell* class. Note that, in general,

only the *original* summoner of a djinn D has the right to permanently dispell D, but again, these semantics are up to D's designer.

4.4.3.4 The Service Subsystem

The *service* subsystem is responsible for handling both outbound and incoming service requests. If a djinn A has a reference to another djinn B, either via summoning B directly or obtaining the reference through a third party, it can invoke a service call upon B.

For A to make a service request, it invokes the *serviceDjinn()* method of the *Service* class. When the service request completes or fails, a result with be returned to the djinn or an exception will be raised.

When B receives the service request, its *service* method (on its base class) will be invoked by a *BackgroundThread*. This means that each djinn can permit or restrict as much concurrency as appropriate for its service and capability.

Note: service requests are synchronous on the sending side, but asynchronous on the receiving side (unless the djinn implementer decides otherwise).

4.4.3.5 The Session Management and User Interface Subsystems

The *session management* service provides the support for the *session* construct described in Section 4.3. When a djinn is summoned, a specification of its mailbox bindings can be provided via the *OutputAssociationTable* class. If such a specification is provided, then the infrastructure automatically performs the specified *binds()* on the djinn's mailboxes. Thus, this mechanism provides a simple means of constructing sessions. Our *djinn initiator*, described in Section 4.5.1 used this mechanism.

The *user interface* subsystem is responsible for determining if a djinn has access to a display device. It also displays a welcome dialog and copyright information. Such a display only takes place once in the lifetime of a Java virtual machine, thus copyright displays don't keep popping up all over your display as djinns are summoned and dispelled.

A djinn is not informed if it has access to a legal display device, it has to attempt to instantiate a frame and see if an exception is raised.

The reason for the existence of the *AWTThread* class is to work around several bugs and design flaws in the JDK 1.0 and 1.1 AWT systems. Essentially, neither provide any means of actually shutting down a virtual machines background AWT threads once they are started. Thus, to detect when a djinn goes quiescent requires some subtle manipulation of threads and threadgroups within the VM by *AWTThread*. We are hoping these issues are remedied in JDK 1.2.

This completes the summary of the `info.djinn` package.

As an example of the expressiveness of DESML, I will provide a partial specification of `info.djinn` in an object network diagram in Figure 4.19.

4.5 The Infospheres Infrastructure: Implementation

I will make a few notes about the implementation of the II. Please see our releases Web page at `http://www.infospheres.caltech.edu/releases/` for more information or to download the

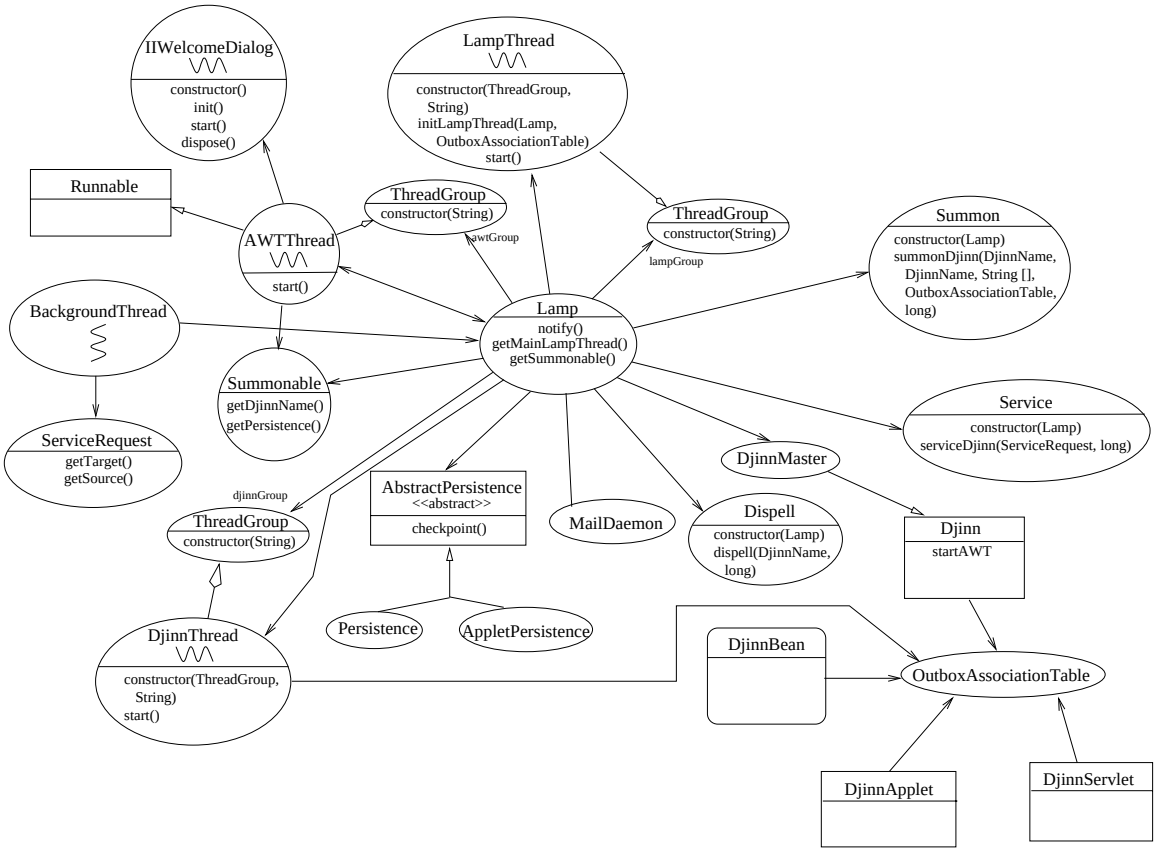


Figure 4.19: `info.djinn` object network diagram

framework.

4.5.1 Implementation Supplementary Components

The II comes with several supplementary tools and packages to help a user take advantage of the II and build their own djinns.

A tool called the *Djinn Initiator*, or *Initiator* for short, was included in early versions of the II. The initiator was a tool that supports the graphical composition of sessions.

In the Initiator, djinns are represented as icons, each of which had a set of assignable attributes. These attributes help the Initiator user describe which djinn each icon is meant to represent. Djinns are visually wired together, (or more precisely, their mailboxes are wired together), and the session can be instantiated with the push of a button. We had hoped to add a "djinn palette" to the initiator, but the project was not continued when we moved to adding more functionality to the core of the framework.

The *Djinn Summoner* is the primary tools used to summon arbitrary djinns and demonstrate the framework. The Summoner provides both a command-line and a graphical interface to summoning, requesting services from, and dispelling arbitrary djinns. A screen snapshot of the summoner can be seen is Figure 4.20.

Finally, we also supply a comprehensive debugging package with the II. `info.util.Debug` provides for the display and logging of several different classes and levels of debugging messages. Such a package was quite useful as we developed and debugged the II.

From the experience we gained in using this debug package, we designed and this author implemented a new, general-purpose distributed debugging package for Java which is being used extensively in the development of II 2.0 and Dan Zimmerman's ÜberNet [180].

4.5.2 Implementation Size

For such a comprehensive framework, the implementation of II 1.0 is surprisingly small. Table 4.1 shows the summary of the implementation size and comment ratios, as provided by the `CommentCounter`[2] tool written by this author.

If this same framework were to be reimplemented with Java 1.2, I estimate that we would see a 25% reduction in size. This difference is primarily due to the fact that we had to implement object persistence in II 1.0, whereas the use of JDK 1.2 would permit us to replace that code with the *RMI Serialization* mechanism. Additionally, the use of new collection classes in JDK 1.2 would reduce the code size by a few hundred lines.

4.5.3 Hindsight Design and Implementation Improvements

Our major design realization concerns indirect component coupling. We now recognize that the use of individual classes per message type, while a nice model, is not efficient or necessary. In fact, it is now our opinion that this was a poor design decision. This kind of message design indirectly couples the distributed components too tightly. If the implementation of a single message is changed, a potential trickle-down effect can impact all components that use that message.

[2] `CommentCounter` is one of a set of tools designed and built by this author that assist in the development, testing, and evaluation of Java software.

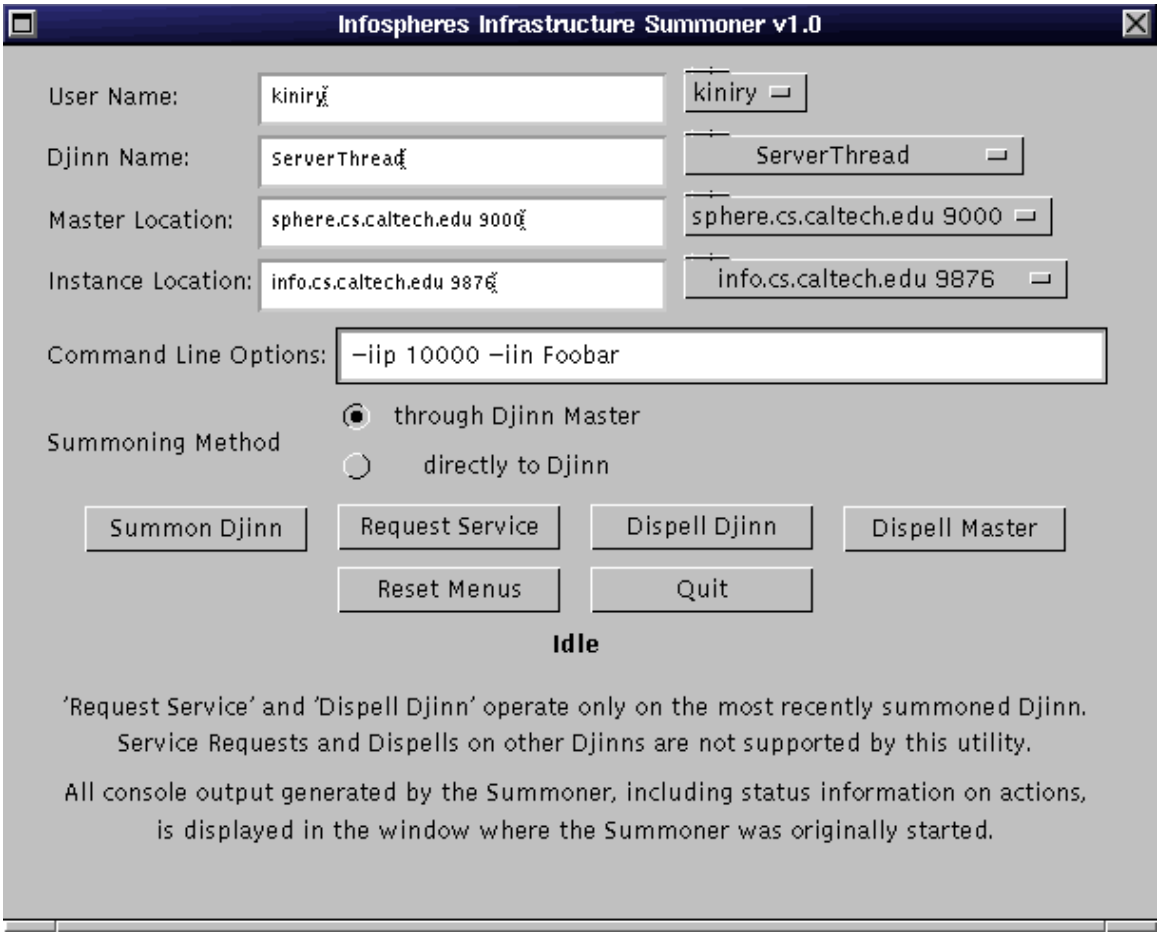


Figure 4.20: The Infospheres Infrastructure Summoner — GUI version

`info.net` package	
Total number of lines of code	4,351
Total number of lines of comments	2,140
Ratio (comments/code)	49%
`info.djinn` package	
Total number of lines of code	6,431
Total number of lines of comments	2,739
Ratio (comments/code)	43%
`info.master` package	
Total number of lines of code	2,301
Total number of lines of comments	793
Ratio (comments/code)	34%
Total number of lines of core code	13,083
Total number of lines of comments in core code	5,672
Total ratio (comments/code)	43%
`info.util` package	
Total number of lines of code	301
Total number of lines of comments	174
Ratio (comments/code)	58%
`info.demo` package	
Total number of lines of code	14,445
Total number of lines of comments	4,995
Ratio (comments/code)	35%
Grand Totals	
Grand total number of lines of supplied code	27,829
Grand total number of lines of supplied comments	10,841
Grand total ratio (comments/code)	39%

Table 4.1: A summary of the II 1.0 implementation size and internal documentation

To uncouple these dependencies, I suggest the use of a semantic messaging infrastructure, similar to that proposed by Sims in [150][3].

Such a change would not only decouple the communicating components, but would again reduce the code size by another few thousand lines.

4.5.4 Infrastructure Impact

Versions of the II have been downloaded by schools, companies, research labs, and others. To date (May, 1998), just over one thousand downloads of just the last four software releases (beta2, beta3, final candidate, and final release) have taken place. Collaborators have used the II at the University of Florida, Indiana University, and Trinity Collage for research and teaching.

Additionally, it has come to our attention that several companies have downloaded the package and incorporated ideas into their internal research and development work as well as product development. In particular, Sun Microsystems, ObjectSpace, Digital, Novell, IBM, Hewlett-Packard, and Microsoft have shown considerable interest.

4.5.5 Infrastructure Uses

Several distributed systems have been built with the II over the past sixteen months. I will briefly describe several of these in the next chapter. Also, I will propose an algorithm for archiving distributed states in a dynamic, component distributed system with the II.

[3] In fact, this author has such a package (called *Semantic Data Objects*) under development at this time.

Chapter 5

Examples

5.1 Distributed Systems Built with Infospheres

The II has been used to build several distributed systems. We will briefly mention a few of them here.

- *Autonomous poker-playing objects.* The II is being used in our *CS 141, Distributed Computation Laboratory* course here at Caltech. The class used the package extensively in November, 1996, writing autonomous poker-playing djinns that competed in a gaming tournament. This test pitted several dozen djinns against each other, interacting in up to five simultaneous sessions/games. Students' recommendations helped us make our packages substantially smaller, faster, more reliable, and more consistent than the previous releases. See the class's home page at `http://www.cs.caltech.edu/~cs141/` for more information.

- *Global resource reservation tool.* A global resource reservation tool was built that integrates a Tk/Tcl front-end with a Java/II back end to provide the user with an interactive interface for reserving slots on geographically distributed supercomputers with individual resource reservation schemes. See [138] for more information.

- *JEDI: A research framework for developing client-server systems.* The JEDI system lets a developer build client-server systems without having to go through the process of stub/skeleton compilation. See [6] for more information.

- *SimulEdit: A peer-to-peer distributed editor.* SimulEdit is an editor that lets users join and leave an editing session dynamically and is fault tolerant. See [164] for more information.

- *DALI: A distributed artificial life simulator infrastructure.* This system lets the user construct an distributed asynchronous simulation system for simulating artificial life systems (genomes specify behavior, speciation, pack behavior, etc.). The system scales extremely well given Infospheres 1.0's properties (i.e. hundreds of thousands to millions of interacting agents are possible). See [104] for more information.

- *Virtual Swap Meet: A distributed agent marketplace.* This project is a peer-to-peer autonomous agent auctioning system. This system lets the user specify a set of items that they are interested in purchasing and a set of items that they are interested in selling. The example application

uses compact discs or used books. The user also specifies how much they are willing to pay for/get for the items and what sort of strategies to employ in buying or selling the items (first hit, highest bid after a time window, etc.) The system then autonomously posts the selling information to a variety of communication venues (organized NetNews groups, Web servers, and multicast addresses are in the current design) and spawns agents to autonomously search through these same sources for items of interest. Once a possible match is found, bartering agents then attempt to make the purchase/sale according to the user's preferences. See the group's project summary for more information [120].

- *Distributed Games.* A clone of the game *Diplomacy* was developed using II but could not be released due to licensing restrictions imposed by the owner company. Additionally, several entertaining client/server games (the card game "Spit", TRON light cycles, and the classic Pong) were constructed with the `info.net` package. See [181] for more information.

Further example system implementation will take place on top of II 2.0, expected to be available in third quarter of 1998. My remaining distributed system's implementation work (to complete the Ph.D.) will all use II 2.0.

5.2 Archiving Distributed States

We have now described our prototype software infrastructure; next, we describe an algorithm that can be used by the infrastructure to archive distributed states. This is a variant of the global snapshot algorithm [29] in which a clock, or sequence number, is stored with the snapshot state. Within the snapshots, these logical clocks can be used for timestamping [107]. Note that this algorithm has not been implemented, this is just a specification of the algorithm.

5.2.1 The Global Snapshot Algorithm

If all components recorded their complete states (including the states of their mailboxes) at a specified time T, then the collection of component states would be the state of the distributed system at time T. The problem is that the clocks of the components can drift and, as illustrated by the following example, even a small drift can cause problems.

Two components P and Q share an indivisible token that they pass back and forth between them. P's clock is slightly faster than Q's clock. Both processes record their states when their clocks reach a predetermined time T. Assume that the token is at Q when P's clock reaches T, so so P's recorded state shows that P does not have the token. Then, after Q has sent the token to P, Q's clock reaches time T. Q's recorded state then shows that Q does not have the token. Therefore, the recorded system state — the combined recorded states of P and Q that shows that no token is anywhere in the system — is erroneous. The basic problem arises because Q sends a message to P after P records its state but before Q records its state.

We describe our algorithm in terms of taking a single global snapshot. In practice, we will need to take a sequence of global snapshots, and extending the single snapshot algorithm to take sequences of snapshots is straightforward.

Initially, some component records its state; the mechanism that triggers this initial recording is irrelevant. Perhaps a component records its state when its local clock gets to some predetermined time T, and the component with the clock that reaches T first is the first to record its state.

Each message sent by a component is tagged with a single boolean which identifies the message as being either (i) sent before the component recorded its local state, or (ii) sent after the component recorded its local state. In our infrastructure, every message is acknowledged, so each acknowledgment is also tagged with a boolean indicating whether the acknowledgment was sent before or after the component recorded its state. When a message tagged as being sent after the sender recorded its state arrives at a receiver that has not recorded its state, the infrastructure causes the receiver's state to be recorded before delivering the message. Acknowledgments are also tagged, and are handled in the same way. Thus, the algorithm maintains the invariant that a message or acknowledgment sent after a component records its state is only delivered to components that have also recorded their states.

The issue of acknowledgments is somewhat subtle, so I will describe it in more detail. Consider a component P sending a message m to a component Q. The message m is at the head of an outbox of P. The message-passing layer sends a copy of m to Q's inbox, to which that outbox is connected. Note that m remains in the outbox while the copy of m is in transit to Q's inbox. When the acknowledgment for m arrives at P, then and only then is message m discarded from P's outbox. If the acknowledgment is a post-recording acknowledgment, then P's state is recorded before the acknowledgment is delivered, and therefore P's state is recorded as still having message m in its outbox.

5.2.2 Repeated Snapshots

The algorithm for taking a single snapshot of an entire distributed system requires each component to have a boolean indicating whether that component has recorded its state. Also, each message and acknowledgment has a boolean field indicating whether that message or acknowledgment was sent before or after the sender of that message or acknowledgment had recorded its state. For repeated snapshots, the boolean is replaced by a date represented by a sequence of integers for year, month, day, time in hours, minutes, seconds, milliseconds, and so on, to the appropriate granularity level. The date field of a component indicates when the component last recorded its state, and this date field is copied into messages and acknowledgments sent by the component. If a component receives a message or acknowledgment with a date that is later than its current date field, it takes a local snapshot, updates its date field to the date of the incoming message, and (if necessary) moves its clock forward to exceed the date of the incoming message.

5.2.3 Replaying a Distributed Computation

There is a distinction between having the saved state of a distributed computation and being able to replay the computation. An archived snapshot helps in a variety of ways but, because some distributed computations are nondeterministic, it does not guarantee that the distributed computation can be replayed.

Our components are black boxes, so we cannot tell whether a component is deterministic. Reexecuting the computation of a nondeterministic component from a saved state can result in a

different computation, even though the component receives a sequence of messages identical to the sequence it received in the original computation. Replaying precisely the same sequence of events requires each component to execute events in exactly the same order as in the original sequence, so the replay has to be deterministic. For example, if there is a race condition in the original computation, then the replay must ensure that the race condition is won by the same event as in the original. Since components are black boxes, the II cannot control events within a component. Therefore, we rely on the designers of the components to have a record-replay mechanism for recording the event that occurs in each nondeterministic situation and playing back this event correctly during replay.

During replay, the II ensures that messages are delivered to a component in the same order as in the original computation, provided all components in the computation send the same sequences of messages. If the components have deterministic replay, the computation from the saved state will be an exact replay: a sequence of events identical to those of the original computation.

The II guarantees that messages are delivered in the same order as in the original computation in the following way: a maildaemon executes on each computer that hosts components, logging the outbox, inbox and message id for each incoming message. Because the contents of the messages are not necessary to properly deal with nondeterminism in the message-passing layer, they are not recorded by the maildaemon. During replay, the maildaemon holds messages that arrive in a different order, delivering them to the appropriate inboxes only after all previous messages in the original computation have been delivered.

5.2.4 A World Wide Web of Distributed Spaces

The existing Infospheres Infrastructure supports saving the states of components and summoning components from these archived states to form new sessions. When a component is summoned from an archived state, it resumes computation from that state. It is convenient to treat each archived component as being unique; for instance, there may be a solid-mechanics computation component that is persistent (and, for practical purposes, lives forever), but an experimenter may have a sequence of related components corresponding to states of that component used at different times in different experiments. Our intent is to provide access to these archived components through a Web browser, using the standard summoning mechanism.

Chapter 6

Conclusion

I have described some of the problems inherent in the specification of complex systems, both at the macro (system) and micro (component) level. Additionally, I have provided several contructs that begin to solve these system specification problems at the macro level.

6.1 Contributions

DESML is an evolving layer for system modeling languages. Note that I have only described a few constructs that begin to solve the dynamism and component-related problems specified in Section 2.5.

As defined here, DESML is a variant of UML, not an extension. I have redefined the its meta-model, thus the new language is no longer compatible at the meta-level with UML.

6.2 Future Work

The problems that DESML set out to solve have been specified by this author for several years. But, during that interval, in its attempt to modeling emergent systems, metalevels and knowledge, I believe that OOCL has begun to handle the balance of the problems specified in Section 2.5.

System modeling is no longer in our core research agenda. Once the full OOCL method becomes publicly available I will evaluate its constructs and incorporate appropriate elements into DESML. Or, put more appropriately, OOCL might become the new base on which to put the thin-layer that is DESML.

Knowledge representation and component specification are still in our research agenda. The problem of tying knowledge to specification, especially at the metalevel, are topics to be addressed in the next phase of our research program.

Bibliography

[1] Martin Adabi and Luca Cardelli. *A Theory of Objects*. Springer–Verlag, 1996.

[2] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko. The self 3.0 programmer's reference manual. Technical report, Sun Microsystems, 1993.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[4] Gul Agha and Carl Hewitt. *Research Directions in Object-Oriented Programming*, chapter Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming, pages 49–74. The MIT Press, 1987.

[5] Agilis corporation web site. Available as `http://www.agiliscorp.com/`, 1998.

[6] Jonathan Aldrich, James Dooley, Scott Mandelsohn, and Adam Rifkin. Providing easier access to remote objects in distributed systems. In *Hawaii International Conference on System Sciences*. IEEE Computer Society, January 1998.

[7] Peter Andrews. *A Transfinite Type Theory with Type Variables*. North-Holland Publishing Company, 1965.

[8] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

[9] Apple Computer Eastern Research and Technology Center. Dylan — an object-oriented dynamic language. Technical report, Apple Computer, Cambridge, MA, 1992.

[10] Ken Arnold. *The Java Programming Language, Second Edition*. Addison-Wesley Publishing Company, 1998.

[11] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. Springer–Verlag, 1994.

[12] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA begin*. Studentlitteratur, 1979.

[13] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor. Cogito: A methodology and system for formal software development. *International Journal of Software Engineering and Knowledge Engineering*, 5(4):599–617, December 1995.

[14] Daniel G. Bobrow, Linda G. Demichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification.* X3J13 Committee, document 880992r edition, June 1988.

[15] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley Publishing Company, 1994.

[16] Grady Booch. *Object Solutions — Managing the Object-Oriented Product.* Addison-Wesley Publishing Company, 1996.

[17] Grady Booch, Jim Rumbaugh, and Ivan Jacobson. *Unified Modeling Language User Guide.* Addison-Wesley Publishing Company, 1997.

[18] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. Technical report, World Wide Web Consortium, 1998. `http://www.w3.org/TR/WD-xml-names`.

[19] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML). Technical report, World Wide Web Consortium, 1998. `http://www.w3.org/TR/REC-xml`.

[20] Edmund Burke and Eric Foxley. *Logic and its Applications.* Prentice-Hall, Inc., 1996.

[21] Rich Burridge. *Java Shared Data Toolkit User Guide.* Sun Microsystems, Inc., 1.2 edition, April 1998.

[22] P. Butcher. A behavioral semantics for Linda-2. *Software Engineering Journal*, 6(4):196–204, July 1991.

[23] Luca Cardelli. Obliq — a language with distributed scope. Technical report, DEC Systems Research Center, 1994.

[24] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[25] CCITT. Specification and Description Language (SDL), recommendation z.100. Technical report, CCITT, 1988.

[26] K. Mani Chandy, Joseph R. Kiniry, Adam Rifkin, and Daniel M. Zimmerman. Webs of archived distributed computations for asynchronous collaboration. *Journal of Supercomputing*, 11, 1997.

[27] K. Mani Chandy, Joseph R. Kiniry, Adam Rifkin, Daniel M. Zimmerman, Wesley Tanaka, and Luke Weisman. A framework for structured distributed object computing. California Institute of Technology Technical Report Caltech-CS-TR-97-07, California Institute of Technology, February 1997.

[28] K. Mani Chandy, Joseph R. Kiniry, Adam Rifkin, Daniel M. Zimmerman, Wesley Tanaka, and Luke Weisman. A framework for structured distributed object computing. crpc-tr CRPC-97-2, California Institute of Technology, February 1997.

[29] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining the global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

[30] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.

[31] K. Mani Chandy and Adam Rifkin. Systematic composition of objects in distributed Internet applications: Processes and sessions. In *Hawaii International Conference on System Sciences*, pages 63–75. IEEE Computer Society, January 1997.

[32] K. Mani Chandy, Adam Rifkin, and Eve Schooler. Using announce-listen with global events to develop distributed control systems. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998.

[33] K. Mani Chandy, Adam Rifkin, Paolo Sivilotti, Jacob Mandelson, Matt Richardson, Wesley Tanaka, and Luke Weisman. A worldwide distributed system using Java and the Internet. In *International Symposium on High Performance Distributed Computing*, pages 11–18, 1996.

[34] Conceptual Knowledge Markup Language (CKML) DTD. `http://asimov.eecs.wsu.edu/WAVE/Ontologies/CKML/CKML-DTD.html`.

[35] Peter Coad and Edward Yourdon. *OOA: Object-Oriented Analysis, 2nd Edition*. Prentice-Hall, Inc., 1990.

[36] Peter Coad and Edward Yourdon. *OOD: Object-Oriented Design*. Prentice-Hall, Inc., 1990.

[37] Derek Coleman et al. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, Inc., 1993.

[38] COMP.LANG.ML Frequently Asked Questions and Answers. `http://www.cis.ohio-state.edu/hypertext/faq/usenet/meta-lang-faq/faq.ht%ml`.

[39] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice-Hall, Inc., 1994.

[40] Rational Software Corporation et al. *UML Notation Guide, version 1.1*. The UML 1.1 Consortium, September 1997.

[41] Rational Software Corporation et al. *UML Semantics, version 1.1*. The UML 1.1 Consortium, September 1997.

[42] Rational Software Corporation et al. UML Summary, version 1.1. Technical report, The UML 1.1 Consortium, September 1997.

[43] Brad Cox. *Superdistribution: Objects As Property on the Electronic Frontier*. Addison-Wesley Publishing Company, 1996.

[44] Brad Cox and Andrew Novabilsky. *Object-Oriented Programming, An Evolutionary Approach*. Addison-Wesley Publishing Company, 1987.

[45] O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[46] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer–Verlag, 1990.

[47] Desmond D'Souza and Alan Wills. *Objects, Components, and Frameworks with UML: the Catalysis Approach.* Addison-Wesley Publishing Company, 1998.

[48] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. Technical Report 94-45, Software Verification Research Centre, School of Information Technology, The University of Queensland, December 1994.

[49] E.H. Dürr and J. van Katwijk. VDM++ — a formal specification language for object-oriented designs. In Bertrand Meyer, Georg Heeg, and Boris Magnusson, editors, *Proceedings of Technology of Object-oriented Languages and Systems (TOOLS Europe)*, pages 63–78. Prentice-Hall, Inc., 1992.

[50] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit.* John Wiley & Sons, Inc., 1998.

[51] David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. Lclint: A tool for using specifications to check code. In *Proceedings of the Symposium on the Foundations of Software Engineering*, December 1994.

[52] G. Florijn. Object protocols as functional parsers. In *Proceedings of the European Conference on Object-Oriented Programming*, number 952 in Lecture Notes in Computer Science, pages 351–373, 1995.

[53] USGS NSDI formal metadata information and tools. `http://geochange.er.usgs.gov/pub/tools/metadata/`.

[54] Jay W. Forrester. *Industrial Dynamics.* Productivity Press, 1961.

[55] Jay W. Forrester. *Urban Dynamics.* Productivity Press, 1969.

[56] Jay W. Forrester. *World Dynamics.* Productivity Press, 2nd edition, 1971.

[57] Jay W. Forrester. *Collected Papers of Jay W. Forrester.* Productivity Press, 1975.

[58] Svend Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* The MIT Press, 1996.

[59] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects.* Oxford University Press, 1994.

[60] R.P. Gabriel, J.L. White, and D.G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34:942–960, 1991.

[61] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[62] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation Systems Research Center, 1991.

[63] Mauro Gaspari and Gianluigi Zavattaro. An algebra of actors. Technical Report UBLCS-97-4, Department of Computer Science, University of Bologna, May 1997.

[64] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989.

[65] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[66] Adele Goldberg and K.S. Rubin. *Smalltalk-80: The Language edition = Revised, publisher = pub-aw, year = 1995,*.

[67] Martin Goldstern and Haim Judah. *The Incompleteness Phenomenon: A New Course in Mathematical Logic*. A.K. Peters, 1995.

[68] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, 1996.

[69] The Infospheres Group. *The Infospheres Infrastructure version 1.0 User's Guide*. The Infospheres Group, California Institute of Technology, Aug 1996.

[70] The Infospheres Group. *The Infospheres Infrastructure version 2.0 Tutorial*. The Infospheres Group, California Institute of Technology, 1998.

[71] The Infospheres Group. *The Infospheres Infrastructure version 2.0 User's Guide*. The Infospheres Group, California Institute of Technology, 1998.

[72] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. Foundations of Computing. The MIT Press, Cambridge, MA, USA, 1993.

[73] John Guttag, James J. Horning, et al., editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer–Verlag, 1993.

[74] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Software Engineering Conference*, pages 246–257. IEEE Press, March 1996.

[75] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. In *ACM Transactions on Software Engineering and Methodology*, volume 5/4, October 1996.

[76] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[77] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices, pages 411–428. ACM SIGPLAN, ACM Press and Addison-Wesley, 1993.

[78] Richard Hayton, Jean Bacon, John Bates, and Ken Moody. Using events to build large scale distributed applications. In *SIGOPS European Workshop '96*, 1996.

[79] Matthew Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.

[80] C. Hewitt, P. Bishop, and R. Steiger. A universal modelar ACTOR formalism for AI. In *Proceedings, Third International Joint Conference on Artificial Intelligence*, 1973.

[81] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[82] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., 1985.

[83] John H. Holland. *Hidden Order: How Adaptation Builds Complexity.* Addison-Wesley Publishing Company, 1995.

[84] John J. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, 1975.

[85] Walter L. Hürsch. User's guide to the Demeter Tools/C++. C++ Demeter System Documentation, May 1991.

[86] A. Hutt. *Analysis and Design: Comparison of Methods.* John Wiley & Sons, Inc., 1994.

[87] A. Hutt. *Analysis and Design: Description of Methods.* John Wiley & Sons, Inc., 1994.

[88] IBM et al. *Object Constraint Language Specification, version 1.1.* The UML 1.1 Consortium, September 1997.

[89] IFAD. IFAD VDM tools. `http://www.ifad.dk/products/vdmtools.html`.

[90] Darrel C. Ince. *An Introduction to Discrete Mathematics, Formal System Specification and Z.* Oxford University Press, 1992.

[91] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications.* Apple Computer and Walt Disney Imagineering, 1997. `ftp://st.cs.uiuc.edu/Smalltalk/Squeak/docs/OOPSLA.Squeak.html`.

[92] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Process: A Software Engineering Process Using the Unified Modeling Language.* Addison-Wesley Publishing Company, 1998.

[93] Ivar Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering.* Addison-Wesley Publishing Company, 1992.

[94] Ivar Jacobson et al. *Object-Oriented Software Engineering — A Use Case Driven Approach.* ACM Press/Addison-Wesley, 1992.

[95] Prashant Jain and Douglas Schmidt. Java ACE homepage. `http://www.cs.wustl.edu/~schmidt/JACE.html`.

[96] Ralph Johnson. Personal email communication. dist-obj Mailing List: `http://www.infospheres.caltech.edu/mailing_lists/dist-obj/msg01196.html%`, May 1998.

[97] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, Inc., 1986.

[98] Markus Kaltenbach. Model checking for UNITY. Technical report, Department of Computer Sciences, The University of Texas at Austin, 1994.

[99] Stewart Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution.* Oxford University Press, 1993.

[100] Stewart Kauffman. *At Home in the Universe: The Search for Laws of Self-Organization and Complexity.* Oxford University Press, 1995.

[101] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley Publishing Company, 1989.

[102] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[103] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, XEROX Corporation, February 1997.

[104] Joseph R. Kiniry, Alexander Nicolson, and Donald Pinkston. DALI: A distributed artificial life simulator infrastructure. WWW, 1997. Available at `http://www.cs.caltech.edu/~kiniry/projects/alife/index.html`.

[105] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A general purpose event-action system. *ACM Transactions on Software Engineering and Methodology*, 21(10):845–857, October 1995.

[106] Bent B. Kristensen, Ole L. Madsen, Birger Møller-Pederson, and Kristen Nygaard. *The BETA Programming Language*, pages 7–48. The MIT Press, 1987.

[107] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[108] Don Larkin and Greg Wilson. Object-oriented programming and the Objective-C language. Technical report, NeXT Software, Inc. (now Apple Corp.), 1993.

[109] Ora Lassila and Ralph Swick. Resource Description Framework (RDF) Model and Syntax. Technical report, World Wide Web Consortium, 1998. `http://www.w3.org/TR/WD-rdf-syntax`.

[110] Doug Lea. Design for open systems in Java. In *Proceedings of the Second International Conference on Coordination Models and Languages*, Berlin, Germany, September 1997.

[111] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[112] Mark Lutz. *Programming Python.* O'Reilly & Associates, Inc., 1996.

[113] Ole L. Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley Publishing Company, 1993.

[114] Pattie Maes and Daniele Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[115] Silvano Maffeis. iBus — the Java intranet software bus. Technical report, SoftWired AG, Switzerland, 1997.

[116] Eve Maler and Steve DeRose. XML Linking Language (XLink). Technical report, World Wide Web Consortium, 1998. `http://www.w3.org/TR/WD-xlink`.

[117] Eve Maler and Steve DeRose. XML Pointer Language (XPointer). Technical report, World Wide Web Consortium, 1998. `http://www.w3.org/TR/WD-xptr`.

[118] James Martin and James J. Odell. *Object-Oriented Methods Pragmatic Considerations*. Prentice-Hall, Inc., 1996.

[119] James Martin and James J. Odell. *Object-Oriented Methods, A Foundation*. Prentice-Hall, Inc., 1997.

[120] Elise Matefy and Jessica Walton. Virtual swap meet: A distributed agent marketplace, August 1997. `http://www.infospheres.caltech.edu/papers/vsm/paper.html`.

[121] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 2nd edition, 1988.

[122] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

[123] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[124] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[125] Jayadev Misra. *A Discipline of Multiprogramming*. Published via the web, 1996. Available as `ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z`.

[126] P. Naur and B. Randell, editors. *Proceedings, NATO Conference on Software Engineering*, Brussels, Belgium, October 1969. NATO Science Committee. Published as a book in 1976.

[127] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Inc., 1991.

[128] Inc. Oberon microsystems. Component pascal language report. Technical report, Oberon microsystems, Inc., September 1997. Available at `http://www.oberon.ch/docu/language_report.html`.

[129] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification (CORBA), revision 2.0*. Object Management Group (OMG), 2.0 edition.

[130] Ontology Markup Language (OML) DTD. `http://asimov.eecs.wsu.edu/WAVE/Ontologies/OML/OML-DTD.html`.

[131] Andrew Paepcke, editor. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 1993.

[132] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 1995.

[133] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer–Verlag, 1994.

[134] Lawrence C. Paulson. *ML for the Working Programmer (2nd Edition)*. Cambridge University Press, 1996.

[135] L.L. Peterson and B.S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 1996.

[136] L.J. Pinson and R.S. Wiener. *Objective-C: Object-Oriented Programming Techniques*. Addison-Wesley Publishing Company, 1991.

[137] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice-Hall, Inc., 1991.

[138] Ravi Ramamoorthi, Adam Rifkin, Boris Dimitrov, and K. Mani Chandy. A general resource reservation framework for scientific computing. In *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, December 1997.

[139] Rational Software Corporation. Rational ROSE. `http://www.rational.com/products/rose/`.

[140] Tom S. Ray. An evolutionary approach to synthetic biology: Zen and the art of creating life. *Artificial Life*, 1(1/2):195–226, 1994.

[141] M. Reiser and Niklaus Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula*. Addison-Wesley Publishing Company, 1992.

[142] James Rumbaugh and Stephen Blaha. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[143] Jim Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, Bill Lorensen, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[144] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley Publishing Company, 1997.

[145] Douglas C. Schmidt and Steve Vinoski. OMG event object service. *SIGS*, 9(2), February 1997.

[146] Peter Senge. *The Fifth Discipline, The Art and Practice of Learning Organization*. Doubleday, 1990.

[147] Peter Senge et al. *The Fifth Discipline Fieldbook*. Doubleday, 1994.

[148] Sally Shlaer and Steven Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice-Hall, Inc., 1988.

[149] Sally Shlaer and Steven Mellor. *Object Life Cycles: Modeling the World in State*. Prentice-Hall, Inc., 1991.

[150] Oliver Sims. *Business Objects: Delivering Cooperative Objects for Client-Server*. McGraw-Hill, 1994.

[151] Brian C. Smith. Reflection and semantics in LISP. In *ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1984.

[152] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[153] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1992.

[154] G. Starovic, V. Cahill, and B. Tangney. An event-based object model for distributed programming. Technical report, Department of Computer Science, University of Dublin, Trinity College, December 1995.

[155] Guy Steele. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[156] W.R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley Publishing Company, 1994.

[157] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Birkhduser, 1991.

[158] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*. Sun Microsystems, Inc., 1.2beta1 edition, October 1997.

[159] Sun Microsystems, Inc. JavaBeans API specification, version 1.01. Technical report, Sun Microsystems, Inc., July 1997.

[160] Edward Swanstrom. *OOCL Workbook*. Agilis Corporation, 1997.

[161] Edward Swanstrom. *Creating Agile Organizations with the OOCL Method*. John Wiley & Sons, Inc., 1998. See `http://www.agiliscorp.com/ooclforum.html` and `http://www.agiliscorp.com/oocldraft/`.

[162] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, 1997.

[163] PLATINUM technology, Veronica Bowman, Edward Swanstrom, et al. *Paradigm Plus: Methods Manual Addendum (OOCL)*. PLATINUM technology, 3.52 edition, 1997.

[164] Louis Thomas, Sean Suchter, and Adam Rifkin. Developing peer-to-peer applications on the Internet: The distributed editor, SimulEdit. *Dr. Dobb's Journal*, 1997.

[165] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley Publishing Company, 1991.

[166] Kenneth J. Turner, editor. *Using Formal Description Techniques: An Introduction to Estelle, LOTOS, and SDL.* Wiley Series in Communication and Distributed Systems. John Wiley & Sons, Inc., 1993.

[167] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3):223–242, 1991.

[168] D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, 1991.

[169] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS.* Elsevier Science, Inc., 1989.

[170] Guido van Rossum. Python reference manual. Technical Report 1.5, Corporation for National Research Initiatives, December 1997. Available at `http://www.python.org/doc/ref/`.

[171] Aaron Watters, Guido van Rossum, and James C. Ahlstrom. *Internet Programming with Python.* IDG Books, 1996.

[172] Peter Wegner. Interactive foundations of computing. `http://www.cs.brown.edu/people/pw/papers/finaltcs.ps`, 1997.

[173] R.S. Wiener. Programming with object-pascal, pascal 5.5 from borland international. *SIGSnet Journal of Object-Oriented Programming*, 2(2):62–65, July/August 1989.

[174] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice-Hall, Inc., 1990.

[175] Niklaus Wirth. The programming language PASCAL. *Acta Informatica*, 1:35–63, 1971.

[176] Niklaus Wirth. *Programming in Modula-2.* Springer–Verlag, 1982.

[177] Niklaus Wirth and J. Gutknecht. *Project Oberon — The Design of an Operating System and Compiler.* Addison-Wesley Publishing Company, 1992.

[178] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall International Series in Computer Science. Prentice-Hall, Inc., 1996.

[179] J.H. Wright. A reference model for object-oriented distributed systems. *British Telecom Technology Journal*, 6(3):66–75, 1988.

[180] Daniel M. Zimmerman. A preliminary investigation into dynamic distributed workflow. Technical Report CS-TR-98-09, Department of Computer Science, California Institute of Technology, 1998.

[181] Daniel M. Zimmerman, Brian Rothstein, Yevgeniy Kaganovich, and Khai Pham. Constructing client-server multi-player asynchronous games using a single-computer model. In *Proceedings of IASTED, International Conference on Software Engineering*, November 1997.