Adaptive Caching of Distributed Components

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur
(Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Christoph Pohl
geboren am 5. April 1977 in Pirna

Gutachter:

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill	TU Dresden
Prof. Dr.-Ing. Wolfgang Lehner	TU Dresden
Prof. Dr. rer. nat. habil. Heinrich Hußmann	LMU München

Tag der Verteidigung: 12. Mai 2005

Dresden im Mai 2005

Preface

Locality of reference is an important property of distributed applications. *Caching* is typically employed during the development of such applications to exploit this property by locally storing queried data: Subsequent accesses can be accelerated by serving their results immediately form the local store.

Current middleware architectures however hardly support this non-functional aspect. The thesis at hand thus tries outsource caching as a separate, configurable middleware service. Integration into the software development lifecycle provides for early capturing, modeling, and later reuse of caching-related metadata. At runtime, the implemented system can adapt to caching access characteristics with respect to data cacheability properties, thus healing misconfigurations and optimizing itself to an appropriate configuration. Speculative prefetching of data probably queried in the immediate future complements the presented approach.

How to read this book

The introduction in section 1 on page 1 outlines the motivation behind this work, specifies the addressed problems, lists possible application scenarios where these problems occur, states the objectives in terms of goals, non-goals, and eventually the claims of this thesis. Finally, a number of requirements are derived and the fundamental building blocks of the contributions of this thesis are explained.

This thesis crosscuts a large area of applied computer science, including distributed systems, middleware, and database technology, software engineering, and adaptive systems. Section 2 on page 9 gives an overview of the state-of-the-art of the relevant subset of these areas.

Consequentially, section 3 on page 81 introduces the related work, i.e., approaches, concepts, and other research work that parallels part of the functionality presented in this thesis.

As the first chapter describing the actual contribution of this thesis, section 4 on page 107 starts the concept of static caching, i.e., preconfiguring cacheability properties before deployment. However, cacheability metadata may be initially misconfigured and it is subject to changing access characteristics. Hence, adaptive countermeasures augment the implemented middleware service with self-healing and self-optimization properties. A heuristic for im-

proving weak consistency is presented as well as extensions for speculative prefetching.

The integration of modeling aspects for cacheability metadata into the software development cycle is discussed in section 5 on page 131. A UML Profile for caching comprises light-weight UML extensions, which can be used to model such properties and reuse them for model-driven generation of code fragments.

For the discussion of implementation issues, section 6 on page 147 explains two different approaches how the conceived middleware services for caching can be integrated in exisisting component-oriented middleware platforms. The second approach, based on invocation interception, is finally chosen as a basis for the prototype.

Finally, section 7 on page 161 concludes the presented contributions and reasons about potential benefits as well as the issues of a proper evaluation. An outlook points to directions of future research.

Typographic Conventions

Text typeset in *italics (Palatino)* is used for new *keywords/concepts*, *foreign language terms*, and *formulas*. The `typewriter (Courier)` font is used for `source code` and `WWWlinks`. And sans-serif (AvantGarde) for identifiers in the Unified Modeling Language (UML).

Acknowledgements

First of all, I would like to thank my advisor Prof. Dr. Alexander Schill for supporting my researching and for giving me the opportunity of conducting this work in the creative atmosphere of his chair for computer networks. Special thanks also go to my second and third reviewers, Prof. Dr. Wolfgang Lehner, and Prof. Dr. Heinrich Hußmann, who both readily agreed to support my thesis in an uncomplicated manner. I already learned to value both of them as competent contact persons and pleasant discussion partners in the COMQUAD project; Prof. Hußmann also during the organization of the ≪UML≫ 2004 workshop on Models for Non-functional Aspects of Component-Based Software (*NfC'04*).

During the compilation of this thesis, I have received material, insights and general hints from the following persons during countless discussions: Prof. Dr. Alexander Schill, Prof. Dr. Wolfgang Lehner, Prof. Dr. Heinrich Hußmann, Prof. Dr. Klaus Meyer-Wegener, Dr. Olaf Neumann, Dr. Thomas Springer, Dr. Thomas Ziegert, Hamud Al Hammoud, Sven Buchholz, Steffen Göbel, Marcus Meyerhöfer, Daniel Pfeifer, Simone Röttger, Daniel Schaller, Stefan Urbansky, Steffen Zschaler, and the many anonymous reviewers of my publications. I wish to thank all of them for their time and patience.

Furthermore, my friends and fellow climbers deserve a warm thank you and “Berg Heil” for the many beautiful, relaxing days in the rocks of Elbsandstein and elsewhere, for sharpening my sense for responsibility, and for teaching me how to control my fears. The pleasant hours in their company helped me to regain the strength needed for completing this work.

Finally, special thanks go to my parents and Noreen for love, support, sympathy, and understanding, especially during the last busy months.

Dresden, May 2005 Christoph Pohl

Contents

List of Figures

List of Tables

Listings

The ancient Masters did not try to educate the people but kindly taught them to not-know.

Tao te Ching by Lao Tsu (∗570–†490 BC), Chinese philosopher.

1 Introduction

1.1 Motivation and Problem Statement

Component-oriented programming models took hold in the area of distributed middleware platforms in the last couple of years. One of the main challenges addressed by deployment of middleware is to hide the details of concrete communication relationships from application developers—a feature also known as *distribution transparency*. It would thus be desirable to treat potentially remote components like local objects during development of distributed business applications. Today's platforms already provide this from a static point of view, e.g., by transparent installation of proxy objects. However, the naive use of this communication abstraction can considerably restrain the efficiency of such applications because every remote access crossing process spaces or node boundaries is an order of magnitude slower than its local equivalent. Besides design patterns for optimized data transmission, especially data replication techniques like *caching* and *prefetching*[1] proved of value in the past for solving this issue.

Caching is an orthogonal aspect of application logic

The problem in connection with the deployment of these techniques is that they represent orthogonal base functionality, unrelated to the actual business logic. Their proper use requires additional knowledge from application developers, which in turn increases the error-proneness of the corresponding code. Hence, it is sensible to outsource these techniques for data transfer optimization as an independent middleware service—an aspect—which is descriptively configurable and thus taken from the direct responsibility of the application developer. Hardly any approaches have been published that tackle

[1] In the following, we will simply use the term *caching* when referring to both caching and prefetching, since the latter can be seen as an extension of the former. (Cf. section 3.3 on page 100.) The distinction between caching and *replication* is made in section 2.1 on page 9.

this issue transparently at the interface of application components. This poses the main contribution of this work.

After years of skepticism, *Computer-Aided Software Engineering (CASE)* gradually managed to gain momentum. Model-driven approaches added to its success, allowing to directly generate code by transforming models.

Development process

Descriptive configurability of the caching aspect imposes the issue of integration into the software development cycle. The application developer can naturally enrich the application model with valuable information at design time regarding the applicability of caching for attributes of components and the application's tolerance to inconsistencies. This metadata can be transformed and used for generating code segments in the course of application development. The issues of modeling and the separation of roles and concerns in this respect form the second pillar of this work.

Adaptive, self-managing systems attracted a lot of attention in the recent past. Their goal is to let systems adapt themselves to changing environmental conditions.

Adaptivity

The gathered cacheability metadata mentioned above represents just an initial value that is subject to change at runtime. Such changes may have a lasting influence on the suitability of certain attributes for caching. This especially applies to read/write ratio of attributes, access patterns of component users and thus the working set of the cache, or even temporal dependencies between attribute accesses and the corresponding implication on prefetching. The runtime environment has to be instrumented and monitored to be able to dynamically react to such changes. Information gathered in this way can be used to adapt the system to changing conditions, to correct misconfigurations, or to automatically determine optimal configurations.

1.2 Scenarios

Cache method results of business tier

A number of universal application areas and usage scenarios for the proposed extensions shall be outlined in this section. In general, we will focus on *multi-tiered architectures* (see section 2.2.3 on page 21) like they are commonly found in today's (Web-based) business applications. Components of the *business tier* model business objects and processes. They run in application servers and encapsulate *state*, which they expose through their interfaces as *attributes*. Accessor and mutator *methods* may be used to query and manipulate these attributes. We basically try to *cache method results* of accessor methods in a transparent manner.

Web servers and fat clients

Clients benefitting from this extension are thus all application parts above the business tier. This includes both Web applications, i.e., Web servers constructing dynamic Web pages from business content, as well as traditional "fat client" applications that provide presentation logic by themselves.

eLearning

Examples for the latter category comprise applications with a higher degree of interactivity at the user interface than common Web-based applications. This includes distributed collaborative multimedia in general or online learning platforms in particular, e.g., the *Java-based Teleteaching Kit (JaTeK)* [Neu03]. Some sort of learning objects are typically used concurrently by a group of users. Changes to this material can only be made by tutors and authors; they occur rather seldom and consistency requirements are weaker than, e.g., for business applications. Thus, this class of data is suitable for caching.

Mobile Computing

Distributed mobile applications also belong to this category. Users are communicating using heterogeneous end devices and changing connectivity. In the context of the BIB3R project, a context framework [Lös02] was developed for centrally managing the state of online users and the capabilities of their devices. Client side caching of this information would help to reduce latency and expensive bandwidth consumption. This example scenario could even be extended towards weakly connected / partially detached applications, allowing offline clients to perform *disconnected operations.*

1.3 Objectives

Problems

Various problems have already been identified introductorily in the motivation. This section resumes this discussion, aiming at concrete goals and claims of this thesis. The major problems outlined above included the following points:

- Letting application developers integrate caching logic in their component code is inadequate because it forces them to reason about an orthogonal aspect.
- Transparent integration of caching at the level of application component interfaces has not been investigated sufficiently.
- Caching-related descriptive metadata for middleware configuration is not considered at design time by existing approaches. Valuable information already available at this early stage is thus lost for reuse and consistency requirements cannot be observed appropriately.
- Caching-related metadata is subject to changes at runtime. Related approaches consider either static preconfiguration or dynamic determination but not the combination of both, which would reduce the necessary learning phase while healing misconfigurations at the same time.

1.3.1 Goals

To address these challenges, we define a number of goals we want to achieve with the contributions of this work:

Middleware service for caching. Caching is an orthogonal aspect, which shall be transparently integrated as a descriptive middleware service.

Improved performance. The deployment of caching serves the original goal of reducing the user-perceived latency of access to application components in comparison to non-caching solutions.

Network traffic reduction. A side effect of the previous goal is that reusing cached method results at client side also reduces bandwidth consumption between client and server.

Distribution transparency. Since modern middleware platforms (see section 2.2 on page 14) already provide most if not all of the transparency concepts defined by ISO/ITU-T [ISO95], one major goal of our work is to maintain existing distribution transparency concepts as far as possible for the planned extensions of these systems.

Design process. Capturing caching-related metadata, i.e., cacheability, consistency, and prefetchability, shall be supported by the process of software design. Available information is to be gathered as early as possible for later reuse in model transformation and code generation.

Stability. Continuous and generative integration of modelling issues into the software developement cycle shall enable stabler software development.

Ease of use. The separation of caching from application concerns should also simplify the task of application component programmers since they will not have to care anymore for orthogonal caching issues.

Adaptivity. The system shall continuously adapt itself to changing access patterns, aiming at an optimal configuration of the caching service.

1.3.2 Non-Goals

A similar importance appertains to issues considered out of the scope of this work. This differentiation helps to focus on the main goals.

No Design patterns

Design patterns capture concepts and strategies that proved of value in exemplary application scenarios. They provide guidelines for application developers for building new applications. However, the awareness of the issues behind a pattern is explicitly required from developers. Our approach relieves developers from this awareness, letting them concentrate on solving the actual problems of their application domain. Design patterns for optimized state transfer may be used by the middleware service in a transparent manner but their explicit use at application level is discouraged.

No Caching algorithms

Much work has been invested in research and development of *caching algorithms*, including replacement strategies and consistency protocols. Although this work will have to deal with these issues as well, the primary goal is to reuse existing concepts as far as possible.

1.3.3 Claims

The following claims shall be verified in the context of this thesis:

1. The aspect of caching of component state can be transparently outsourced as a middleware service.
2. Early gathering of caching-related metadata stabilizes the development of distributed applications and reduces the necessary amount of hand-written code.
3. Adaptivity helps to reach appropriate configurations.

1.4 Requirements Definition

In this section, we will dispose ancillary conditions and requirements for the proposed extensions. These will help to further differentiate our solution from related work.

Why not cache everything?

Existing knowledge about access patterns and anticipated user behavior should be captured and utilized as early as possible to avoid wasting resources at runtime, especially during the initial learning phase before an optimal configuration is reached. The specification of application-specific tolerance to inconsistencies at the level of component attributes is furthermore crucial to inform the system about the precise conditions for employing caching technology. Without this knowledge, the system would potentially try to inappropriately cache data with stringent consistency requirements.

Why components?

Software components as we define them in section 2.2.1 on page 15 are a concept for separation of concerns. They encapsulate business logic in a self-contained manner while allowing higher-level service functionality to be built by composition. Additionally required orthogonal middleware services can be configured in a descriptive manner. We refer to this concept as *descriptive middleware* (see section 2.2 on page 14). Thus, the concept of components forms the natural basis for our approach to descriptive configuration of the caching aspect.

Why MDA?

Modeling tools and languages are a fundamental concept in Computer-Aided Software Engineering, which finally took hold in software development (cf. section 2.5 on page 58). The popularity of UML for application design led to model-driven concepts that try to bridge the gap between models and code. We will try to utilize these concepts to enable a continuous reuse of caching-related metadata throughout the development cycle.

Maximum transparency

Required distribution transparency can be divided into client transparency and server transparency, i.e., the necessity of additional precautions on either side. We want to avoid such violations of transparency. Hence, language modifications, additional hook methods or callbacks, etc. are discouraged, both at

the component business interface and below within component application logic. Descriptive markup is to be used exclusively to configure the middleware service.

Strict client-server

The heterogeneity of presented application scenarios may result in a variety of possible network connectivity alternatives between client and server below the application layer. We require strict adherence of client-server relationships, i.e., servers may only respond to client requests, no direct callbacks or broadcasts are permissible (e.g., for invalidation messages). Packet filters (*firewalls*) and Network Address Translators (*NAT*) between communicating partners are plausible reasons why such callbacks may simply fail at lower layers.

Weak consistency

We do not require strong consistency (see section 2.1.1 on page 11) but instead resort to weaker, *best-effort consistency* of cached data, i.e., we will tolerate staleness of data up to a certain degree the possible occurence of read/write anomalies. But the system shall nevertheless try to avoid conflicts by expiring cached objects adaptively according to their current access patterns. This degree of consistency is sufficient for a significant subset of the relevant application scenarios.

1.5 Cornerstones

According to the objectives mentioned in section 1.3 on page 3, the thesis at hand basically builds on three major cornerstones as depicted in figure 1.1.

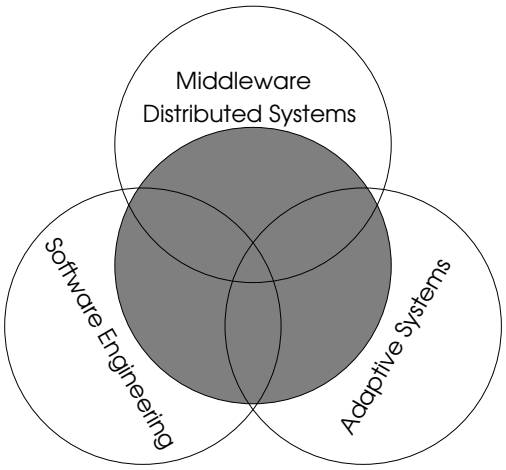

Fig. 1.1: Cornerstones of the thesis

Transparent middleware service. The aspect of data replication, i.e., caching and prefetching, shall be made transparent to the application developer and it shall be outsourced as an orthogonal, configurable middleware service. This cornerstone is subject of section 4 on page 107.

Software development process. The necessary non-functional properties shall be captured as early as possible during application development, at design time, in the form of additional metadata. Transformation of this metadata into necessary configuration descriptions for the runtime environment shall be enabled by appropriate code generators. This cornerstone will be discussed in section 5 on page 131.

Runtime adaptation. Metadata of component attributes concerning their cacheability and prefetching dependencies are subject to gradual changes at runtime due to changing access characteristics of application clients. The proposed middleware service shall be designed to dynamically adapt to changes of these parameters, aiming at automatic correction of misconfigurations and an optimal utilization of the infrastructure—especially of the caching service. The whole aspect of adaptation is the core topic of section 4.2 on page 111.

The nicest thing about standards is that there are so many of them to choose from.

Ken Olson (∗1926), founder of Digital Equipment Corp., 1977.

2 State of the Art

The purpose of this chapter is to provide the necessary background information on commonly used technology in the setting of this thesis. It provides definitions for required terms and explains many of the fundamental concepts used later for implementation. Although written in a consecutive style, its sections may also be read separately out of their context.

2.1 Caching in General

Caching is a fundamental concept of this work. A thorough introduction is therefore necessary. In its original meaning, a *cache*[1] is a hiding place or a secure place of storage for concealing and preserving provisions which are inconvenient to carry. In computer science, caches generally denote small fast memories holding recently accessed data that is slow or expensive to fetch, designed to speed up subsequent access to the same data. Since caches have been used over several decades in very different fields of computer science, we will give a general overview in this section.

Origin

Based on the idea that data queried or computed once will probably be needed again in the near future[2], *caches* store (a limited amount of) data retrieved from a certain *remote* source location near to its *local* sink. The meaning of "remote" and "local" depends entirely on the concrete context of use. The main goals for the deployment of caching are:

Performance in terms of lower perceived response times from the perspec-

[1] French, from *cacher*: to press, to conceal, to hide.
[2] i.e., *temporal locality of reference*, see below.

tive of data sinks, since required data can be served locally to some degree; and

Scalability in terms of increased throughput and higher maximum number of concurrently servable data sinks (clients) by reducing bandwidth consumption of the data channel between source and sink, and workload of the data source (the server), since clients are able to operate autonomously to some degree.

Caching vs. replication

In contrast to *replication* (cf. section 2.3.2 on page 50) of data, caching does not address *availability* and *fault-tolerance* of the augmented systems. While replicas may cooperate in an equal fashion, caches always represent only the "soft state" of a certain *primary copy*, usually the server. Furthermore, caches retain data that has been referenced; replicas may be filled in advance. Lenz distinguishes three levels of data objects [Len97, p.62]: application domain, logical level and physical level. At the physical level, several copies (replicas) of logical objects may exist. Logical objects map to real world objects of the application domain. They may contain redundancy in terms of overlapping information, which can be computed as a function of other data objects. In the context of our work, *cached method results* represent potentially redundant data items at the logical level, which may be replicated several times at the physical level. However, in the case of strict interpretation of the assumptions from definition 2.8 on page 17, redundancy is avoided and updates of logical data items are equivalent to updates of physical replicas.

Locality of reference

The exploitation of *locality of reference* is the most important way to achieve both goals, performance and scalability. There are several different distinctions for reference locality, whereof the first two account for the most common subset:

Temporal Locality. Once referenced objects will probably be accessed again in the near future;

Spatial Locality. Given a certain object, referencing "nearby"[3] objects increases the likelihood of accessing that object;

Geographic Locality. Geographically collocated clients will access similar objects (only relevant for distributed systems).

Semantic Locality. Knowledge about the semantics of accessed data may help to extract result data of subsequent queries, or at least portions thereof[4].

Stack model

The sequence of accessed objects is usually modeled as a *stack*, i.e., when an

[3] The meaning of "nearby" depends on the context of the referenced objects: Web pages contain links to nearby objects; an application's memory pages should be physically near to each other etc.

[4] This concept forms the basis of *Semantic Caching* as introduced independently by Keller/Basu and Dar *et al.* [KB94, DFJ+96] for queries of relational databases. Cf. section 3.1.3 on page 87.

object is accessed, it is "put" on top of the stack and it gradually moves deeper as other objects are accessed. The maximum number of objects on the stack, i.e., the *stack depth*, is referred to as the *working set*. This terminology shows the obvious parallels to memory and buffer management in operating systems and databases. Almeida *et al.* [ABCdO96] showed that temporal locality can also be characterized by the marginal distribution of the stack distance trace, whereas spatial locality relates to the notion of self-similarity, i.e., long-range correlations in the dataset.

The effectiveness of caching depends upon a number of factors, including the size of the user population that a cache is serving and the size of the cache serving that population (i.e., the cardinality of its working set).

In the following, we will discuss common problems in connection with caching and their solutions.

2.1.1 Consistency and Coherence

Just like replication, caching introduces global redundancy by creating multiple copies of single data items. Redundant copies have to be kept consistent to a defined degree, i.e., *coherence* of copies has to be ensured in such a way that different copies give the same values. This problem becomes even more pressing if multiple users/clients may concurrently alter these copies. Caches are usually client replicas grouped around one (or few) central server(s). The parallels to replication in distributed databases suggest the applicability of approaches for replication control (see section 2.3.2 on page 50) and distributed concurrency control (see section 2.3.3 on page 52). Different degrees of consistency are achievable depending on the selected methods. Gruber [Gru97] and Franklin *et al.* [FCL97] give comprehensive overviews of existing cache consistency protocols, a few of which will be discussed in the context of section 3.1.3 on page 87.

Strong Cache Consistency

Strong consistency is usually defined by *one-copy serializability (1SR)*, which is already a weaker degree of consistency than strict or atomic consistency: All processes see all operations in the same global order (cf. section 2.3.3 on page 52 and [BHG87]). There are several alternative approaches for achieving strong cache consistency, based on the assumption that modifications are synchronized at the primary copy (i.e., the server), e.g., *Client validation* assumes polling the status of a cache item upon every access, which does however not prevent concurrent modifications between the server's validation reply and the client's actual access. *Server invalidation* is based on invalidation messages the server broadcasts upon modifications of cache items, which is prone to poor scalability due to client state managed at server side. To achieve strong consistency with server invalidation, updates have to be delayed until all client

Validation vs. invalidation

caches have been invalidated, which in turn raises the question for appropriate time-out values. A more thorough discussion of client validation vs. server invalidation in the context of Web caching can be found in [RS02, Chap. 10].

Weak Cache Consistency

Convergence

Adaptive expiration time and piggyback validation

In general, weak consistency refers to any degree of consistency weaker than sequential consistency (1SR). The idea is to specify constraints for convergence, i.e., after a given period of time without modifications of cache items, caches converge to the same state. Strategies may include a number of heuristics, e.g., adaptive expiration timers assume that the average time between changes also applies in the future. Validation messages as explained above may be transferred piggyback with other messages between client and server: Either the server responds to a list of cache items for which validation was requested or it proactively emits a list of outdated items.

2.1.2 Replacement Strategies

Characteristics for classification

Cache size is usually limited, and if it gets exhausted, *replacement strategies* decide which items to keep and which to discard. For instance, Podlipnig and Böszörmenyi [PB03] mention the following characteristics for classifying replacement strategies: *recency* (time since last reference to an object), *frequency* (number of requests to an object), *size* of an object, *cost* of fetching an object, time since last *modification*, and (heuristic) *expiration* time. Different factors may be combined in weighted functions as a basis for replacement strategies. Furthermore, randomized strategies exist that incorporate a nondeterministic element into their decisions.

The most prominent representatives of cache replacement strategies are doubtlessly *Least-Recently-Used (LRU)* and *Least-Frequently-Used (LFU)*, which are also widely used in other application areas, e.g., management of database buffers and disk buffers, memory paging etc. LRU discards the least recently used items first. Obviously, this requires keeping track of what was used when, which is usually implemented by a stack or linked list. LFU counts how often an item is needed. Those that are used least often are discarded first. The *SIZE* strategy discards the biggest objects first. Cost, modification and expiration time (aging) are usually integrated factors in variants of recency/frequency-based strategies. A number of cache replacement strategies have been compared in the context of Web caching (see section 3.1.1 on page 84) by Podlipnig and Böszörmenyi [PB03].

Optimal strategy

Belady [Bel66] explained that the optimal replacement strategy is the one which replaces the object that will not be used for the longest time in the future. This optimum cannot be achieved because this would imply fortune-telling. However, it poses a goal and limit for realistic replacement strategies. The success of a cache replacement strategy is characterized by a low *cache*

miss rate (i.e., number of objects transferred from the original data source per total number of objects served), which corresponds to *page faults* in memory management.

2.1.3 Cacheability

What to cache?

Another issue in caching systems is which data items should be cached and which not—the *cacheability* of data. The simplest approach is to cache everything. This may however be too inefficient for application scenarios where (a subset of) data is too volatile, i.e., that changes too rapidly. The overhead for frequent invalidation would frustrate the benefits of caching, leading to a worse overall performance than simple access without any caching at all. Therefore, ways to specify cacheability of individual data items or classes of items have been introduced in areas where data items vary in expiration time, size, and cost for fetching. For instance, HTTP/1.1 defines a simple method for per-object specification of cacheability [FGM+99, Sect. 13.4].

When to cache?

The question about the time of incorporation of data in the cache is rather simple, in contrast to memory paging in operating systems, for instance. A cache in its original form contains only data that has been queried before by the client application it is serving. Strategies that shift the point of data transfer to the cache to an earlier time are commonly referred to as *prefetching* (see section 3.3 on page 100).

2.1.4 Granularity

Granularity generally refers to the size of parts of a greater whole. Given the need to transfer a certain amount of data D in packets of a certain size—the *granularity* g—the number of required interactions n can simply be calculated by

$$n = \frac{D}{g}$$

which generally shows that the number of required interactions increases reciprocally proportional with the granularity:

$$n \sim \frac{1}{g}$$

Let t_o denote the constant overhead time for each interaction and τ the factor time per data. The time per interaction t is then given by

$$t(g) = \tau g + t_o$$

and the overall transfer time by

$$T(g) = D\left(\frac{t_o}{g} + \tau\right)$$

Theoretically, the optimal case would be $n = 1$ for $g = D$, i.e., to transfer needed data all at once in $T = t_o + \tau$. However, it is usually impossible to determine in advance which specific data items actually belong to D. Responsiveness, i.e., the user-perceived time between request and delivery of data items, would furthermore increase to its maximum value $t = \tau D + t_o$. For caches, *coarse grained* cache items may waste bandwidth but save overhead for additional interactions required by *fine grained* cache items. Coarser grained cache items may additionally introduce benefits because of spatial locality of reference, i.e., subsequently required data may already be contained in the current working set. In practice, the optimal case is usually a trade-off between different factors, including overhead for interactions, expected average response time, cache size, etc.

2.1.5 Usage Examples in Information Systems

It has been mentioned introductorily that caches can be found almost everywhere in computer and software architecture. We will briefly give a few examples to illustrate the bandwidth of possible application scenarios.

Hardware

Hardware caches can be found at all levels of the memory hierarchy of modern computer architectures. For instance, a multi-level cache hierarchy accelerates data transfer between CPU registers and main memory. Physical cache memory resides on most modern hard disk drives to buffer access to the slower magnetic storage.

Operating systems

Most modern operating systems additionally use portions of main memory to further cache (hard) disk accesses. The *paging* mechanism of virtual memory management can also be seen as a form of caching since it obviously benefits from spatial locality of reference. A plethora of approaches for other *software caches* can also be found in *distributed systems*, a few of which will be introduced in the context of related work in section 3.1 on page 81.

2.2 Component-Oriented Middleware Platforms

In this section, we will give an overview of *software composition* (see section 2.2.1 on the facing page) and *middleware* (see section 2.2.2 on page 18) in general, followed by an explanation how these concepts are used in combination to build *multi-tiered architectures* (see section 2.2.3 on page 21).Three examples for standard component-based middleware platforms—EJB, CCM, and .NET—are given in sections 2.2.4–2.2.6. The relation of these platforms to *Web Services* is discussed in section 2.2.7 on page 41.

2.2.1 Software Components

The origin of the concept of software components is attributed to Doug McIlroy's famous paper about *"Mass Produced" Software Components* [McI69]. They are a fundamental basis of this thesis. Therefore, a clear definition is needed. The main problem with existing definition approaches found in literature are the different perspectives of participants in the software development process, which resulted in a number of different and sometimes contrary definitions.

Brown and Wallnau [BW98] collected four illustrating definitions for software components from different views. The first describes interfaces and implementation as the crucial constituents of a component on which the principle of loose coupling is based:

Definition 2.1 (Philippe Kruchten, Rational Software)
A component *is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.*

It should be noted that the architectural composition context of a component is also an integral part of this definition, whereas no assumptions are made about how to discover and bind components at runtime. This is subject of the second definition:

Definition 2.2 (Gartner Group)
A runtime software component *is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime.*

Independent deployment

A third definition outlines that components should have no implicit dependencies to other components or subsystems, a prerequisite for independent deployment:

Definition 2.3 (Clemens Szyperski [Szy02, p. 548])
A software component *is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.*

Business components

Components are often used to model business objects and processes. Hence, to enable loose coupling, a software component has to encapsulate all professional aspects of a self-contained business concept:

Definition 2.4 (Wojtek Kozaczynski, SSA)
A business component *represents the software implementation of an 'autonomous' business concept or business process. It consists of all the software artifacts necessary to express, implement and deploy the concept as a reusable element of a larger business system.*

These four definitions share a few common points but also reveal a number of differences as it was already pointed out in [BW98]: Kozaczynski and Kruchten agree on the coarse-grained nature of components whereas Szyperski and Gartner leave this issue open. Component autonomy and declaration of dependencies also lack a uniform description.

Definition 2.5 ([OMG03b, p. 8])
Component: *A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).*

Components are identifiable, modules are not.

The terms "component" and "module" are sometimes used interchangeably, like hinted at in definition 2.5 by mentioning replaceability. However, there are subtle differences: As noted in definition 2.3 on the page before, the major benefit of components is re-usability by composition. This is a feature not necessarily required from modules. Oestereich [Oes02] realized furthermore that components are usually identifiable in opposition to modules, which have no concept for identity of instances. Therefore, we shall distinguish a software module as follows:

Definition 2.6
A software module *is an autonomous, self-contained program unit characterized by information hiding, data abstraction, encapsulation, minimal interface specification, straightforwardness, and testability.*

Code reuse necessitates modularity

Aßmann [Aßm03] requires *modularity* as a prerequisite of *component models* to enable code reuse, the former being characterized by *information hiding* as well as *syntactic and semantic substitutability*. Other requirements for component models include *parameterization* to enable adaption and *standardization* to enable improved reuse. *Software components* do not only require a *component model* but also *composition techniques* like connection, extensibility, aspect separation, scalability and metamodeling, as well as a *composition language* to serve as a foundation for *software composition*.

MIL

Although the concept of software modules is much older, they have been in the focus of research in the mid 1980s when various *Module Interconnection Languages* (MIL, [PDN86]) evolved. Modules are often understood as units of packaging classes and routines, i.e., libraries. They are therefore also sometimes used to refer to units of component deployment, as seen in definition 2.5. Definition 2.7—a summary of Cheesman and Daniels' *UML Components* [CD00], which will be discussed in more detail in section 2.5.1 on page 63—does not explicitly mention the term "module" but it nevertheless confirms the distinction of modules and components based on identifiability.

UML Components

Definition 2.7 (compiled from [CD00])
A Component *conforms to a* Component Standard. *It needs a clear* Component Specification *that includes a set of supported* Component Interfaces. *The specifi-*

cation can be realized by multiple Component Implementations *that can in turn be deployed as* Installed Components*. These installations can be used to instantiate concrete* Component Objects*, which have their own data and a* unique identity*.*

Software components are conceptually based on *Component Models*—or *Component Standards* as in definition 2.7 on the preceding page—which manifest a metalevel for defining and describing components of a certain kind, i.e., their interfaces, parts, ports, roles, and other structural features. Cheesman and Daniels [CD00] furthermore understand a component model as a set of supported services, accompanied by rules that have to be obeyed to take advantage of these services. This anticipates a component model's implication of a *Component Framework* comprising the runtime and execution environment for components based on the aforementioned model. It interprets a *Component Language* to connect components according to a *Composition Technique* implicitly associated with the component model. Examples for such models and frameworks include component-oriented middleware platforms like CORBA Components (see section 2.2.5 on page 29), Sun's (Enterprise) JavaBeans (see section 2.2.4 on page 22), the Microsoft Component Object Model and it's extensions (see section 2.2.6 on page 38), but also simpler constructs like Architecture Description Languages (ADL [MT00], descendants of the aforementioned MIL [PDN86]) or pipes & filters [McI64, BMR+96].

Our Refined Definition

In the context of this work, we will embrace the definitions listed above. We furthermore refine definition 2.4 on page 15 with special regard to identifiability, state encapsulation, and access, which is reflected in various component models:

Definition 2.8
Components are identifiable program artifacts that encapsulate the state *of business objects and business processes. This state is manifested in* component attributes*, which a component exposes via accessor and mutator* methods *of its interface.*

Business components

Hence, we focus especially on business components and put less attention on other component types, e.g., infrastructure components. We will heavily rely on the assumption of definition 2.8 for the implementation of our caching service in section 4 on page 107. For instance, JavaBeans [Sun02b] use a pattern of `get/set` method pairs for modeling accessors and mutators. However, this definition does not narrow the impact of our contribution because this assumption is present in many of today's component-oriented middleware platforms, as we will discover later in this section.

2.2.2 Middleware in General

While most component models as defined in section 2.2.1 on page 15 imply a component framework that implements the corresponding runtime environment, no assumptions are made about the presence of middleware services.

Middleware evolved long before the advent of software componentization—among other sources—from the host of *Transaction Processing Monitors* (TP Monitors) [MW88]. A convenient general description of the term *Middleware* is given in definition 2.9

Definition 2.9 (David E. Bakken [Bak02])
Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system, [...] it provides a higher-level building block for programmers than APIs provided by the operating system. [...] Middleware can be considered to be the software that makes a distributed system programmable.

Middleware services

This definition not only requires a middleware to provide a communication abstraction for distributed applications, it also hints at the existence of *orthogonal middleware services* transparently provided by the framework according to additional descriptors further delineating *non-functional properties*[5]. Examples of such programming abstractions include but are not limited to security, transactionality, persistence, replication, concurrency, location, and mobility. Another objective of middleware platforms is to shield programmers from *heterogeneity* imposed by distribution, operating systems, hardware but also by programming languages, which closely relates it to the goal of *distribution transparency* of the Reference Model for Open Distributed Processing (RM-ODP) [ISO95, p. 17, p. 64 et seq.].

Distribution transparency

The ISO and ITU-T standardization organizations defined the Reference Model for Open Distributed Processing (RM-ODP) in 1995 to foster portability across heterogeneous platforms, collaboration between ODP systems in terms of information exchange and use of functionality, as well as *distribution transparency*, for which a number of selective concepts are defined to hide the consequences of distribution from both end users and system developers: Access, failure, location, migration, relocation, replication, persistence, and transaction transparency.

These orthogonal services and heterogeneity abstractions are also in the focus of newer research approaches like *Aspect-Oriented Programming*, which shall be subject to closer inspection in section 2.5.6 on page 75

[5] While the business application that is running on a certain middleware platform is responsible for the *functional* properties of its interface implementation, i.e., the business logic itself, the middleware platform caters for all remaining *non-functional* properties, which are also sometimes referred to as "ilities"[FBLL02] meaning qualities like reliab*ility*, availab*ility*, manageab*ility*, dependab*ility*, etc.

Definition 2.9 is more or less confirmed by Schantz and Schmidt:

Definition 2.10 (Schantz and Schmidt [SS01])
Middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to

1. *Functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate and*
2. *Enable and simplify the integration of components developed by multiple technology suppliers.*

Bakken [Bak02] further categorizes currently existing middleware approaches into the following classes:

Remote Procedure Call (RPC) provides the abstraction of transparent synchronous procedure invocation across system boundaries but offer only limited support for concurrency or exception handling.

Distributed Object Middleware raises the object-oriented axioms—encapsulation, inheritance, and polymorphism—to the level of transparent distribution by leveraging the primitives of RPC. Most platforms provide a variety of programming abstractions and services partly inherited from their roots in Transaction Processing Monitors (TPM)[6]. Examples include the Object Management Group's Common Object Request Broker Architecture (CORBA) [OMG04a], Microsoft's Distributed Component Object Model (DCOM) [MSd], and Sun Microsystems' Java Remote Method Invocation (RMI) [Sun02a]. Figure 2.1 on the following page shows for the example of CORBA how *Object Request Brokers (ORB)* of this middleware category transparently distribute requests to local and/or remote servants. In the case of CORBA, the Internet Inter-ORB Protocol (IIOP)—the Internet binding of the Generic Inter-ORB Protocol (GIOP)—is used for communication between ORBs at remote hosts. Client *stubs* and server *skeletons* proxy object requests and take care of argument and result (un)-marshalling.

Object Request Brokers

Message-oriented Middleware (MOM) generalizes the well-known mailbox principle to asynchronous message queues, which can be used to loosely couple system topologies. Further middleware services include persistence, replication, and real-time capabilities[7].

Distributed Tuples comprise distributed Relational Database Management Systems (RDBMS)[Cod70], TPMs, but also tuple space approaches like *Linda* or *Java Spaces* in *Jini* [Gel85, SCA, Sun].

[6] See for example [MW88].
[7] See [DSO03] for an overview.

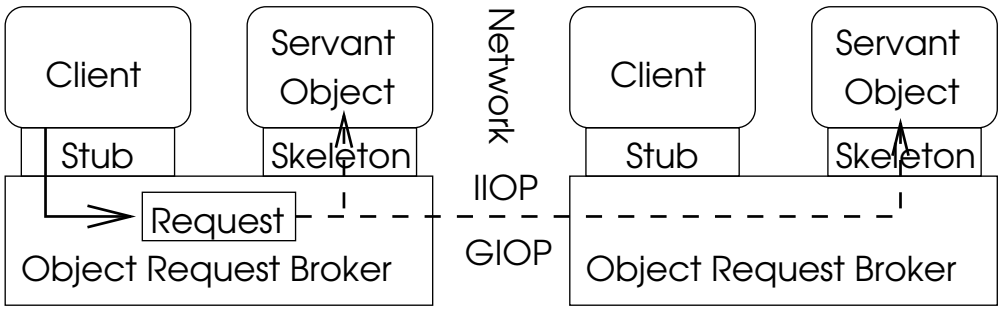

Fig. 2.1: Object Request Brokers according to [OMG04a]

Another category that Bakken left out in his enumeration is formed by *Distributed Shared Memory Systems (DSM)* [PTM97], which shall be mentioned here for the sake of completeness:

Distributed Shared Memory Systems (DSM) are typically found in high speed parallel computing environments. They share a mutual set of distributed memory pages whose consistency is maintained by the memory subsystem using multi-cast and similar technologies. Support for distribution is provided by both, hardware and software architectures.

If caching is used as a special form of partial replication, DSM systems typically rely on page-oriented caching strategies and procedural or message-oriented middleware on object caching algorithms, respectively. More details are described in section 2.1 on page 9.

Distributed Component-Oriented Middleware

Distributed Component-Oriented Middleware platforms technically form extensions of the above mentioned *Distributed Object Middleware*, merging this approach with older technology of Transaction Processing Monitors, like CICS, Tuxedo, and Encina, which is why they are also often referred to as *Component Transaction Monitors (CTM)*. The major benefit of component-oriented platforms is that they provide *descriptive* middleware, i.e., they allow the use of middleware services to be specified descriptively, relying on the runtime environment to interpret and fulfill these descriptions. Components simply implement business/application logic; the middleware provides the container that encapsulates this logic and integrates additional services. In contrast, *explicit* middleware requires application programmers to explicitly access these services in their program code. Schmidt *et al.* [SSRB00] describe a similar concept in their pattern *Component Configurator* (cf. section 2.5.4 on page 70). A few example platforms shall be examined closer in the ensuing discussion.

Descriptive vs. explicit middleware

2.2.3 Multi-tiered Architectures

Component-oriented middleware as described above is commonly used for the business tier of so-called *Multi-tiered Architectures*. The schematic principle of this architecture style for building distributed applications is drawn in figure 2.2.

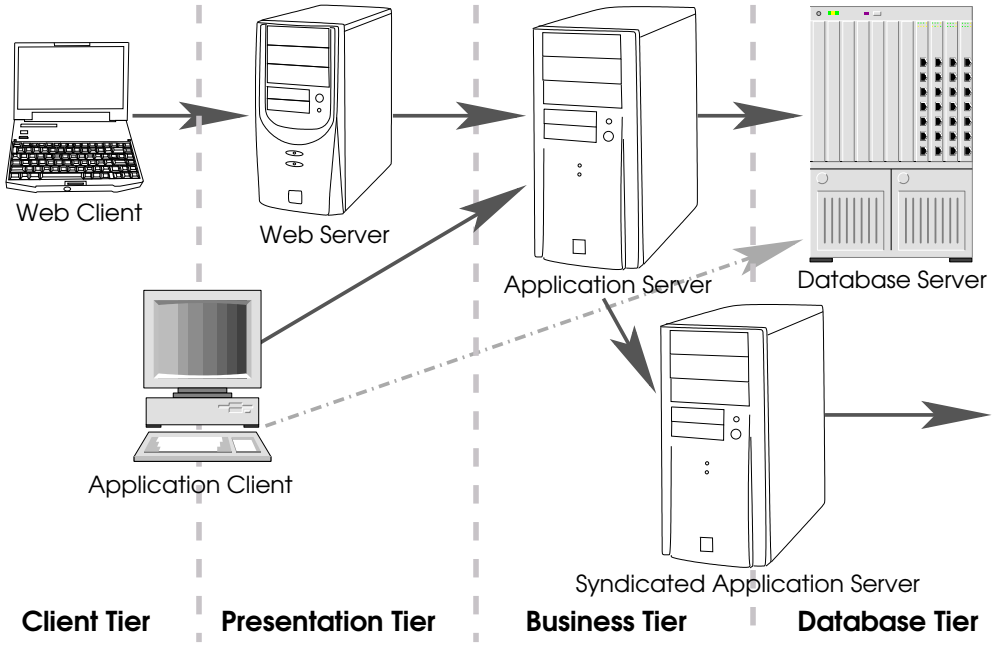

Fig. 2.2: Multi-tiered architecture for distributed applications

1. The **database tier** forms the basic layer for accessing business data.

2. On top of that, the **business tier** encapsulates business objects and processes—typically modeled as components running inside an application server that provides the respective component framework as a runtime environment. Apart from the database tier, application servers may in turn syndicate back-end services of other application servers not necessarily conforming to the same component model. This process is part of a larger concept often referred to as *Enterprise Application Integration (EAI)*.

   EAI

3. The third layer is formed by the **presentation tier**, which is usually implemented either by Web servers or Web containers serving dynamic Web content.

4. The **client tier** is formed by thin Web clients (browsers) or by fat application clients, which have to provide the presentation logic, i.e., the user interface, by themselves[8].

[8] Some publications assume that *fat clients* also execute business logic and refer to our distinc-

It was quite common for older applications to directly access the database tier without an intermediate business tier. They had to provide the business logic as well.

In today's complex scenarios, the above described EAI process introduces a recursive element to the tier principle. This range of different possible combinations—two, three, four, or even *n* tiers—explains why the term "multi-tiered" was coined. A few application server platforms that enable such architectures will be introduced in the following sections.

2.2.4 Enterprise JavaBeans

The Enterprise JavaBeans technology will be used as a reference platform for the demonstration of various concepts that will be proposed in section 4 on page 107. We will therefore introduce it in a slightly more detailed manner than the other example platforms in section 2.2.5 on page 29 and section 2.2.6 on page 38.

Sun Microsystems' *Enterprise JavaBeans (EJB)* component technology emerged in 1998 when the Version 1.0 of its specification [MH98] appeared as a result of Sun's endeavors to raise the older *JavaBeans* technology[9] [Sun02b] for client-side component-oriented programming to the level of *component-oriented middleware*—much in the way that Microsoft extended its *Component Object Model (COM)* to *COM+* and *DCOM* (see section 2.2.6 on page 38). The specification has gone through a number of revisions since then [MH99, DYK01, DeM03] during which it evolved into a *de facto* industry standard. A new major revision—Version 3.0 [DeM04]—is currently under development. The main goals of EJB include:

- a common component model for developing and composing Java-based distributed applications;
- abstractions to hide the complexity of transactionality, access control, and persistence from application developers, i.e., *distribution transparency* in terms of the RM-ODP [ISO95];
- a platform for *Commercial-Of-The-Shelf (COTS)* components, enabling a marketplace for different component and container vendors.

Relation to the Java 2 Platform Enterprise Edition

The EJB component model and container framework is part of the Java 2 Platform Enterprise Edition (J2EE)—Version 1.4 [Sha03] in the case of the current

tion by the terms of *thin* and *think Web clients*. We will nevertheless denote application clients with more than simple Web browser functionality as *fat clients*.

[9] JavaBeans was first released in 1996/97.

EJB Version 2.1 [DeM03]—the larger runtime environment around the mere component container. J2EE also comprises infrastructure services like security, transaction management, naming, and various APIs for processing of *Servlets* and *JavaServer Pages (JSP)*, messaging (JMS), mail, database connectivity, XML parser technology, support for the *Simple Object Access Protocol (SOAP)* and *Web Services* (see section 2.2.7 on page 41), among others. Figure 2.3 gives a high-level overview of the elements of the J2EE architecture.

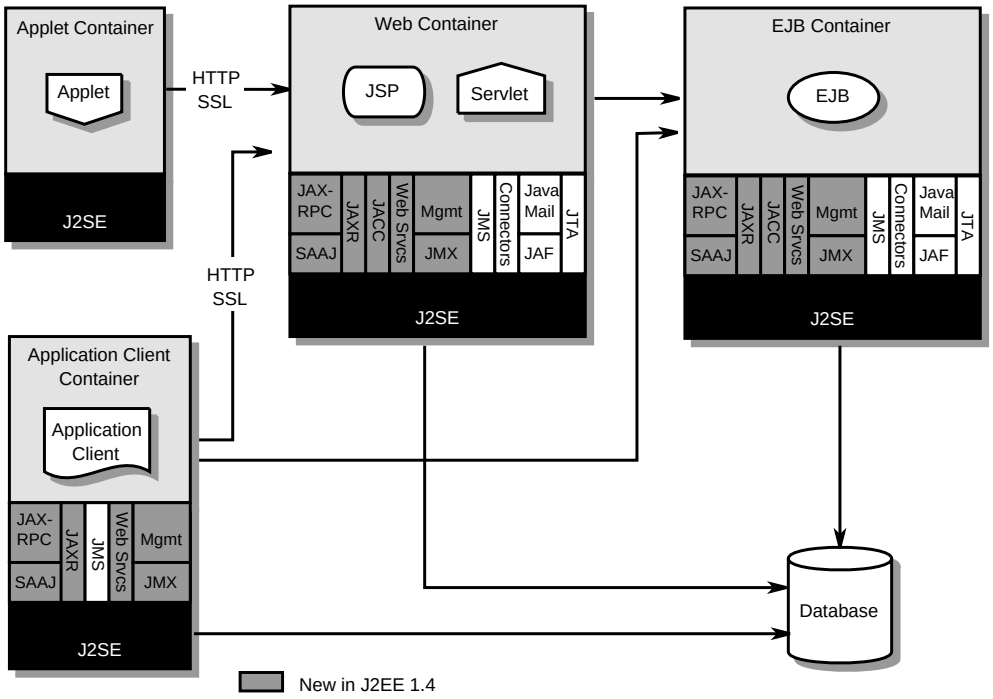

Fig. 2.3: J2EE architectural overview [Sha03]
Legend: EJB—Enterprise Java Beans; J2SE—Java 2 Standard Edition; JACC—Java Authorization service provider Contract for Containers; JAF—Java Activation Framework; JAXR—Java API for XML Registries; JAX-RPC—Java API for XML Remote Procedure Call; JMS—Java Message Service; JMX—Java Management eXtensions; JSP—Java Server Pages; JTA—Java Transaction API; SAAJ—SOAP with Attachments for Java

Multi-tiered architecture

The J2EE describes a *multi-tiered architecture* (see section 2.2.3 on page 21): Web clients (i.e., plain Web browsers and Java Applets) and application clients access server-side J2EE applications via the Hypertext Transfer Protocol (HTTP, [FGM+99]) over a Web container providing the *presentation logic* either by means of JavaServer Pages or Servlets. Application clients furthermore have the opportunity of directly accessing the *business logic*—encapsulated by EJBs in the EJB container—taking care of presentation logic by themselves.

Roles in Component Life-cycle

One of the crucial achievements of Enterprise JavaBeans was the definition of six independent roles in the life-cycle of software components:

1. *Enterprise Bean Providers* are application domain experts without special knowledge about system-level programming. They provide the actual business logic as reusable components, i.e., Enterprise Beans.
2. *Application Assemblers* are also domain experts who compose components from various Bean Providers into larger deployable application units, which also includes incorporating other application components like JavaServer Pages.
3. *Deployers* install single Enterprise Beans and whole Enterprise Application Archives in particular operational environments about which they have expertise. This installation process also includes customization of applications by means of provided parameters.
4. *EJB Server Providers* are experts in operating systems, middleware, or database systems catering for low-level platform services like distribution, transaction management, persistence, etc.
5. *EJB Container Providers* are currently assumed to be the same vendors as the EJB Server Providers. Their responsibility is to provide the operational environment for components, including tools for deployment, monitoring, and management.
6. *System Administrators* configure and administrate a concrete server installation using the management and monitoring tools provided by server and container providers.

Roman *et al.* [RAJM01] further mention *Tool Vendors* as an additional party responsible for providing necessary tools especially for Bean Providers and Application Assemblers.

These roles—in combination with the contracts specified between them throughout the specification—assure a clear *separation of concerns* and enable a modular design of the component environment. This can also be seen as an interpretation of the Latin proverb *divide et impera*[10] that proposes to maintain power and effectiveness by splitting up a larger problem and distributing responsibilities.

The EJB Component Model

The Enterprise JavaBeans component model distinguishes three different life-cycle categories of Beans:

[10] divide and conquer/rule

Entity Beans form the basic building blocks of most EJB-based applications. They wrap the (persistent) state *business objects*, thus mapping directly to back-end database tables—instances representing tuples, attributes representing columns. Roman *et al.* [RAJM01] call them the "nouns" of business applications. The object-relational (O2R) mapping of these entities to persistent tuples in a database can be handled either manually by the Bean provider, i.e., *Bean Managed Persistence* (BMP), or automatically by the container, i.e., *Container Managed Persistence* (CMP). The latter alternative furthermore introduces the option of *Container Managed Relations* (CMR), which includes automatic loading and consistency handling of different relation types between entities.

Session Beans are—corresponding to the above mentioned analogy—the "verbs" of business applications operating on entities, i.e., they represent the *business processes* manipulating the state of business objects. The specification introduces yet another distinction:

Stateless Session Beans maintain no conversational state between consecutive operation calls.

Stateful Session Beans work exclusively for a certain client, maintaining a session state between this client's calls.

Message-driven Beans represent actions similar to Stateless Session Beans. However, they can only be invoked asynchronously by sending weakly typed messages to them. This Bean type has been introduced in Version 2.0 [DYK01] to open up the platform to the *Message-oriented Middleware* paradigm [DSO03].

Table 2.1 summarizes the differences between these component life-cycle categories.

Tab. 2.1: Enterprise JavaBeans life-cycle categories

Bean type	State	Identity	Behavior
Stateless Session	no	no	potentially transactional
Stateful Session	transient	non-persistent	potentially transactional
Message-driven	no	no	asynchronous invocation
Entity	persistent, visible, managed by component or container	persistent, visible as `PrimaryKey`	potentially transactional

Bean providers essentially have to create the programming artifacts in figure 2.4 on the following page for every single Bean.

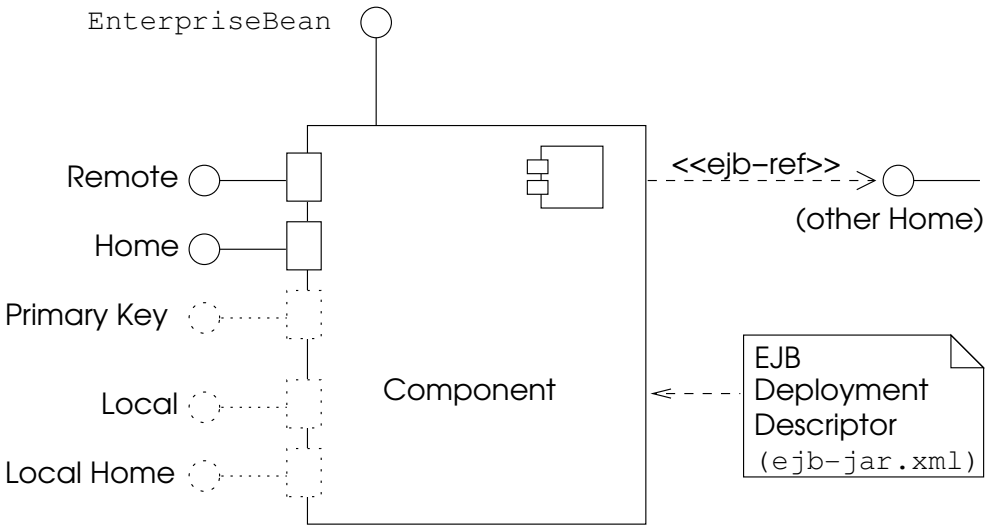

Fig. 2.4: Abstract model of EJB

Bean class. The Bean class captures the actual business logic of the bean, i.e., the operations working on some kind of state (either conversational or persistent). It has to implement `javax.ejb.EnterpriseBean` or one of its sub-interfaces as an internal handle for the container, depending on the bean type (see table 2.1 on the page before).

Remote interface. The Remote interface is also often called *Business interface* since it comprises all externally callable operations of a business object or process, including reading and updating attributes of its internal state.

Home interface. The Home interface was introduced to bundle operations that are not specific to a particular instance of a Bean but rather concern the whole *extent*[11] of a bean type. This primarily includes life-cycle operations[12] for creating and destroying instances but also *finder* and *select* methods for retrieving (collections of) instances according to some query criteria or for performing certain aggregating operations, respectively.

Primary keys. Persistent identification of *Entity Bean* (see table 2.1 on the preceding page) instances in the context of an installation of a *Home interface* is provided by *Primary Key* classes, which usually capture the contents of one or more columns of an entity's corresponding tuple in the back-end database.

Local interfaces. Version 2.0 [DYK01] furthermore introduced the concept of *Local Interfaces* for efficient communication within the boundaries of the server Java Virtual Machine (JVM). They basically correspond to Remote and Home interfaces but lack support for transparent distribution.

[11] In object-oriented databases, the *extent* comprises all instances of a particular type.
[12] so called *factory* operations, named after the Factory Method pattern described in [GHJV94]

Deployment descriptors. The specification requires the presence of an `ejb-jar.xml` descriptor file in every Bean archive containing the metadata necessary for configuring middleware services appropriately during deployment and for referencing collaborating components (≪ejb-ref≫). This is part of the concept called *implicit* or *descriptive middleware* by Roman *et al.* [RAJM01], which means that middleware services like transactions, security, persistence, etc. do not have to be accessed in an explicit, programmatic fashion. The runtime environment transparently inserts the desired functionality instead, according to the descriptions composed by Bean Providers, Application Assemblers, and/or Deployers. EJB container providers usually provide support for special container functionality by means of additional vendor-specific deployment descriptors.

Note that Message-driven Beans do not have any synchronous external interfaces (Home, Remote, etc.) at all since they are invoked asynchronously upon arrival of messages in certain *Queues* or *Topics*.

Loose coupling

The motivation behind this decoupling of interfaces and implementation is to enable the container to proxy each and every access to Bean instances by providing the direct implementations of Home and Remote interfaces by itself as a kind of adapter[13]. This allows container providers to transparently add additional behavior[14] as required. Apart from transactions, security, and persistence, this also includes *instance pooling*, another important concept of Enterprise JavaBeans: The container may keep a certain number of Bean instances, which it does not have to instantiate once they are needed for servicing client requests. Deleted instances are not destroyed immediately but rather returned to the "pooled" state, which enables fast reuse for the next request. The same technique is also applied to other resources, like threads, database connections, etc. to improve overall throughput and scalability.

Pooling

There are basically two alternative ways to implement the container's role as a runtime environment: Either by static generation of container glue code at deployment time or by using meta-programming mechanisms (see section 2.5.5 on page 71) at runtime. The latter provides more flexibility because additional functionality may be inserted after the application has already been deployed.

Transaction Management

Transaction management is a key element of the EJB architecture. The container transparently coordinates different (distributed) *resource managers*[15]. De-

[13] Cf. structural pattern "Adapter" in [GHJV94].

[14] Cf. "Chain of Responsibility" pattern in [GHJV94] or "Interceptor" pattern in [SSRB00].

[15] Resource managers are servers like, for instance, database management systems, capable of taking part in the *Two-Phase-Commit (2PC)* protocol. More details on distributed transactions are explained in section 2.3 on page 46.

marcation of transactions can be distinguished as follows:

User transactions allow clients to make explicit calls to `begin()` and `commit()` or `abort()` each transaction using `UserTransaction` contexts of the *Java Transaction API (JTA)*.

Bean-managed Transactions leave it to the bean provider to explicitly `begin` and end (i.e., **`commit`** / `abort`) transactions.

Container-managed Transactions require the bean provider only to give per-method specifications of transactional capabilities for its components using the values for the transaction attribute in the deployment descriptor as shown in table 2.2. The container will handle existing and/or new transaction contexts appropriately. It assumes that a bean wishes to commit the transaction unless it explicitly invokes the method `setRollbackOnly()`.

Transactions and caching

Especially user transactions require more attention if results of Bean invocations are subject to caching as intended in section 4 on page 107. Protocols for *intertransactional caching* will be introduced in section 3.1.3 on page 87 as a possible approach to this challenge.

Tab. 2.2: Transaction attributes in EJB

Transaction attribute	Existing context	No context
`NotSupported`	suspended before and resumed after method processing	no context created
`Required`	honored and used	new context started
`Supports`	honored and used	no context created
`RequiresNew`	suspended before and resumed after method processing; new context started	new context started
`Mandatory`	has to be set by calling client	exception
`Never`	exception	no context created

More details of the EJB component model (up to Version 2.0 [DYK01]) are explained, for instance, in Ed Roman's "Mastering EJB" [RAJM01].

History and Future

The initial Version 1.0 [MH98] was not much more than a preliminary release trying to provide a common basis for the emerging market of EJB server/-container vendors. It was quickly superseded by Version 1.1 [MH99], which made support for Entity Beans mandatory, introduced assembly descriptors,

and required explicit description of environmental dependencies, among other improvements.

EJB 2.0

The next major release, Version 2.0 [DYK01], introduced a number of new features: *Message-driven Beans* are a new Bean type, triggered by arriving JMS messages. *Container Managed Persistence (CMP)* support was enhanced, including *Container Managed Relationships (CMR)*. *Local Interfaces* have been introduced for lightweight access to components without support for potential distribution. *EJB QL* is an object-relational query language for `finder` and `select` methods. *`Select` methods* and additional business methods are now allowed on Home interfaces, similar as in CORBA Components (cf. section 2.2.5).

EJB 2.1

Enterprise JavaBeans v2.1 [DeM03] standardized further developments introduced meanwhile by various container vendors. These developments basically include support for accessing and providing *Web Services*. A *timer service* allows for coarse-grained, transactional, time-based event notifications, which is useful for managing business processes. Furthermore, enhancements to Message Driven Beans and EJB QL complemented the addition of this version.

EJB 3.0

Future Developments. The upcoming EJB 3.0 (at the time of writing in early draft state) targets at major points of criticisms of previous revisions [DeM04, pp. 7]:

- The number of programming artifacts shall be reduced by *extensive use of metadata annotations* as proposed by Java Specification Request 175 [Bra04] and soon provided by Java 2 Runtime Environment v5 for "configuration by exception", encapsulation of environmental dependencies, and object-relational mapping.
- *Simplification of Bean types* allows for programming EJBs like "plain old Java objects" (POJOs) as already provided by the JBoss open-source application server [FR03, BB03], among others;
- EJB QL shall be enhanced further.

Reduced complexity

These improvements can be summarized as a tendential reduction of the gradually bloated specification's complexity towards leaner, more modular concepts for plug-in–based introduction of new middleware services in an aspect-oriented manner (cf. section 2.5.6 on page 75).

2.2.5 CORBA Components

The Object Management Group (OMG) started in 1999 to create its own specification for server-side component-oriented application development—*CORBA Components (CCM)* [OMG02a]—as a response to Microsoft's older approaches based on the (Distributed) Component Object Model (D/COM) and

the Microsoft Transaction Server (MTS) (see section 2.2.6 on page 38) and Sun Microsystems' JavaBeans technology [Sun02b] for client-side component-oriented programming. The emerging Enterprise JavaBeans specification (see section 2.2.4 on page 22) furthermore led to number of extensions and improvements. The CORBA Components Model (CCM) is based on the OMG's Common Object Request Broker Architecture (CORBA) [OMG04a] as the underlying distributed object middleware platform. On account of being a unification attempt, the CCM tries to leverage potentials of both concurring worlds, EJB and COM, while adding further advantages at the same time. Examples include platform-neutrality in opposition to DCOM/MTS and language-neutrality in contrast to Java/EJB.

Unfortunately, this "best-of-breed" strategy is responsible for the even higher complexity of the CORBA Components specification in comparison to the Enterprise JavaBeans or Microsoft COM technology. This major drawback is the reason for the comparatively small number of commercial and open-source implementations of this specification. However, the OMG's proposal provides a much cleaner architecture from an academic viewpoint than, e.g., the EJB specification. For this reason, we will closer examine a few features and concepts of this architecture. A more in-depth comparison between the three competing technologies—Enterprise JavaBeans, Microsoft (D)COM, and CORBA Components—can be found, e.g., in [Poh99].

CORBA Components and CORBA

While CORBA [OMG04a] itself provides only a basic platform of explicitly programmable middleware services, the CORBA Components Model offers more descriptive means for specifying implicit component behavior, similar to and in some points far beyond Enterprise JavaBeans. It explicitly aims at modeling business objects and processes even though any business semantics have been strictly avoided in the specification. Building on top of CORBA as the underlying communication layer and middleware service platform, the component model is added as an additional abstraction layer for describing application architectures, as depicted in figure 2.5 on the facing page. Just as in EJB, the *container* provides the runtime environment for components, encapsulating access to them and transparently integrating middleware services as required by additional component descriptions.

IDL The *CORBA Interface Definition Language (IDL)* [OMG04a] has been augmented by numerous meta-model elements for capturing the necessary additional constructs. However, these augmentations can be mapped completely to equivalent CORBA IDL v3.0 statements. These additional language constructs also provide an easier access to component-oriented programming than, e.g., EJB, where components do not exist as explicit programming language artifacts.

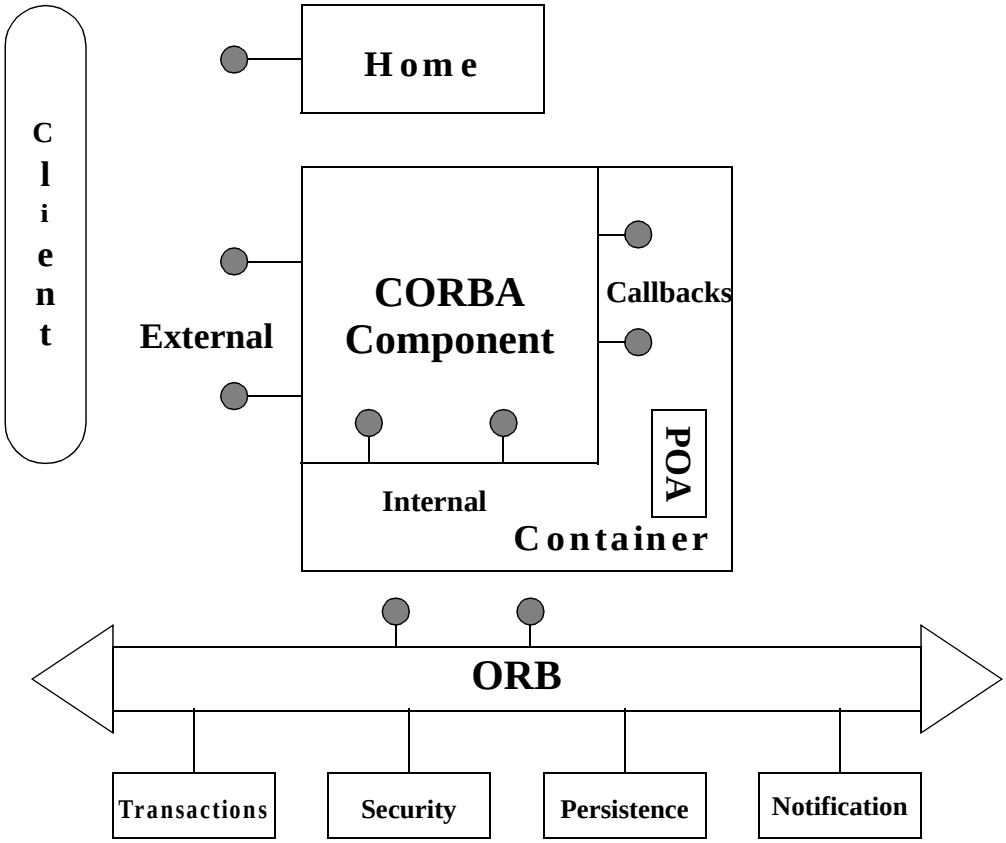

Fig. 2.5: CORBA Components container architecture [OMG02a]

The Development Cycle

Development of component-oriented applications in CCM comprises the following steps:

1. A `component` specification is described in IDL, which comprises its interface(s), including one or more Home interfaces (cf. EJB).
2. Next, the *Component Implementation Definition Language (CIDL)* is used to describe the framework for a possible corresponding implementation of the component specification. This primarily includes the life-cycle category, persistence mapping, and segmentation of facets. Code generators are used to create various interfaces and abstract base classes for the actual implementation in an arbitrary target language for which a CORBA-binding exists, in accordance with the *Component Implementation Framework (CIF)*. *CIDL* *CIF*
3. These base classes and program skeletons have to be extended and implemented by the application programmer.
4. Complete implementations are bundled in archives together with their appropriate component descriptors.

5. Components can then be assembled and pre-configured in using assembly descriptors and component property files. This also poses an extension of the EJB specification, which introduces only one descriptor (`ejb-jar.xml`) for everything from component and resource description to persistence mapping an assembly.

6. Such component packages can finally be deployed at conforming application servers.

The CORBA Components Model

CORBA Components provide a much finer distinction of a component specification's different artifacts than its competing models, COM and EJB. Figure 2.6 shows an abstract view on *features* of the CORBA Components Model, including *supported interfaces*, *facets* and *receptacles*, *event sources* and *event sinks*, as well as *attributes*.

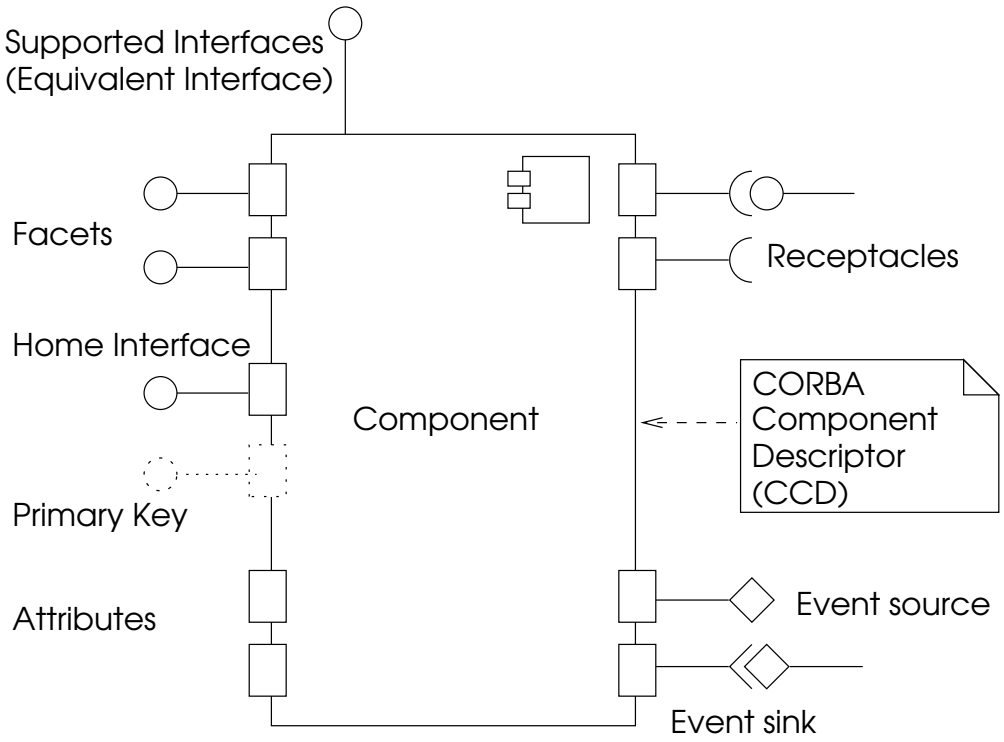

Fig. 2.6: Abstract model of CCM

Supported interfaces can be used to extend the *equivalent interface* of a component—responsible, for instance, for navigational access to *facets*—by means of normal inheritance.

Facets are used to model *provided interfaces* of distinct application aspects or

roles of a component[16]. This opens up the possibility to encapsulate the implementation of different facets by separate *servant* classes. Available facets of a component can be queried and navigated from its *equivalent interface*.

Receptacles or *used interfaces* are used to model unidirectional connections of interfaces provided, for instance, by other components. This mechanism can be used during *component assembly* to configure networks of collaborating components by assigning provided and used interfaces to each other according to their specified roles[17].

Event sources *emit* $(1:1)$ or *publish* $(1:n)$ arbitrary event objects with call-by-value semantic[18]. Technically, event sources are special *receptacles* since they call back *event sink* interfaces that have to be announced to them.

Event sinks are the corresponding *consumers* of emitted or published events, which are "pushed" by *event sources* to this special kind of facet. Events are primarily intended for notification of components within the context of a deployment[19].

Attributes represent externally visible properties of a component, which can be read and written by *accessors* and *mutators*. Their primary purpose is to allow (pre-)configuration of components and component instances during assembly and deployment.

Home interfaces are modeled differently from the *features* explained above. The Component IDL has a special construct for defining Home interfaces and their management relationship to a certain component. In analogy to EJB, Homes are used to capture operations that are not bound to a specific component instance. This primarily includes *factory* and *finder* functionality but also arbitrary auxiliary methods. A `home` definition in IDL may also include a `primarykey` definition if needed for identification.

Primary keys are value objects used for persistent identification of *Entity* components (see table 2.3 on the next page) in the context of a concrete installation of their *Home interface(s)*.

Deployment descriptors provide metadata necessary for configuring middleware services appropriately during deployment, just as in EJB (cf. figure 2.4 on page 26). Details about CORBA Component Descriptors are discussed on page 35.

[16] This concept is taken from COM (cf. section 2.2.6 on page 38).

[17] At this point, CCM goes far beyond what other component models provide; e.g., EJB only allows to specify expected naming service entries of collaborating Beans but leaves the processes of instantiation entirely in the responsibility of the Bean itself.

[18] Cf. [OMG04a] for the differences between reference and value semantics.

[19] Although the concrete implementation of the required event mechanism is left to the container, asynchronous invocation as with *Message-driven EJBs* is not explicitly supported by this simple mechanism.

Similar to the Enterprise JavaBeans model (cf. table 2.1 on page 25), CORBA Components distinguishes different life-cycle categories, as shown in table 2.3.

Tab. 2.3: CORBA Components life-cycle categories

Category	State	Identity	Behavior
Service	no	no	non-transactional
Session	transient	non-persistent	non-transactional
Process	persistent, but invisible to the client, managed by component	persistent, potentially visible through user-defined operations, managed by component	potentially transactional
Entity	persistent, visible, managed by component or container	persistent, visible as `primarykey`	potentially transactional

Service components correspond to *Stateless Session Beans*, *Session* and *Process* to *Stateful Session Beans*, and *Entity*, of course, to *Entity Beans*. EJB's *Message-driven Beans* have no direct equivalent in CCM. *Process* components represent a special type of components that is best compared with DCOM components with Microsoft Transaction Service (MTS) support (cf. section 2.2.6 on page 38).

For the implementation of components, CCM introduced the concept the *Component Implementation Definition Language (CIDL)* as an intermediary step between interface specification and binary implementation, which is much closer to the concepts of UML Components[20]. The basic element of a CIDL definition is the `composition` as the unit of implementation, which selects:

Life-cycle category as in table 2.3;

Catalogs of persistent storage locations for later referencing within *abstract storage home bindings* (optional);

Executors as servants for home and component implementations, including the allocations between them;

Segments within executors for encapsulation of independent state and activation capability of (groups of) *facets* defined before in IDL;

Delegation of operations of home or component executors (per segment) to *storage homes* or *storages*, respectively, which either contain traditional

[20] Cf. definition 2.7 on page 16 and section 2.5.1 on page 63.

hand-written code for object-relational mapping (*self-managed persistence, SMP*) or generated code (*container-managed persistence, CMP*) according to the persistent state declaration code within the CIDL definition; and

Proxy homes for increased scalability, e.g., by installing remote servants with caching functionality (albeit the specification is a bit vague at this point).

CIF

The *Component Implementation Framework (CIF)* provides the architectural framework behind the syntax and semantics of the CIDL. There already exists a UML Profile[21] [OMG04c] that maps component specifications to CIF elements for automatic (CIDL) code generation. The major goal of this additional step in component development is to further minimize the necessary artifacts the application programmer has to implement for a component.

EJB mapping

The CORBA Components specification also provides a mapping to and from the Enterprise JavaBeans component model, including definitions for runtime interoperability in both directions. *Bridges* allow for transparent encapsulation of interfaces for clients against each other. Thus, CCM clients see a CCM view of EJB components and EJB Clients an EJB view of CCM components, respectively. The server-side component-container contracts are however far more different from each other as to allow an easy interoperability at this level, too. Hence, there is currently no way to deploy EJB components to a CCM container or CCM components to EJB containers, respectively.

Descriptors and Deployment

Just as Enterprise JavaBeans, CCM follows the concept of *descriptive middleware*, i.e., the configuration of component behavior can be adapted to a specific purpose and/or runtime environment by means of additional descriptors that are evaluated by the container at deployment time. CCM features a much richer collection of descriptors than EJB. An overview is given in figure 2.7 on the following page.

Component Software Descriptors (CSD) are XML files describing physical implementations as `softpkg` elements, including external dependencies to other subsystems, libraries, or additional components, as well as details of the target platform, like processor, operating system, etc.

CORBA Component Descriptors (CCD) are XML files initially generated from CIDL composition declarations. It primarily contains redundant—for avoiding access to the interface repository—descriptions of the component ports as well as life-cycle category and facet segmentation. Developers further have to specify non-functional properties, like transactional properties of operations[22], semantics of event handling, security in

[21] Cf. section 2.5.1 on page 61.

[22] Transaction demarcation—especially *container-managed transactions*—closely resembles the concepts of EJB as shown in table 2.2 on page 28.

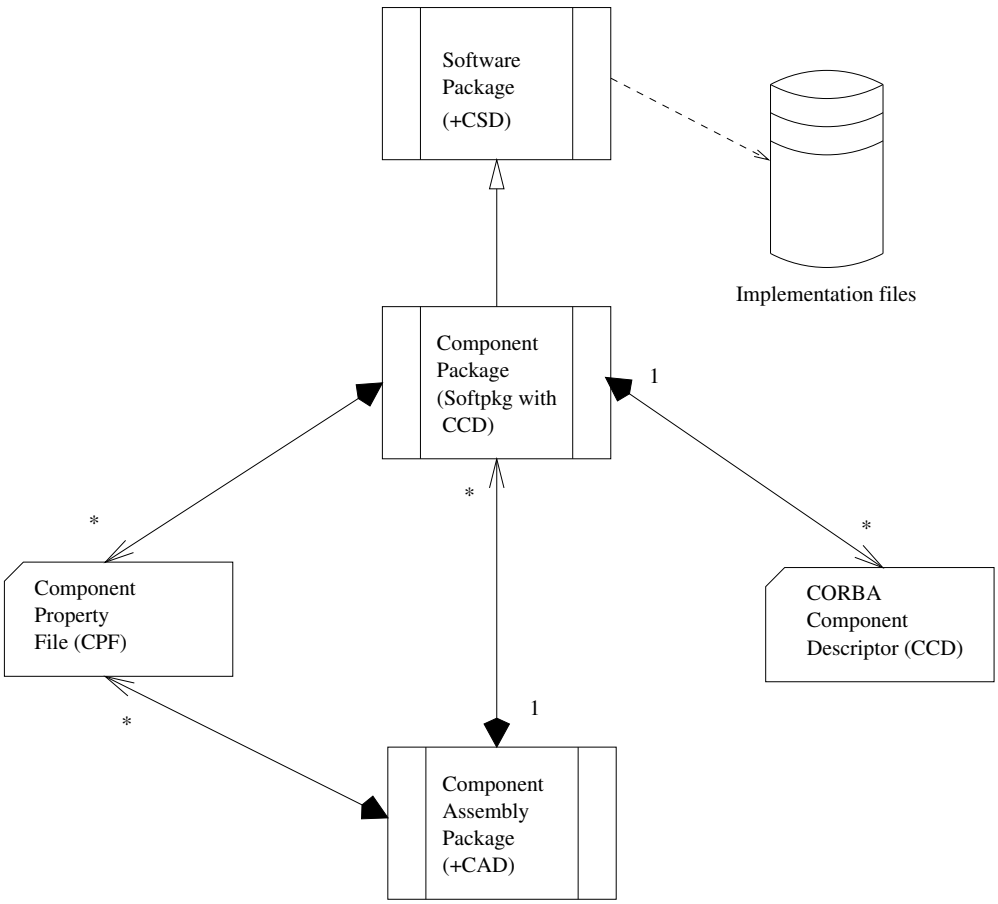

Fig. 2.7: Relationships between component descriptors and packages

terms of role-based access control, reentrance (capability for concurrent access to code segments) etc.

Component Property Files (CPF) are XML files describing pre-configurations of component attributes.

Component Assembly Descriptors (CAD) are XML files that comprise elements describing the components used in the assembly, connection information, and partitioning information (i.e., collocation of components on specific hosts and processes). Instantiated components can be connected by their facets and receptacles or event sources and sinks, respectively.

Software Packages are ZIP archive files grouping a *Component Software Descriptor* with a set of binary implementation files (libraries etc.), which may be either contained in the archive or referenced externally.

Component Packages are special *Software Pages* additionally containing *CORBA Component Descriptors* and optionally *Component Property Files*.

Component Assembly Packages bundle multiple *Component Packages* together with a *Component Assembly Descriptor* and optionally a number of *Component Property Files* containing component attribute configurations.

Especially the aspect of assembly shows many parallels to Architecture Description Languages (ADL) [MT00]: Components are connected using their *ports*, i.e., facets, receptacles, and event sources / sinks. Attributes can be used to pre-configure component instances. However, CCM has no explicit concept for *connectors*—the container plays this role implicitly—and no assumptions are made on how component assemblies evolve at runtime. *ADL*

Due to the more intricate constellation of various archive and descriptor types, the process of deployment is also somewhat more complicated in comparison to, for instance, Enterprise JavaBeans. Figure 2.8 depicts the instantiation of various runtime container constructs at deployment time.

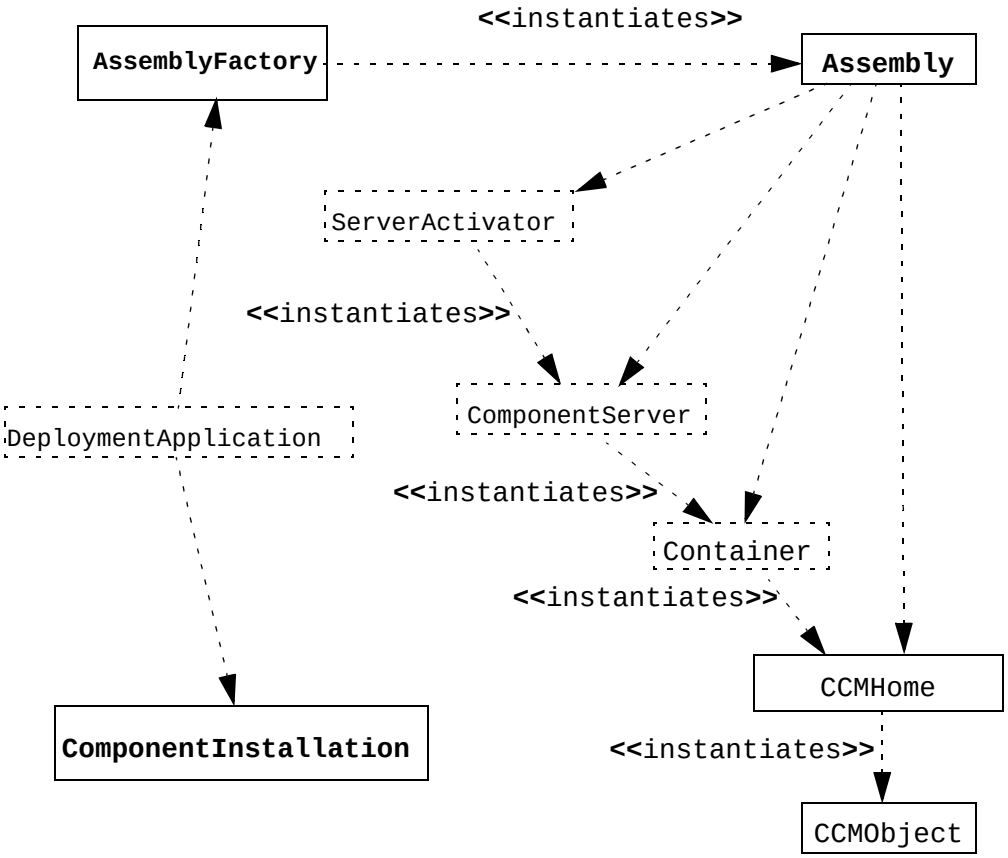

Fig. 2.8: Deployment architecture [OMG02a]

The *Deployment Application* opens an *Assembly* and checks dependencies to pre-installed subsystems using the information in the *Software Package*. It is thus able to determine necessary steps to prepare target hosts using special

agents. In the next step, component servers are started remotely in different processes according to the declaration in the Assembly. This in turn allows to create instances of homes and components and to pre-configure these instances with the help of Properties with the purpose of subsequent announcement to naming and trading services. Finally, connections are established between the component ports as specified in the Assembly.

2.2.6 Microsoft .NET

The Microsoft .NET framework and its ancestor, the *Component Object Model (COM)*, play only a marginal role in the context of this work since we primarily use the Enterprise JavaBeans platform to demonstrate our concepts. An implementation on the .NET platform would however be possible as well. Hence, we will take a closer look at this competing approach and compare its features with those offered by EJB (see section 2.2.4 on page 22) and CCM (see section 2.2.5 on page 29).

COM

The Microsoft *Component Object Model (COM)* [MSc] was first introduced in 1993 for component-oriented programming of local (client-side) applications. Nevertheless, it still provides the fundamental concepts for all of its extensions, which will be subject of the following sections. The concepts of COM originate from the *Object Linking and Embedding (OLE)* technology, which was primarily used for features like "copy and paste" / "drag and drop" between different applications, as well as for managing compound document objects. COM was actually the underlying object model for OLE 2. The basic elements of this object model are shown in figure 2.9.

OLE

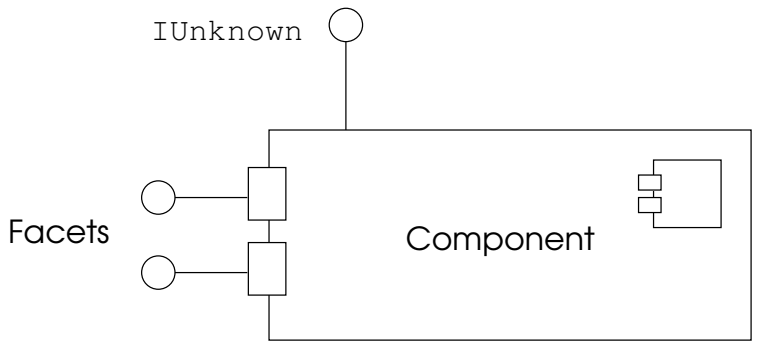

Fig. 2.9: Abstract model of COM

`IUnknown` is the standard interface every COM object has to implement. It comprises three methods:

`AddRef()` and `Release()` for client reference counting, which is necessary to determine if an object can be disposed because it is not referenced anymore, and

`QueryInterface()` for retrieving *Facets*.

Facets are additional, independent interfaces an object may provide to service client requests to its functional services.

Class factory

Object instantiation is typically accomplished by querying a *class factory* from the COM runtime library using the class ID (CLSID) of the desired object. These factories are able to create objects, which a client can in turn query for their facets after it has added itself as a new reference to that object. There is also a host of standard literature available to explain the main features of this model, e.g., [Rog97].

ActiveX

ActiveX [MSa] was introduced in 1996 as a synonym for some parts of OLE relating to Internet integration. This technology is comparable to both, Sun's JavaBeans [Sun02b] specification and Java Applet technology, in that it enables distribution of small client programs over the Internet, integrated on Web sites. However, the lack of security mechanisms for code integrity and access control to system functions posed a major drawback of ActiveX compared to Java Applets, which was only partially compensated after the retroactive integration of code signing mechanisms.

DCOM and MTS

OSF-DCE

The *Distributed Component Object Model (DCOM)* [MSd], originally called "Network OLE" is an extension of COM to allow COM objects to communicate remotely in a reliable, secure, and efficient manner. It is based on the Remote Procedure Call (RPC) specification of the Open Software Foundation's Distributed Computing Environment (OSF-DCE). It was introduced in 1996 as a response to the Object Management Group's Common Request Broker Architecture (CORBA) [OMG04a].

Among a number of new features, DCOM also introduced a concept for custom proxy objects[23], which can be used to inject code on the client side. Apart from load balancing and fault tolerance, this mechanism can also be used for caching data at the client side.

Only one component type

The *Microsoft Transaction Service (MTS)* [MSe] builds on DCOM to combine object request broker functionality with that of a Transaction Processing (TP) monitor. Provided services include transactions, scalability services, connection management, and administration. Hence, it provides a comparable set of middleware services as Sun's Enterprise JavaBeans technology and the OMG's CORBA Components Model. Unlike its competitors, MTS supports only one

[23] Cf. section 2.5.5 on page 71.

component type, which can be compared to EJB Session Beans (cf. table 2.1 on page 25) and CCM Process components (cf. table 2.3 on page 34). The concept of *automatic transactions* allows the demarcation of transaction attributes similar to EJB and CCM, but only coarser grained at component level (cf. table 2.2 on page 28).

A deeper insight to programming with MTS is given, e.g., in Gray's book [GJL97].

COM+

Alignment of COM and DCOM

Common services

Interception

In 1997, Microsoft introduced *COM+* [MSb] as a further extension of COM with the goal to re-align COM and DCOM as a single product family. The most prominent enhancement has been the introduction of a set of common services, such as transactions, role-based security at method-level, monitoring and administration facilities, message queuing and event services, and load balancing / clustering, among others. Interoperability between components written in different languages was also improved. A *context* interface and *attributes* were added to the component interface. The concept of *interception* (cf. section 2.5.5 on page 71) was added to enable simple architecture extensions. The programming model was simplified to allow designing components as simple as single-user, single-threaded applications and to let the runtime environment take care of concurrency and distribution issues automatically. In that respect, the concept COM+ closely resembles the competing approaches of CCM and EJB, especially the *resource pooling* mechanisms as an extension of MTS *resource dispensers*.

.NET

Microsoft .NET [MSf] was released in 2002 as a framework for components that facilitate integration by sharing data and functionality over a network. It features a client runtime environment as well as a number of server implementations for various middleware tasks. It is furthermore accompanied by a set of development tools for building component-oriented applications. In many ways, it parallels Sun's Java 2 Enterprise Edition (J2EE).

CLR

While its predecessors (COM etc.) where mainly focused on the Microsoft (Windows) platform .NET tries to make up for former disadvantages by providing neutrality from both, platforms and languages. The basic building block of the framework is the *Common Language Runtime (CLR)* as depicted in figure 2.10 on the next page. The CLR can be implemented on practically any platform[24]. Although the Java-like C# [MSg] is the preferred language for programming .NET applications, the CLR supports over 20 different programming languages, which also poses a major benefit in comparison to its prede-

[24] Currently, there even exist rudimentary CLR implementations for open-source operating systems like Linux. However, most integrated COM+ services are still platform-dependent.

cessors (COM etc.) as well as its competitors (EJB, CCM). Language neutrality is provided by means of the *Microsoft Intermediate Language (MSIL)*, a machine-independent low-level language comparable to Java byte-code, to which respective compilers have to translate high-level code. Similar to the Java approach, a Just-In-Time (JIT) compiler translates MSIL fragments into managed native code for execution within the runtime engine. A *Common Type System (CTS)* similar to CORBA IDL [OMG04a] or the Meta Object Facility [OMG03a] ensures cross-language type-safety.

MSIL

CTS

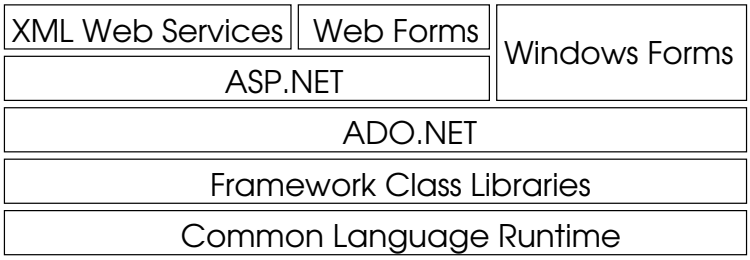

Fig. 2.10: Elements of the .NET framework

Apart from language integration, the CLR caters for security and resource management (memory, processes, threads, etc.) as well as life-cycle management of components, among other functions. On top of the CLR reside the *Framework Class Libraries (FCL)*—a set of base libraries for standard input/output, string manipulation, security, networking, multi-threading, and graphical user interfaces. *ADO.NET* comprises XML and Database access libraries. *ASP.NET* supports the development of Web applications (user front-ends) and Web services (back-end integration of third-party services). It can thus be compared to Java ServerPages (JSP) and Servlet technologies in J2EE. *Windows Forms* allow for easy programming of desktop applications. They are intended as an alternative to ActiveX controls.

FCL

Backward-compatibility to COM is provided by means of *Runtime Callable Wrappers (RCW)* and *COM Callable Wrappers (CCW)*, which have to be implemented by COM / .NET components to become accessible by the other technology, respectively.

Parts of the .NET Framework have been submitted to the ECMA for international standardization by the name of *"Common Language Infrastructure" (CLI)*, i.e., the virtual machine (CLR) and class library (FCL) behind the .NET architecture.

CLI

A more in-depth introduction to programming applications with the .NET framework is given, e.g., in Prosise's book [Pro02].

2.2.7 Web Services

Web services [BHM+04] are a technology for integrating heterogeneous applications in a very loosely coupled manner. They provide interoperability

between various platforms because of their neutrality from implementation languages and executing platforms. The related protocols are based on XML, making it easy for developers to debug applications because of the human-readable nature of exchanged messages. Protocol data is typically shipped over HTTP, which allows easy contacting of globally distributed services regardless of most firewall security measures that are usually a challenge for other middleware platforms. In the context of *multi-tiered architectures* (see section 2.2.3 on page 21), they can be used as technology for *Enterprise Application Integration (EAI)*.

EAI

Interestingly, the relation of services and components has been realized by Cheesman and Daniels before the advent of web services [CD00, p. 8]:

> A component isn't a service, although one of the things you can do with components is build *Service-Based Architectures*, where each Component Object provides a specific function, using specific data.

Another concept that is often discussed lately in connection with web service technology are *Service-Oriented Architectures (SOA)* [BHM+04, Sect. 3.1], or service-based architectures. They denote an architectural style characterized by loose coupling between interacting software agents. As illustrated in figure 2.11, services are units of work performed by *service providers* for *service requesters*, both acting as agents on behalf of their owners. Loose coupling is achieved by a simple set of generic interfaces through which extensible, descriptive messages are exchanged.

Service-Oriented Architectures

It takes no wonder that web service technology has been gradually integrated into the component architectures introduced above. Sun's EJB features support for web services since v2.1 [DeM03], and web services are an integral part of J2EE [Sha03]. Microsoft's .NET platform [MSf] also heavily relies on web services as a technology for integrating external services.

Web service technology builds on a number of standards and specifications, which are shown in their relation to each other in figure 2.11.

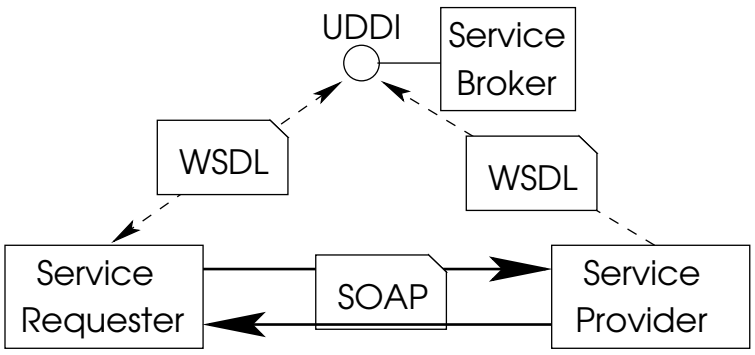

Fig. 2.11: Web service technology overview

Simple Object Access Protocol (SOAP) [Mit03] is a light-weight application level protocol for message exchange between object-oriented (and typically component-oriented) systems. It basically defines an XML-based format for request/response messages between systems. Remote procedure calls (RPC) can be emulated with SOAP. It is typically run on top of HTTP [FGM+99], although it was designed to run on top of virtually any Internet protocol. SOAP messages have a header for meta-information about transactions, security, etc. and a body consisting of an *envelope* that contains the payload information. Besides SOAP, XML-RPC can also be used as a more lightweight protocol for invoking web services.

Web Service Description Language (WSDL) [CCMW01] is an XML-based format for describing the public interface of web services, i.e., an abstract description of supported operations and message types as well as concrete protocol bindings and message formats.

Universal Description, Discovery, and Integration (UDDI) [JM02] is an XML-based registry interface for global businesses. A UDDI business registration consists of three components:

White pages contain information about address, contact, and known identifiers;

Yellow pages categorize a service based on standard taxonomies; and

Green pages comprise technical information about services exposed by the business

UDDI registries are meant to be queried by *SOAP* messages and to serve *WSDL* documents for the web services they list. Unlike *SOAP* and *WSDL*, the UDDI initiative was not led by the W3C but by OASIS instead. Note that a (central) registry like that proposed by UDDI is not necessarily required for service discovery[25].

Web service framework stack

However, SOAP, WSDL, and UDDI form only the fundamental basis of the larger Web services framework as depicted in figure 2.12 on the next page. This framework consists of three ever-growing stacks of emerging specifications for communication *protocols*, service *descriptions*, and *discovery* facilities.

Discussion and Relevance Apart from the advantages web services provide for the easy integration of distributed services, there is also a number drawbacks:

- Performance is usually slower by magnitudes in comparison to other middleware platforms.

[25] For instance, *peer-to-peer (P2P)* approaches would also be feasible. The whole area of P2P systems would however lead out of this work's scope.

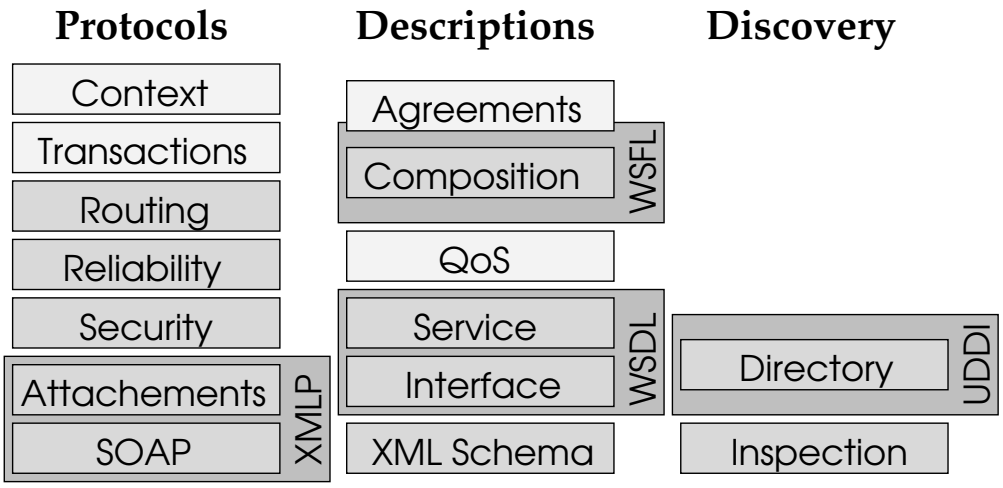

Fig. 2.12: Web service framework stack according to [RW02]
Legend: QoS—Quality of Service; SOAP—Simple Object Access Protocol; UDDI—Univeral Description, Discovery, and Integration; WSDL—Web Service Description Language; WSFL—Web Service Flow Language (composition language proposed by IBM); XMLP—XML Protocols (W3C Working Group)

- Circumventing firewalls by tunneling all traffic over HTTP introduces new security issues that are much harder to audit.
- Support for more sophisticated middleware features like security or transactions is currently still missing or under development.

For a more detailed introduction to web service technology, the interested reader may consider one of the huge number of books and articles available about this topic, e.g., Cerami's book [Cer02].

Web services vs. distributed objects

In the context of this work, web services play only a marginal role. They just provide yet another access protocol to (potentially) component-based services. This opinion is also shared by Birman [Bir04], who argued in contrast to Vogels [Vog03] that Web services are indeed essentially just another access technology for distributed objects. Vogels [Vog03] complained about the document-centered nature of Web services, which ultimately restricts their computing model, rendering the concept inappropriate for general use. But although Web service protocols and concepts evolved in an *ad hoc* fashion to overcome platform interoperability issues by loose coupling using standard protocols, their rapidly growing popularity, platform-independence, and ubiquitous use will accelerate the addition of missing middleware functionality like reliability/availability.

It would pose no major technical obstacles to implement the proposed approach for method-based caching on top of this technology and accessing

clients would also benefit from this enhancement. However, supporting this additional access technology would bear only little scientific value.

2.2.8 Summary

Section 2.2 comprised a rather long discussion of current state-of-the-art component-oriented middleware platforms, namely Sun's Enterprise JavaBeans, the OMG's CORBA Components Model, and Microsoft's .NET (section 2.2.4 on page 22, section 2.2.5 on page 29, and section 2.2.6 on page 38). An in-depth comparison of all features has been provided elsewhere (e.g., in [Poh99]) and would go beyond the scope of this work.

However, a few similarities and common architectural features shall be summarized here to emphasize the general applicability of the proposed concepts in section 4 on page 107, although the demonstration of these concepts has only been implemented for the EJB platform.

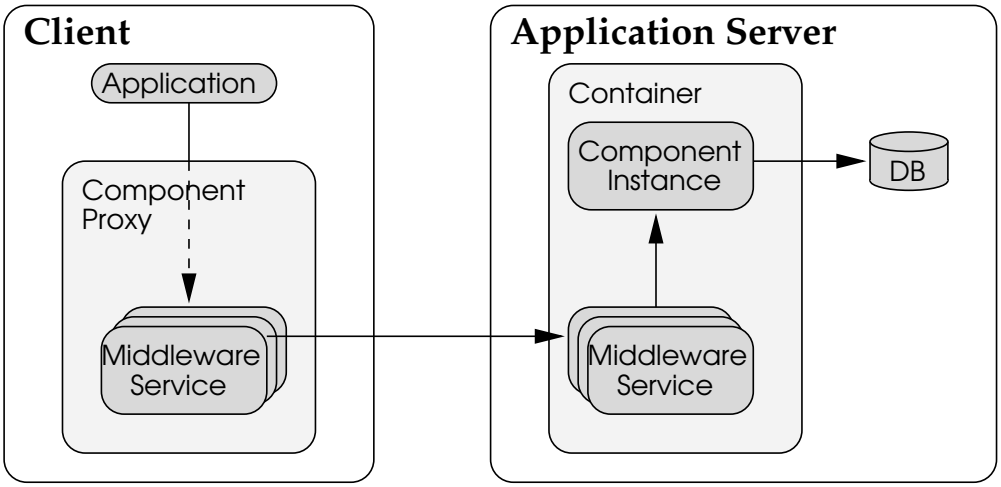

Fig. 2.13: General architecture of component-oriented middleware

The general architecture of the component-oriented middleware platforms introduced above is depicted in figure 2.13: Client applications access (potentially distributed) components through some kind of proxy/stub object that provides distribution transparency as well as hooks for arbitrary middleware services. The proxy ultimately transmits invocations to a (potentially remote) application server where the actual component instance resides in a runtime environment, which we will shortly refer to as *container*. In analogy to the client-side proxy, the container in turn provides hooks for middleware services and takes care of life-cycle functionality etc. before it finally invokes the component instance to perform the requested service. The component instance may in turn invoke additional components (not shown in the figure) or query a database directly or indirectly (via the container). Invocation results are then transferred back to the invoking client application in reverse order.

It is furthermore important to note that definition 2.8 applies to all of the example platforms presented above. In other words, component state is always accessed by means of accessor/mutator methods. Hence, caching component state based method results is feasible for all major platforms.

2.3 Distributed Data Management

In section 2.2 on page 14 the *business tier* (cf. figure 2.2 on page 21) was put into the center of interest, elaborating about persistence and transactionality, among other middleware services provided by application servers. However in this respect, application servers only act as mediators to the *database tier*, comparable to TP monitors [MW88]. Databases may in turn be replicated and distributed, which opens up a completely different research area—the field of distributed databases [BHG87, Rah94, Len97, ÖV99]. However, a number of fundamental concepts from this field are important when talking about transactionality of cached access to components in section 4 on page 107, since this can be seen as a special case of partial replication. Hence, we will give a brief overview of this subject.

Distributed databases

2.3.1 Transactions and Concurrency Control

The concept of *transactions* is known in the database community since several decades as a concept for programs operating on a shared resource (e.g., a database) without interfering with each other. Härder and Reuther summarized the most important properties of transactions under the acronym *ACID* [HR83]:

ACID properties

Atomicity requires a transaction to be processed in an "all-or-nothing" manner, i.e., the whole transaction program has to succeed, including each individual operation. Otherwise it has to be rolled back (*undo*), i.e., its effects have to be made invisible.

Consistency of the database has to be preserved across transactions, i.e., transactions may commit only legal results.

Isolation preserves the illusion of single-user-operation for concurrently running transactions, i.e., the events within transactions are hidden from other transactions until they are committed.

Durability has to be guaranteed for results of successfully completed transactions by the database management system, i.e., the state must be reconstructible after a sudden system crash (*redo*). Once completed transactions can only be undone by corresponding compensatory transactions.

These properties are also influencing each other, e.g., isolation is a prerequisite for atomicity; consistency is required for durability. A database man-

agement system (DBMS) has to implement a number of concepts to ensure the adherence of ACID properties:

Recovery caters for atomicity and durability by ensuring that aborted transactions can be rolled back (*undo*) and committed transactions can be replayed (*redo*) after system crashes. An important technique to achieve recoverability is *logging*: The recovery system has to log redo and undo information continuously about every transaction, either logical in terms of data manipulation operations (i.e., the "delta") or physically in terms of before- and after-images. A number of books discuss the details of logging and recovery, e.g., [BHG87]. *Logging*

Concurrency Control is a form of *synchronization* that is especially required for keeping up the isolation property, but also for consistency. For obvious performance reasons DBMS concurrently serve multiple transactions. *Serializability* theory deals with the synthesis of *serializable histories*, i.e., non-conflicting execution plans of concurrent transactions. *Serializability*

> An execution is *serializable* if it produces the same output and has the same effect on the database as some serial execution of the same transactions. [BHG87]

The transaction manager of a DBMS achieves serialization of concurrent transactions by using *schedulers* that execute, reject, or reorder operations of transactions by delaying them. The goal of schedulers is to ensure the serial execution of transactions accessing the same data by *mutual exclusion*. A basic distinction can be drawn between *locking and non-locking schedulers*, for both of which we will discuss a number of general strategies a few paragraphs later. *Scheduler*

Without concurrency control, a number of *anomalies* may occur due to interfering read/write operations. Only serializable executions can prevent all of these phenomena[26]. *Anomalies*

Lost Update phenomena occur when two transactions read the old value of the same data object before writing it both in an arbitrary sequence.

Dirty Read is also referred to as "read uncommitted data", i.e., data that is not fixed and might be rolled back later.

Inconsistent Retrieval is a more subtle type of interference that occurs when a transaction reads one data item before it is updated by another transaction and another data item after the same update transaction has updated it. That is why this anomaly is also called "non-repeatable read".

Phantoms are data items that are inserted or deleted while another transaction is operating on a set of data items to whose predicate the "phantoms" would match. However, they are either not considered anymore

[26] Weaker degrees of consistency are described, e.g., by Gray et al. [GLP75].

or even falsely considered further on, since they were either not yet present or not present anymore at the begin of the second transaction.

Concurrency Control by Locking

From the historical perspective, locking schedulers have been the first solution for solving the problem of mutually exclusive data access of concurrent transactions. Each data item has an associated lock, which must be acquired as a kind of token before access to that data item is granted. As an optimization, a distinction is made between *shared* (or "read") locks and *exclusive* (or "write") locks: Read operations may concurrently access the same data items without interference but write operations need mutually exclusive access[27].

Read and Write locks

Another important issue for the performance and scalability of a DBMS is the granularity of locks. Typical granularity levels for locking are database, area, file, record, or for relational databases [Cod70] table, row, and column. Coarse grained locking benefits from low overhead for locking itself but it reduces throughput due to the increased likeliness for conflicts of operations. So the tradeoff for more fine grained locking is overhead vs. concurrency.

Locking granularity

The most common protocol to accomplish locking of data items in the context of transactions is *Two-Phase-Locking (2PL)*. In the *growing phase*, the scheduler acquires locks for accessed data items as a given transaction proceeds. After the *shrinking phase* has begun, locks may only be released but not anymore acquired. This does not prevent *dirty read* anomalies since unlocked written data items may still have to be rolled back. *Strict 2PL* was introduced for this reason: Instead of a shrinking phase, a *shrinking point* is defined where all locks have to be released at once.

2PL

A major problem of locking schedulers are race conditions for data items that may lead to deadlocks and starvation of transactions. Timeouts for locks are the simplest solution but also the one with the most undesired side-effects, e.g., unfair abortions. More sophisticated schedulers use detection algorithms.

Deadlocks

Optimistic Concurrency Control

The first alternative for non-locking schedulers is optimistic concurrency control. Also known under the name *certifiers* [BHG87, Sect. 4.4], this concept is based on the assumption of a low probability of conflicting transactions and follows a three step scheme:

Certifiers

1. *Read.* The transaction may read data items as normal and write to its private "sandbox".

[27] Bernstein et al. [BHG87] furthermore discuss the locking of additional "increment" and "decrement" operations, which shall be omitted here for brevity.

2. *Validation.* After commit the scheduler checks the contents of the transactions sandbox for possible conflicts, aborting it retroactively if necessary.

3. *Write.* The sandbox content is written to the central database if no conflicts were detected.

Härder [Här84] discusses two implementation variants: Backward and Forward Oriented Concurrency Control (BOCC/FOCC). Both variants require the specification of read set $RS(T_i)$ and write set $WS(T_i)$, i.e., the set of data items read and written by a transaction T_i with $WS(T_i) \subset RS(T_i)$, prior to validation.

BOCC considers all transactions that have been committed during execution of the validated transaction. If $RS(T_i) \cap WS(T_j) \neq \emptyset$ for any T_j that has committed during execution of T_i, T_i has to be rolled back. This imposes the danger of starvation if the validated transaction is continuously rolled back. BOCC furthermore suffers from bad scalability due to the need to keep write sets of all committed transactions.

FOCC considers all transactions that are currently running at the point of validation for potential conflicts. If $WS(T_i) \cap RS(T_j) \neq \emptyset$ for any T_j that is active during validation of T_i, T_j or T_i have to be rolled back. This leaves greater flexibility for selecting transactions to abort, which comes at the cost of required locking of write sets during validation.

Timestamp-based Concurrency Control

Another non-locking scheduling mechanism is based on ordering of timestamps, which are assigned in a globally unique fashion to transactions $TS(T)$ as well as their contained operations in form of read timestamps $RTS(x)$ and write timestamps $WTS(x)$ of data items x. Every transaction is required to know all changes made by other transactions that have begun earlier, but it must not see any changes of younger transactions. Thus, a transaction T is rolled back if it tries to read a data item x that has been changed by a younger transaction, i.e., $TS(T) < WTS(x)$, or if it tries to write a data item y that a younger transaction has already read or written, i.e., $TS(T) < max\{RTS(y),WTS(y)\}$. Hence, the probability of aborting increases with the time a transaction remains active in the system, which in turn increases the probability of starvation. This *Basic Timestamp Ordering (BTO)* strategy is still prone to *dirty reads*, a limitation that is remedied by *Strict Timestamp Ordering*, which is explained, e.g., in [BHG87, Sect. 4.2]. *BTO*

Multiversion Concurrency Control

Multiversion Concurrency Control (MCC) [BHG87, Chap. 5] is another non-locking concurrency control method that is also based on timestamps to *MCC*

achieve serializability. It assures that a read-only transaction T always instantly gets a consistent, yet potentially slightly out-dated view of its required data objects x, corresponding to its begin timestamp (BTS), i.e., $WTS(x) < BTS(T)$. Read and write transactions have to be declared as such in advance, otherwise this scheduling method degrades to the scheduling used for write transactions. The original variant of MCC imposes the same rules for write accesses as BTO, i.e., a transaction has to be rolled back if it tries to write a data item that a younger transaction has already read or written.

However, other scheduling methods can also be employed for timestamp ordering of write transactions. For instance, Schaller [Sch03] introduces a combination of MCC and FOCC for front-end caches of relational DBMS, which uses FOCC for scheduling write transactions.

2.3.2 Distributed Databases

Distributed DBMS use computer networking technology to connected parts or instances of databases, usually with the goal of improved performance, scalability, and availability, while preserving distribution transparency, autonomy of participating databases, and consistency of the global data set. Bernstein et al. [BHG87] simply define a *distributed database system* as a collection of centralized database system nodes connected by a communication network. Every data item is stored at exactly one node. Systems that store the same data at multiple nodes are called *replicated database systems*.

Replicated databases

In contrast, Rahm [Rah94] introduces a number of classification criteria for distributed database management systems, e.g., allocation of external memory, physical distribution, and type of coupling. Apart from tightly coupled *shared everything* and *shared disk* systems, the most important distinction of *shared nothing* systems is between *integrated* and *federated database systems*:

Integrated Database Systems comprise identical nodes accessing a mutual database, either on a shared disk or physically distributed. They provide full distribution transparency for clients but low autonomy of participating nodes. Since all nodes have the same conceptual database schema, this case is also referred to as *homogeneous* distribution or *replicated databases*.

Federated Database Systems provide a higher level of autonomy by assigning a separate conceptual schema to each participating node. Cooperating databases in a federation may access remote data if its owner permits access. However, increased autonomy comes at the cost of decreased distribution transparency because of diverging schemas in the case of *heterogeneous* distribution.

A good overview of problems, concepts, and solutions for this subset of distributed database systems is given, e.g., by Rahm [Rah94], Lenz [Len97]

and Özsu/Valduriez [ÖV99]. Bernstein et al. [BHG87, Chap. 8] briefly discuss the issues of replicated data.

However, the benefits of distribution come at the cost of additional challenges. Connections can break down, messages can get lost, whole nodes may become unavailable. This leads to additional considerations for recovery as well as for concurrency control, the latter of which will be discussed in section 2.3.3 on the following page.

Replication Control

Another challenge is the maintenance of global consistency. Especially homogeneous, replicated databases require solutions to ensure that changes made at one node are reflected globally at all nodes. This includes invalidating and/or updating outdated copies of changed data items. This problem domain is also referred to as *replication control*, i.e., providences to keep replicated copies coherent to and consistent with their originating data sources. For instance, Rahm [Rah94, Chap. 9] and Lenz [Len97, Sect. 3.3] explain a number of techniques for solving this problem.

Coherence

The simplest strategy is called *Read-Once-Write-All (ROWA)*, which assumes that clients read data from any node and write data to all nodes. This achieves consistency easily, although at the cost of expensive write accesses. The requirement for 2PC (see section 2.3.3 on the next page) or similar protocols limits its scalability with the number of participating nodes.

ROWA

Primary-Copy (PC) strategies aim at eliminating these deficiencies by defining one replica as the "primary copy" that has to be changed. This node will notify the other replicas asynchronously "as soon as possible". Different implementation alternatives are discussed in [Rah94, Sect. 9.2] depending on the necessity of locks for read access to replicas and the required degree of consistency.

PC

Voting methods represent another alternative solution that is based on the idea to gather "votes" of replicas for read and write locks. *Majority Consensus* requires the acquisition of an absolute majority of votes for a lock, which ensures that no two transactions may concurrently alter the same data item. However, this strategy implies significant communication costs. Because of this, the optimized *Quorum Consensus* method was introduced, which weights replicas by assigning them a certain number of votes v. A number of votes—the read quorum r—is required from transactions to read a data item, a write quorum w to write one. The constraint $w > v/2$ ensures that only one transaction may write a data item at a time; $r + w > v$ prevents concurrent reading and writing of a data item.

Majority Consensus

Quorum Consensus

For situations with weaker consistency requirements, *Snapshot Replication* may also pose an alternative. Snapshots are materialized views that are stored as separate data items that can be queried by read transactions.

Snapshots

2.3.3 Distributed Concurrency Control

Scheduling transactions spanning multiple database nodes introduces new challenges for concurrency control methods. Traditionally, solutions are sought that behave exactly like centralized one-copy databases. Such equivalent executions are called *one-copy serializable (1SR)* [BHG87]. This requires extensions of local protocols for the distributed case. Locking protocols like 2PL can be used to build distributed schedulers. However, distributed deadlock detection becomes slightly more complicated. Non-locking protocols also require some more attention for the distributed case. Timestamp-based methods like BTO and MCC require global synchronization of distributed clocks at defined interaction points to remain effective. BOCC and FOCC need special consideration of the time span between validation phase and final commit/abort to prevent dirty read phenomena.

1SR

An important protocol for handling distributed commit/abort decisions is *two-phase commit (2PC)*, which is crucial to support distributed recovery as well. As the name suggests, it splits the process of ending a distributed transaction in two phases, which have to be communicated to participants by the initiator:

2PC

1. *Voting Phase.* The initiator sends a `PREPARE` message as a voting request to all participants. Participants reply to this with `READY` or `FAILED` depending on their local commit decision.

2. *Decision Phase.* In case of a unanimous vote for `READY`, the initiator sends a `COMMIT` message in turn to all participants, who then act accordingly and stop. Otherwise, the initiator sends an `ABORT` message to all participants that voted with `READY` and then stops.

2PC is resilient to both site failures and communication failures, however it is prone to blocking if transactions time out in their uncertainty period. More sophisticated variants of three-phase commit (3PC) have been proposed as a remedy, but they are more complicated to implement which is why they are rarely deployed. More details can be found in [BHG87, Chap. 7].

Convergence and weak consistency

The strict 1SR constraint may also be softened by only requiring *replication convergence* [Len97, p. 74], i.e., mutual consistency of replicas will eventually be reached without having to roll back committed changes. This leads to the application-specific mitigation of consistency to increase efficiency. A comprehensible discussion of related approaches would however go beyond the scope of this work. A few examples will be handled in section 3.1.3 on page 87 in the context of database caching. More details can be found in [Len97, Sect. 4].

2.3.4 Summary

We have shown in this section that solutions for managing distributed data sources in a consistent manner have been researched comprehensively by the distributed database community over the past decades. Today's off-the-shelf application servers (see section 2.2 on page 14) actively use only a small fraction of this functionality, e.g., the 2PC protocol for coordinating transactions in which multiple distributed data sources are involved. However, many concepts from the replicated database domain are also applicable for distributed caches since client-side caches behave like replicas of a *Primary Copy* (cf. section 2.3.2 on page 51)—the server. We will thus refer to these concepts later as needed.

2.4 Adaptive Systems

The title of this work implies a close relation to the area of adaptive systems. We will thus elaborate briefly on the state of the art in this field.

Adaptation, from Latin *adaptare, adaptio*, denotes the act or process of adapting or fitting, the state of being adapted or fitted, or the result of adapting. Springer [Spr04, Sect. 1.4] described the process of adaptation as depicted in figure 2.14 and definition 2.11:

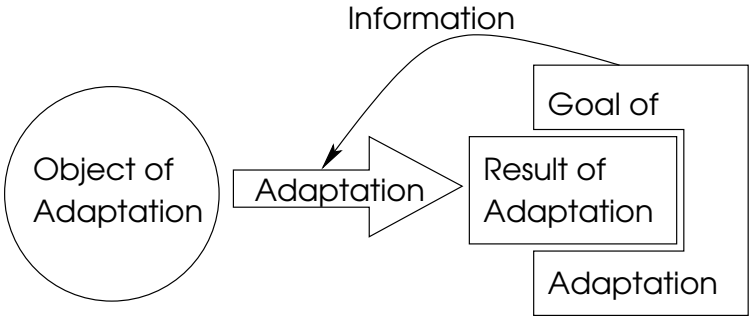

Fig. 2.14: The process of adaptation according to [Spr04]

Definition 2.11 (inspired by [Spr04, Sect. 1.4])
Adaptation is the activity of adapting a certain object of adaptation *to a* goal of adaptation*. The outcome of this process is the* result of adaptation*; information about the difference between goal and result is used again as feedback to the adaptation operation itself.*

The process of adaptation thus forms a *closed control loop*, which is discussed in section 2.4.1 on the next page. Definition 2.11 captures *what* is adapted (the object) and *why* (the goal). What is omitted is *who* (the subject, i.e., an administrator, the runtime environment, or the object itself) and *when* (at the time of development, compilation, deployment, runtime, maintenance, or even continuously).

Closed control loop

2.4.1 Control Theory

Classical control theory basically distinguishes *open* and *closed* control loops:

Open-loop controllers have no direct connection from the system's output to its input. This is the reason for its high sensitivity to the dynamics of the controlled system, which poses its major disadvantage.

Closed-loop controllers or *feedback controllers* were thus introduced in control theory to avoid these problems. Figure 2.15 depicts a simple feedback control loop.

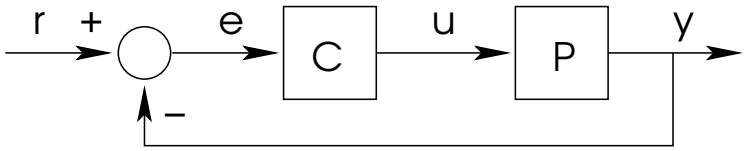

Fig. 2.15: A simple feedback control loop
Legend: r—reference value; e—error, i.e., difference between reference and output; C—controller; u—input/update; P—process/plant under control; y—output

The output $Y(s)$ of a feedback control loop is thus given as:

$$Y(s) = \left(\frac{PC}{1 + PC} \right) R(s)$$

where $PC/(1 + PC)$ is called the *transfer function* of the system. It is obvious that

$$PC \gg 1 \;\rightarrow\; Y(s) \approx R(s)$$

Stability, Controllability, Observability

which leads to *stability* of control, i.e., output will be bounded for any bounded input over any amount of time. Closely related to stability are *controllability*, i.e., the reaction of a system's internal state to external input, and *observability*, i.e., how well the internal state of a system can be derived from its external output.

Hence, the crucial part about judging a system's control properties is knowing its transfer function $Y(s)$. For adaptive software in general as discussed in the following section or for software *components* in particular, it is however often hard to precisely give this transfer function because not all parameters are always known.

2.4.2 Adaptive Software

In the following, we will concentrate on adaptation, adaptability, and adaptivity in the context of software systems in general and component-based software in particular. As a refinement of definition 2.11 on the page before, Lieberherr [Lie96] emphasizes context as the cause of adaptation:

Definition 2.12 ([Lie96, p. 1])
Adaptive object-oriented software *is software that adapts automatically to changing contexts.* Contexts *may be behavior, implementation class structures, synchronization structures, object migration structures, etc.*

Czarnecki and Eisenecker [CE00] further distinguish *adaptable* and *adaptive systems* by differentiating between the ability to be adapted from the ability to adapt itself:

Definition 2.13 ([CE00, p. 397])
Adaptable systems *can be adapted to a particular deployment environment, whereas* adaptive systems *adapt themselves to a deployment environment.*

Adaptability vs. Adaptivity

We will consequentially use *adaptability* to refer to the ability to be adapted and *adaptivity* for the ability of a subject to adapt itself to a given goal of adaptation. The term *self-adaptation* is also often used to refer to this aspect of reflexivity.

In [PG03], we have presented a general classification scheme of issues in connection with adaption as depicted in table 2.4.

Tab. 2.4: Classification of adaptational issues according to [PG03]

Classification	*Adaptation*	*Description*	
Who?	User-driven	administrator	
(Trigger)	System-driven	container	
Where? /	Parametric	within component	
How?	Structural	within composition / application	
Why?	System-side	react to preemption of resources	
	Application-side	react to user requirements	
When?	Static	deploy-time	
	Dynamic	run-time	upon initialization
			during connection
How much?	(Overhead)	additionally required time & resources	

Springer [Spr04] characterizes *adaptive applications*, which we consider synonymous to *adaptive software*, by their ability to dynamically adapt application data, properties of communication, and application structure as a reaction to changes of their environmental context at runtime. He classifies base *mechanisms for adaptation* with focus on context-aware ubiquitous applications as shown in table 2.5 on the next page, which actually represents a refinement of the second criteria of our classification in table 2.4.

The *Proxy* pattern (see section 2.5.5 on page 72)—a prerequisite for the extensions we will introduce in section 4 on page 107—is explicitly mentioned as a structural mechanism for adapting connections between components. Both *Caching*[28] and *Prefetching* (see section 3.3 on page 100) are characterized as

[28] See section 2.1 on page 9 and section 3.1 on page 81.

Tab. 2.5: Base mechanisms for adaptation according to [Spr04]

Structural adaptation		Placement	Migration
			Place of creation
		Connection	Dynamic binding
			Indirection / *Proxy*
			Branch
			Parallelization
			Joining
		Components	Adding
			Removing
			Replication
Parametric adaptation	Communication	Transfer	Protocol parameter
			Exception handling
			Data priority
		Buffering	Queuing
			Logging
			Caching
		Access	*Prefetching*
			Lazy loading
			Delayed write-back
	Application data	Enrichment	Aggregation
			Annotation
		Transformation	Structural transformation
			Format transformation
			Transcoding
		Replacement	Generation
			Extraction
			Selection
		Reduction	Lossy conversion
			Filtering

mechanisms for adapting the parameters of communication between components. Caching buffers queried data in a temporary memory; prefetching shifts the point of data transfer to an earlier time before actual access.

Self-managed Systems

Adaptive systems in general received a lot of attention in the recent past driven by the desire to reduce the complexity and maintenance effort of today's (software) systems. Other causes for adaptation include heterogeneity of operation environments and corresponding availability of resources, user-required quality of service (QoS), as well as malfunctions, failures, and breakdowns. The broad bandwidth of adaptive software comprises online-learning algorithms and genetic code, self-managing systems (see below), adaptation to heterogeneous and mobile application environments, personalized and adaptive user

interfaces, among others. Klamar [Kla04] provides a good overview of current trends in the area of adaptive systems. For instance, the field of *Autonomic Computing* [KC03] created the keywords self-management and so-called "self-x-properties" or *CHOP* properties of systems:

Autonomic Computing, Self-X, and CHOP properties

Self-configuration addresses the autonomous adaptation to changing environment conditions in terms of new servers, software versions etc.

Self-healing systems are able to detect malfunctioning components and to react appropriately by isolating affected components, repairing or replacing them and finally reintegrating them into the system. This requires a minimum of redundancy to maintain uninterrupted operation and user-perceived transparency. Garlan *et al.* [GKW02] collected a representative profile of approaches to self-healing.

Self-optimization targets continuously self-improving systems that try to increase the efficiency in terms of performance and cost of their operation by learning their operating parameters and available resources from experiments.

Self-protecting systems try to shield themselves from malicious intrusion and data corruption. Their capabilities go beyond those of self-healing systems in such a way as they anticipate potential issues by continuously monitoring reports of sensor data.

The challenges for developing *adaptive* software already start with the design and implementation of *adaptable* components. Model-driven approaches (see section 2.5.2 on page 65), e.g., [GS02], and Architecture Description Languages (ADL) [MT00] provide promising approaches for tackling these issues. However, support for adaptation either has to be weaved into components themselves (cf. section 2.5.6 on page 75) or it has to be provided at runtime by the platform on which components are executed—the middleware (see section 2.2 on page 14). This support is crucial for making composed adaptable systems adaptive. To be able to adapt the structure and behavior of a system, the middleware also needs support for reflection (see section 2.5.5 on page 71), i.e., some sort of meta-programming mechanism.

Necessary steps for adaptation at runtime, which have to be performed by adaptive systems, have been summarized in [GKW02] as a superset of Oreizy *et al.* [OGT+99] and Garlan/Schmerl [GS02]. These steps include: *Monitoring* the system and measuring of sensor data; *resolving* measured data and interpreting it at architectural level; *detecting* the necessity or possibility of adaptation; *planning* a resolution in terms of adaptation steps; *deploying* the description of this resolution; and finally *enacting* the adaptation in the runtime environment. Table 2.6 on the following page relates the different terminologies to each other.

Fractal [DL03] is an example for "self-adaptive" component-based middleware: Adaptation to specific execution contexts and their evolution is factored out as a concern, which is handled by the middleware. A component

Tab. 2.6: Adaptation steps at runtime

[GS02]	[GKW02]	[OGT+99]
Monitoring	Monitoring	Monitoring
Interpretation	Resolving	Planning
	Detecting	
Resolution	Planning	
Adaptation	Deploying	Deploying
	Enacting	Enacting

is made *adaptable* by specifying its expected behavior depending on certain contexts. The middleware in turn makes the component *self-adaptive* by interpreting adaptation profiles, which can be loaded dynamically at runtime. An important challenge is posed by the question of how to adapt to changes of parameters that have not been considered at design time. This leads to the general problem that computers can only process what has been specified or what they have been programmed for. Furthermore, correctness of adaptive systems becomes even more complex to ensure in comparison to traditional, static systems.

2.4.3 Conclusion

In the context of this work, we will concentrate especially on *caching*, which has been classified in table 2.5 on page 56 as a mechanism for parametric adaptation of communication between components by buffering query results. We will present the necessary middleware support for this mechanism in section 4 on page 107 and elaborate on our approach for configuring the use of this service at development and/or deployment time in section 5 on page 131. In section 4.2 on page 111, we will show especially how this middleware service can be made adaptive in terms of changed cacheability categorization of component attributes (i.e., method results) to remain efficient at runtime.

2.5 Modeling and Design Concepts

The purpose of this section is to introduce fundamental concepts of software engineering, which will be needed at various points of this work, especially in section 5 on page 131.

2.5.1 Unified Modeling Language

The Object Management Group's *Unified Modeling Language (UML)* forms the elementary building block for the integration of the concepts of this work into the software development cycle, which is presented in section 5 on page 131.

It is an open modeling and specification language for (object-oriented) software systems. The OMG issued a request for proposal for "Object Analysis & Design" in 1996. As a result of a joint effort of the "the Amigos"—Booch, Jacobson, and Rumbaugh—the version 1.0 of UML was released in 1997, indeed unifying the most common modeling techniques up to this date: the *Booch method* [Boo94], Jacobson's *Object-Oriented Software Engineering (OOSE)* [Jac91], and Rumbaugh's *Object Modeling Technique (OMT)* [RBP+91]. UML has gone through a number of revisions since then and has become a *de facto* industry standard. The latest stable release is version 1.5 [OMG03d]. The next generation—UML 2.0—is currently being finalized. It consists of two parts: *infrastructure* [OMG03b], i.e., the "internals" of UML, and *superstructure* [OMG03c], i.e., the high-level constructs and diagrams that users commonly identify with UML.

History of UML

We will not explicitly use any new features of UML 2.0 in the context of this work but diagrams will conform to this newer version wherever applicable. As Cris Kobryn summarized in [Kob04], the major improvements of UML 2.0 include:

UML 2.0

Composite structures enable support for component-based development using *Parts*, *Ports*, and *Connectors* in black-box and white-box–views.

Hierarchical decomposition of structure (*Classes* and *Components*) and behavior (*Interactions*, *State Machines*, and *Activities*), as well as

Integration of these structural and behavioral constructs across different diagram views.

Action semantics —a new feature since UML 1.5—have been integrated with the above mentioned behavioral constructs, allowing the definition of executable UML models.

Incremental language architecture layers: *Basic*, *Intermediate*, and *Complete* levels organize all UML 2.0 packages allowing, e.g., easier tool compliance testing.

There are three major models within UML as outlined in table 2.7 on the next page together with the accompanying graphical representations. Regarding the extensions proposed in section 5 on page 131, we will especially focus on *class* and *component* diagrams. A more in-depth introduction to UML, including version 2.0, is given in Fowler's "Brief Guide" [Fow03].

Note that models exist independently from these diagrams. A binding from the UML metamodel to the OMG's *Meta-Object Facility (MOF)* [OMG03a]—a technology for modeling and representing metadata in distributed object repository architectures that is also used for CORBA, among others—exists for interoperability with other metadata repositories. Currently, the UML 2.0 infrastructure specification and the MOF 2.0 specification are proactively moving towards an architectural alignment of their metamodel

Meta-Object Facility (MOF)

Tab. 2.7: Models and Diagrams in UML

Model	**Aspect**	**Diagram types**
Functional Model	Functionality	Use Case Diagram
Object Model	Structure	Class Diagram
		Object Diagram
		Component Diagram
		Deployment Diagram
Dynamic Model	Behavior	Sequence Diagram
		Collaboration Diagram
		Activity Diagram
		Statechart Diagram

approaches. The goal of this unification attempt is depicted in figure 2.16 on the next page: MOF forms the common meta-metamodel (also called *M3 level*) for UML, CORBA IDL, and other metamodels (*M2 level*). These can in turn be used to create specific models (*M1 level*), e.g., for a certain application, from which concrete objects (*M0 level*) can be instantiated. To give an example: Customer "John Doe" is an object (M0) of the Customer class (M1), which was modeled in UML (M2). In other words, MOF provides the means to create metamodels.

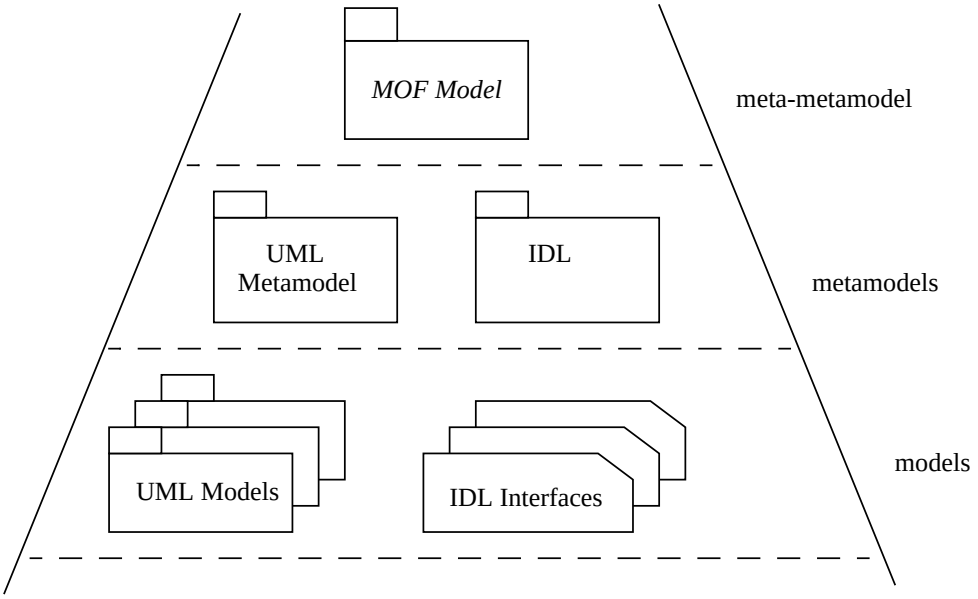

Fig. 2.16: MOF metalevels [OMG03a]

XML Metadata Interchange (XMI)

The *XML Metadata Interchange (XMI)* [OMG03e] format can be used to export UML models (as well as other MOF-compliant models) in an interchangeable form. It allows producing XML schemas or document type definitions (DTD) from object models as well as XML documents from objects. For the tool chain we will propose in section 5 on page 131, XMI plays a crucial role as

the interchange format between modeling tools and model transformers.

Software Design Methods

There is no development methodology dictated by the UML but it forms the basis for at least two software design methods:the *Rational Unified Process (RUP)* [Kru00] and the *Object Engineering Process (OEP)* [Oes02]. Both of these methods are heavyweight processes, in opposition to lightweight processes such as Beck's *Extreme Programming (XP)* [Bec99] or Fowler's *Continuous Integration* [FF03], meaning that the former two are rather suited for large-scale software projects with more than ten developers involved. Details and differences of particular methods are out of this work's scope. It is however important to keep the general procedure in mind, which essentially follows the line of the OEP micro process in figure 2.17: analysis, design, implementation, and validation. This process is often applied in an iterative, incremental fashion throughout the individual project phases during which different actors perform activities belonging to certain disciplines in parallel.

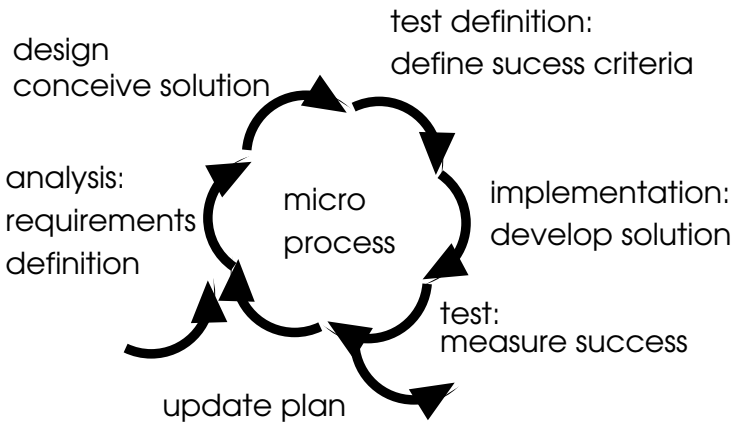

Fig. 2.17: The Object Engineering Process (OEP) micro process (according to [Oes02])

Extension mechanisms

The Unified Modeling Language [OMG03d, Sects. 2.6, 2.14, 3.16–3.18], [OMG03b, Sect. 13], [OMG03c, Sect. 18] defines several standard extension mechanisms, including *Stereotypes*, *Constraints*, and *Tagged Value Definitions*. These mechanisms can be used to specialize or refine the UML for a specific purpose at the metalevel (see figure 2.16 on the facing page) (M2), i.e., to constrain the range of valid models (M1).

In the advent of UML 2.0, these extension mechanisms have been further streamlined, avoiding redundancies in the definition of structural metamodel elements:

Definition 2.14 ([OMG03b, Sect. 13.1.7])
Stereotype *is a kind of Class that extends Classes through* Extensions*. Just like a class, a stereotype may have properties, which may be referred to as* tag definitions*. When a stereotype is applied to a model element, the values of the properties may be referred to as* tagged values*.*

Stereotypes thus inherit the structural capabilities of normal Class, e.g., having Constraints (like any other Element) or Properties (here interpreted as tagged values).

Stereotypes can be used to classify specific meta-classes of the reference meta-model[29]. They are categorized into:

- **Decorative stereotypes** define (visual) notations for meta-classes;
- **Descriptive stereotypes** describe usage relationships by assigning comments to meta-classes;
- **Restrictive stereotypes** constrain the structural and behavioral properties of meta-classes.

Constraints further specialize conditions for contents, states, or semantics of meta-classes that always have to be fulfilled. UML allows constrains to be specified either in natural language or using the *Object Constraint Language (OCL)*;

Tagged Value Definitions are properties (i.e., typed attributes) of stereotyped meta-classes. Assigning certain values to distinct meta-class instances may alter their semantics and constraints.

UML Profiles

A set of such refinements for selected elements of the reference meta-model constitutes a *UML profile*:

Definition 2.15 ([OMG03b, Sect. 13.1.5])
A profile *defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.*

Additionally, natural language *descriptions* can be used to further express constraints and semantics. Visual *notations* may further customize the presentation of extensions. In future versions, *heavyweight extensions* will be available to modify the reference meta-model itself by adding Meta-Object Facility (MOF) [OMG03a] elements to the profile context. Profiles form name-spaces for the uniqueness of their contained extensions. Generalization of profiles can be used to extend the semantics of parent profiles. Elements from related profiles can be imported. Compatibility between profiles can be declared, meaning that elements will not collide. Note that even derived profiles declare *meta*-models (cf. figure 2.16 on page 60), i.e., they operate on M2 to define valid M1 level models.

[29] Stereotypes are bound to model elements of the reference metamodel by *Extensions*, a new feature since UML 2.0.

An OMG white paper [OMG99] describes how to use these extension mechanisms to build custom UML profiles. This approach has been formalized in the current UML 2.0 specification [OMG03b, Sect. 13], [OMG03c, Sect. 18]. The currently accepted way of defining UML profiles is a tabular form, which lists stereotypes together with their base meta-class, parent stereotype, tags, constraints and description (cf. [OMG03c, Appendix C]). Examples for such profiles[30] include, e.g., the UML Profile for Enterprise Distributed Object Computing (EDOC) [OMG02b, OMG04b], the UML Profile for CORBA Components [OMG04c], or the UML Profile for Enterprise JavaBeans [Gre01], which is explained in detail in section 2.5.1 on the following page.

UML Components

A few introductory remarks about how Components are modeled in UML have already been mentioned in section 2.2.1 on page 15. The concept for components themselves is quite simple as described in definition 2.5 on page 16. The underlying concepts and assumptions have been gathered in definition 2.7 on page 16. Figure 2.18 displays the most prominent features of this model: A *Component Specification* includes a set of supported *Component Interfaces*. The specification can be realized by multiple *Component Implementations*, which can in turn be deployed multiple times as *Installed Components*. Concrete *Component Objects* are instantiated within the context of a particular installed component.

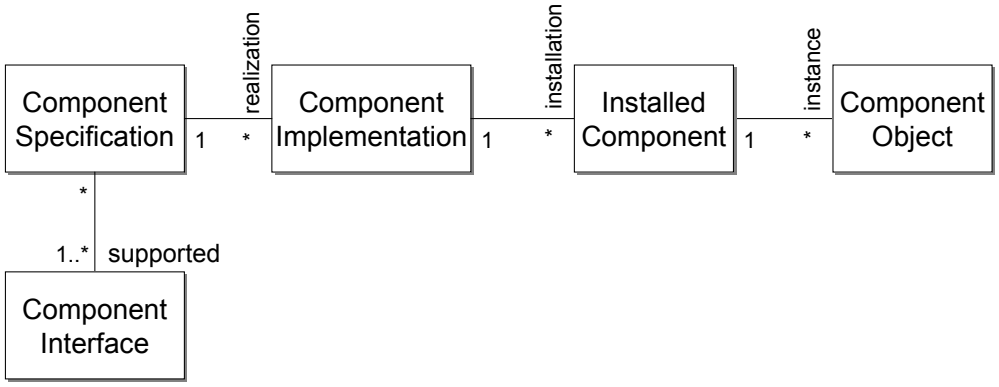

Fig. 2.18: Component forms according to [CD00]

The structural features and visual notation of components has been revised in UML 2.0. Diagrams for components, composite structures (including components), and deployments (i.e., physical placement and installation of components) are described by the UML Superstructure [OMG03c, Sect. 8–10]. Examples for component diagrams have already been given by figure 2.4 on page 26, figure 2.6 on page 32, and figure 2.9 on page 38. The enhanced structural capabilities of components in UML 2.0 (e.g., ports, parts, and connectors)

[30] For a more complete and current list of UML profiles, see `http://www.omg.org/mda/specs.htm#Profiles`.

are already a result of criticism concerning the limited expressiveness of former revisions, which was voiced by Cheesman and Daniels [CD00], among others.

Example profiles

To further constrain UML component modeling capabilities to specific platforms, UML 2.0 superstructure [OMG03c, Appendix C] contains example profiles for common component models, like J2EE/EJB, COM, .NET, and CCM. These examples are however not fully compatible to the competing profiles mentioned in section 2.5.1 on page 61. This inconsistency shows that the profile mechanism is an unstable, yet vivid part of UML under active development.

UML Profile for EJB

It has been mentioned introductorily that UML serves the purpose of a visual modeling notation in section 5 on page 131. Furthermore, Enterprise JavaBeans (EJB) has been anticipated as the example platform for demonstrating the realization of method-based caching as a middleware service in section 4 on page 107. Hence, we will closer examine existing possibilities for modeling EJBs with UML.

UML profiles have been introduced in section 2.5.1 on page 61 as a flexible mechanism for constraining the use of UML model elements to specific platforms or application domains. A number of example profiles have also been mentioned. The need for a *UML Profile for EJB* has been realized quite early, and this realization led to the Java Specification Request 26 [Gre01], which was initiated in 1999 by Rational Software. Basically following the OMG's guide for writing profiles [OMG99], it defined a set of stereotypes and tagged values for modeling component-based applications conforming to the EJB v1.1 specification [MH99]. It was however withdrawn in March 2004, mainly because of personnel deficiencies after restructuring within the participating companies[31].

JSR 26 withdrawn

This is rather unfortunate, since the *Metamodel and UML Profile for Java and EJB Specification* [OMG04b]—the Java-binding of the *UML Profile for Enterprise Distributed Object Computing (EDOC)* [OMG02b]—directly references the above mentioned *UML Profile for EJB* [Gre01]. It remains yet unclear how the OMG intends to bridge this gap. However, it has already been mentioned above that the UML 2.0 superstructure [OMG03c, Sect. C.1] also provides an (incompatible) example profile for EJB, albeit a very limited one.

Model transformer and code generator tool providers, which will be used in section 5 on page 131, need profiles for defining valid input models for their transformations (cf. section 2.5.2 on the next page). Some of them, e.g., AndroMDA [Boh04], follow a different approach to solve the problem of a missing UML profile for EJB: They use a general EDOC-like [OMG02b] profile for

[31] according to a personal e-mail from Andy Dean, the current specification lead for JSR 26 at IBM

modeling and leave the issues of technology projection to the code generator templates. More details of this procedure will be explained in section 5 on page 131.

2.5.2 Model Driven Architecture

While UML (see section 2.5.1 on page 58) supports software design by (graphical) modeling, it does not address issues of implementation. Many tools for *Computer Aided Software Engineering (CASE)* already provide export functions for code generation, the resulting program code is usually decoupled from its originating model. This significantly reduces maintainability, i.e., changes made on either side have to be synchronized by hand. But there are also other challenges, e.g., the transformation of (UML) models of different abstraction levels, where appropriate support for model consistency was missing.

Problem: decoupling of models and code

These challenges were the driving force behind the OMG's initiative towards a *Model Driven Architecture (MDA)* [MM03]. The main goal of this endeavor is to support the whole software development cycle end-to-end by means of reusable models. This implies the ability to describe transformations between models at different abstraction levels to be able to specify systems independently of the platforms that support them:

Model transformation

Computation Independent Models (CIM) are domain models from the viewpoint of practitioners of a specific domain. They do not contain any references to technical artifacts used to model domain concepts.

Platform Independent Models (PIM) capture the domain model in a more technology-centric manner but still at a very general abstraction level without references to platform-specific realization concepts.

Platform Specific Models (PSM) bind the details of a PIM to the specific properties and capabilities of a target platform.

Platform Models specify the concepts, services and technical artifacts of particular target platforms for use in PSMs of that specific platform.

The motivation for this differentiation of models is a matter of separation of concerns in modeling: It prevents aggregation of too much information in a single layer. The distinction makes the different levels manageable by avoiding confusion. The most important models and/or viewpoint in the Model Driven Architecture are the Platform Independent and Platform Specific Models. The idea is to apply transformation operators to PIMs for generating corresponding PSMs, as drawn in figure 2.19 on the following page. This process is iterative, i.e., PSMs can be further refined and again be used as input (PIM) of yet another transformation, resulting in a new PSM. *UML Profiles* as introduced in section 2.5.1 on page 61 are used to constrain the range of valid UML models usable as PIMs / PSMs for generating (further) PSMs. Platform-specific profiles can be used to assign application model elements to concepts

Relation to UML Profiles

and roles of a target Platform Model by using descriptive and restrictive stereotypes. The end of this transformation chain is reached when some template-driven code generator creates runnable code as the final output.

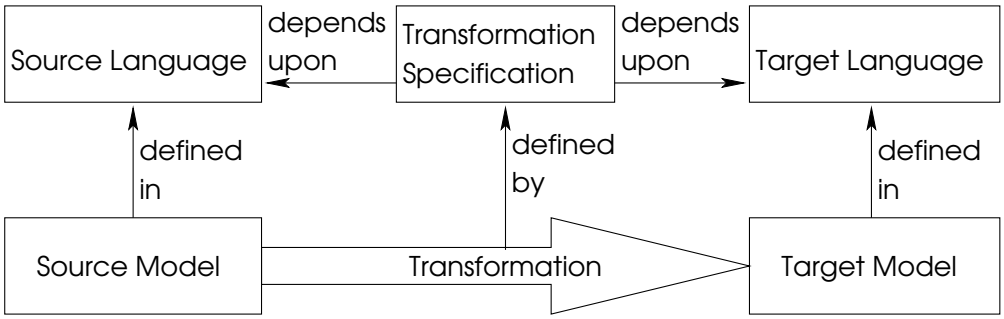

Fig. 2.19: Overview of the MDA process according to [KWB03]

Advantages The MDA approach shows its advantages best when used as a generator for building tiers of applications, e.g., multi-tiered server-based applications running on top of (component-oriented) middleware (cf. section 2.2 on page 14). In this scenario, MDA eases the management of communication dependencies between different target PSMs at model and code level [KWB03], for instance, the relational database model, EJB component model, and Web application model of a given application PIM. MDA furthermore provides the ability to reflect custom changes at PSM or code level back in the originating PIM.

Disadvantages However, MDA offers no real support for integrating of existing code bases. It is furthermore still in its early phase of development, meaning the proposed process is far from being stable. Up to now, there is no real tool interoperability. The definition of transformation descriptions is currently completely vendor-specific.

We will use MDA in section 5 on page 131 as a concept for generating caching-related code elements from a PSM, which uses a specific UML Profile. For a more concise discussion of the MDA approach, we kindly refer the interested reader to books like Kleppe *et al.* 's "MDA Explained" [KWB03].

2.5.3 Attribute-oriented Programming

Code generation The previous section already hinted that code generation is a frequent task in enterprise application development. Apart from its importance in the context of transforming models into runnable code, it gains even more momentum in connection with component-based middleware platforms. For instance, EJB (see section 2.2.4 on page 22) inherently introduces a large amount of mandatory but partially redundant programming artifacts, e.g., remote, home, and local interfaces, descriptors, etc. This also suggests the use of some automated generating mechanism.

This demand led to the development of *attribute-oriented programming* concepts that try to reduce the amount of code application programmers have to write. The term refers to the insertion of special *attributes* into source code with the purpose of semantic evaluation by code generation tools. Since we are going to use a specific code generator—*XDoclet* [ÖSA+03]—for creating caching-related middleware descriptors in section 5 on page 131, we will elaborate in more detail on this specific tool.

XDoclet

The XDoclet project [ÖSA+03] started out in 2001 under the name "EJBDoclet" as a simple code generation tool with the main purpose to concentrate all the code belonging to an Enterprise JavaBean component in a single Java source file. The approach is based upon the *Javadoc* concept, which was introduced in Java Development Kit v1.2 as a facility to enrich source code with additional metadata for documentation purposes. To maintain backward compatibility to the language specification, metadata is inserted by means of special *tags* in source code comments that are ignored by unaware tools. XDoclet introduces a number of such tags as the example in listing 2.1 on the following page shows. It should become obvious that XDoclet tags have the following form:

Javadoc

```
@namespace.tag-name attribute-name="attribute value"
```

The XDoclet tool scans these tags and constructs all necessary programming artifacts, i.e., concrete bean class, remote, home, and local interfaces (complete with business and finder methods), primary key and utility classes, standard and vendor-specific descriptors, etc., from the (abstract) commented bean class in listing 2.1.

However, this "one source" concept has also been criticized for undermining the role model of EJB (cf. section 2.2.4 on page 24): If deployment descriptors are to be generated from Bean class files, application assemblers and deployers need access to source code for reproducibly changing deployment configurations. XDoclet-based applications are also harder to debug since Javadoc-tags are not directly evaluated by compilers, which makes errors harder to trace.

Disadvantages

Besides EJB components, XDoclet also supports Web components (Java ServerPages and Servlets), management components (JMX), and different persistence engines (Hibernate and JDO), among others. These *tasks* generate output based on XML-like *XDoclet template (XDT)* files. Output of the templates can be customized at predefined *merge points* that allow inserting arbitrary user-defined code segments. XDoclet tasks are usually called from *Ant*[32] scripts. Subtasks may be defined for extending the functionality of existing

XDoclet extension mechanisms

[32] Ant is a Java-based make tool. For an introduction to Ant, XDoclet, JUnit, and other open-source, Java-based development tools, refer to Hightower *et al.* 's book [HOV04]. Fowler and Foemmel's *Continuous Integration* [FF03] also shows how to integrate these tools into a lightweight development process.

Listing 2.1: XDoclet usage example according to [ÖSA+03]

```
1  /**
2   * This is an Account entity bean.
3   * @ejb.bean
4   *     name="bank/Account"
5   *     type="CMP"
6   *     jndi-name="ejb/bank/Account"
7   *     primkey-field="id"
8   * @ejb.finder
9   *     signature="java.util.Collection findAll()"
10  *     unchecked="true"
11  * @ejb.transaction
12  *     type="Required"
13  */
14 public class AccountBean implements javax.ejb.EntityBean
    {
15   // ...omitted for brevity...
16 }
```

tasks. Finally, new tasks may be introduced based on XDoclet's API. A further explanation of XDoclet's capabilities is given in Wall *et al.* 's book [WRÖ03].

The list of planned features for the next release of XDoclet, version 2, contains a *runtime attribute access API* (XRAI) that uses byte-code manipulation to augment generated classes with methods for accessing their metadata attributes. This is an interesting parallel to the newer developments discussed in the next section.

Alternative Approaches

A Java Specification Request (JSR) filed under number 175 [Bra04] addresses the need for a *metadata facility for Java*. The goals of this JSR include:

- Definition of a standard language feature for the specification of metadata annotations;
- Definition of a runtime API for metadata access; and
- Definition of namespace rules for metadata annotations.

Annotations in J2SE 5

Meanwhile, the results of this JSR have been incorporated into version 5 of the Java 2 Standard Edition by the name of *annotations*. These annotations may be defined by declaring special interfaces as in line 2 of listing 2.2, which is later assigned in line 4 and queried in line 6. The new *Annotation Processing*

Listing 2.2: J2SE 5 metadata annotations example

```
1 /** A tagging interface for test purposes. */
2 public @interface Test { }
3 /** Some class using the annotation. */
4 @Test public class MyClass {
5   public static void main(String[] argv) {
6     if (MyClass.getClass().isAnnotationPresent(Test.class
        ))
7       System.out.println("Test attribute set.");
8     else
9       System.out.println("Test attribute not set.");
10  }
11 }
```

Tool (apt) can evaluate metadata annotations and start arbitrary actions, e.g., code generation.

The new metadata annotation facility is explicitly propagated as an alternative to existing approaches like JavaBeans' `BeanInfo` concept [Sun02b] or EJB deployment descriptors (cf. section 2.2.4 on page 22). This qualifies this new feature as a serious competitor of the XDoclet approach introduced above.

The upcoming version 3.0 of the Enterprise JavaBeans specification [DeM04] furthermore proposes to completely deprecate descriptors and the various interfaces of bean components in favor of the new annotations feature. An EJB would then consist of only one Java class file—a plain old Java object or POJO—containing the necessary metadata in form of annotations, which the container is able to evaluate directly. Depending on the outcome of the standardization process for EJB 3.0, XDoclet might become completely obsolete for its original target—simplifying the development of EJBs.

Metadata attributes in C#

It is interesting to note that Microsoft's C#—the favorite language for .NET (see section 2.2.6 on page 38) development—also provides a similar feature for metadata attributes [MSg, section 17]. In analogy to tools like the above mentioned AndroMDA [Boh04] or JavaGen [Out04], which transform UML models into Java files with (XDoclet) metadata attributes, there also exist tools like *tangible architect* [Tan], which produce .NET components written in C# containing metadata attributes.

Conclusion

Given the range of alternative solutions and the engagement of major players like Sun and Microsoft, it seems that attribute-oriented programming will definitely gain more attention in the near future—regardless of the fate of tools like XDoclet. Hence, we will try to build on these ideas and concepts in the context of this work, bearing in mind that at least one comparable solution

will eventually prevail.

2.5.4 Design Patterns

Patterns in general are sets of rules that allow modeling or generating items or parts thereof. The first to apply the concept of patterns to *design* was the American architect Christopher Alexander, who proposed *A Pattern Language* [AIS77] for architecture and civil engineering. His approach targets at the same goals as modern *software design patterns*: Patterns capture ideas, constructs, and design decisions that proved to be successful in certain environmental situations. Hence, architectural design is just a matter of consequent (re-)application of simple principles. Alexander stated that patterns cannot be invented but have to be discovered, discerned, or inferred from existing solutions. Patterns are made reusable by abstracting from their concrete usage scenarios.

Origins of design patterns

Although some of McIlroy's ideas, e.g., pipes & filters [McI64] could be interpreted as early design patterns in software engineering, the first software architects to apply Alexander's concepts were Beck[33] and Cunningham [BC87], who introduced a set of patterns for designing Smalltalk programs. But the book that eventually attracted more attention to software design by patterns was "Design Patterns—Elements of Reusable Object-Oriented Software" [GHJV94] by Gamma, Helm, Johnson, and Vlissides (the "Gang of Four"), which introduced 23 general design patterns categorized into *creational*, *structural*, and *behavioral* patterns. One of the main achievements of this book was the definition of a notation for patterns that gradually became standard for later publications. Buschmann *et al.* applied the principle of design patterns to the architecture of whole software systems [BMR+96]. The second edition of this book was split in two volumes [BMR+00, SSRB00] of which the latter especially dealt with constructing middleware (cf. section 2.2 on page 14) architectures using common design patterns. We will occasionally refer to some of these patterns later in this work and explain them as needed.

Pattern-oriented software architecture (POSA)

The component-based software engineering community also produced a number of design patterns, e.g., for the construction of J2EE-based (see section 2.2.4 on page 22) applications [ACM01], of which a few will also be subject of closer inspection in section 3.1.4 on page 91.

2.5.5 Meta-Programming

The general term *meta-programming* refers to programming at the abstract metalevel that is used to describe valid programs. In analogy to the MOF terminology (cf. figure 2.16 on page 60), this means access to level *M2* for programs at level *M1*. Since meta-programming is inherently suited for developing mid-

[33] the same Kent Beck who proposed the *extreme programming* software development method; cf. section 2.5.1 on page 61

dleware services [McA96], we will explore *reflection* and *interception* as two mechanisms that will be employed later in section 4 on page 107.

Reflection

Transparently adapting the behavior of programs is not a new idea: It actually dates back as early as 1982 when Smith submitted his thesis about *Procedural Reflection* [Smi82]. Although models have changed greatly since then, the basic concept is that programs (or components) should have the means to inspect their current context and (limited) control over their interpretative environment.

Reflective systems are defined in Maes' thesis about computational reflection as follows:

Definition 2.16 (Pattie Maes [Mae87])
A reflective system *is one that supports an associated* causally connected self representation (CCSR).

Definition 2.16 addresses the capability of a system to reason about and act upon itself. The "causal connection" refers to the direct way how changes of the self representation immediately effect in the underlying system's actual state and behavior. This implies a number of concepts, which have been summarized by Kon *et al.* [KCBC02] and Blair *et al.* [BCRP98], among others:

Causal connection

Reification denotes the ability to access and manipulate the internal representation of a system at runtime. Access to the meta-level is also often referred to as *introspection*; manipulation as *intercession*, respectively.

Absorption in turn denotes the execution of these changes, thus realizing the causal connection between both abstraction layers.

Meta-level architectures explicitly consist (at least) of a *base-level* for application concerns and a *meta-level* responsible for reflective computation, i.e., capable of performing *reification* and *absorption*.

Meta-object protocols (MOP) are needed to specify the means for manipulating the meta-level of object-oriented reflective architectures, i.e., *meta-objects*.

Structural reflection allows *reification* of structural features of an application including its functionality.

Behavioral reflection additionally allows manipulating the semantics and internal aspects of the runtime environment including non-functional properties.

One of the first—and possibly best—known *meta-object protocols* was introduced in the book "The Art of the Meta-Object Protocol" by Kiczales *et al.*

[KRB91], which describes a mechanism for manipulating the semantics of inheritance, method dispatching, and class instantiation applied to the example of the Common Lisp Object System. Technically, modern languages like Java only provide limited means of *introspection*, which are rather far from the capabilities of full-fledged reification as provided by MOP. This was actually one of the driving forces behind the development of *Aspect-Oriented Programming (AOP)* (see section 2.5.6 on page 75).

AOP

According to McAffer [McA96], distributed systems are inherently predestined for reflectional programming, due to distribution transparency that is usually aimed at[34]. Distribution itself can be understood as a non-functional aspect that should ideally be separated from application logic by means of meta-programming mechanisms. This realization is also reflected in other publications [BCRP98, KCBC02], which leads to the definition of *reflective middleware*:

Reflective middleware

Definition 2.17 (Geoff Coulson [Cou01])
Reflective middleware *is simply a middleware system*[35] *that provides inspection and adaptation of its behavior through an appropriate CCSR*[36].

Buschmann *et al.* [BMR+00] described reflection as an *architectural pattern* (cf. section 2.5.4 on page 70) for adaptable systems (cf. section 2.4 on page 53).

The Proxy Pattern

In distributed object-oriented systems (see section 2.2 on page 14), binding objects called *stubs* (cf. figure 2.1 on page 20) are typically employed as *proxies* to support distribution transparency.

Stubs

Proxy is a common pattern (see section 2.5.4 on page 70) in object-oriented software design [Sha86, GHJV94, BMR+96, BMR+00]. It refers to objects installed as representatives for some (remote) delegate, which they control and whose interface they follow. This approach is typically taken by RPC-style middleware like Java RMI and CORBA GIOP/IIOP (see section 2.2 on page 14) where *stubs* proxy client requests and transparently handle marshalling of arguments and return values, among other tasks.

These proxies are often used as access points for meta-programming. *Smart proxies* and *interceptors* are two approaches for meta-programming at proxy level, which are discussed in the following two sections.

Smart Proxies

Smart proxies have been developed in the CORBA world as a simple meta-programming mechanism for behavioral reflection that intercepts calls from

[34] Cf. [ISO95] for the definition of distribution transparency.
[35] See definition 2.9 on page 18.
[36] See definition 2.16 on the page before.

clients. They are application-provided stub implementations that may transparently override the default stub behavior as defined by the ORB implementation to customize client behavior on a per-interface basis [WPSO01]. Like other meta-programming mechanisms, smart proxies allow adapting existing applications late in the software development cycle by exchanging the custom proxy implementation without having to change existing application code.

Some ORB vendors like TAO [TAO], Inprise, and Iona adopted and pushed the development of smart proxies, but e.g., Iona is already discontinuing support for this feature. Smart proxies provide better performance than *portable interceptors* as shown by Wang *et al.* [WPSO01], but they provide less flexibility and are not standardized by the Object Management Group. Koster and Kramp [KK00] showed that smart proxy functionality can be implemented using interceptors, but not vice-versa.

Interceptors

Interceptors have been in the focus of discussion for a few years as a behavioral reflection mechanism until they were finally integrated in the OMG's CORBA *Portable Interceptor* specification [OMG01, OMG04a]. The document describes interception points at *request* level, where additional metadata may be added or read by custom interceptors. This concept was integrated for the primary purpose of transparent context propagation, e.g., for security and transaction metadata, along with (distributed) object invocations. Later proposals for *network* and *object* level interceptors have not been integrated into the specification. The former were meant for transparent adaptation of used transfer protocols, the latter for object life-cycle manipulation.

Schmidt *et al.* [SSRB00] describe interceptors as a service access and configuration pattern (see section 2.5.4 on page 70) for transparently adding services to a framework, which are automatically triggered upon certain events.

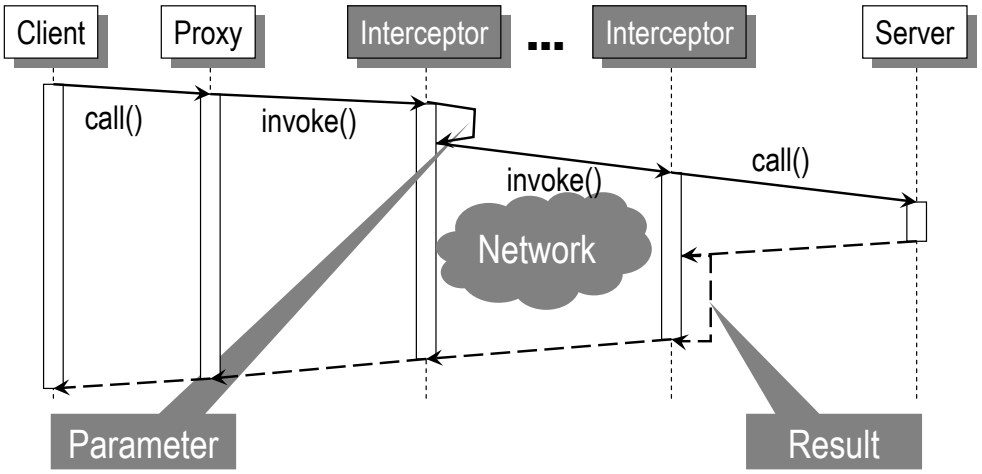

Fig. 2.20: The principle of interceptors

The basic scheme of interceptors is shown in figure 2.20 on the page before. On both, client and server side, interceptors can be hooked into the control flow of (remote) operation calls, basically to add parameters and to augment results, but generally to alter virtually any property of a call's context, even its semantics. The initial stub / proxy translates calls into weakly typed `Invocation` metaobjects containing the necessary information for appropriate processing. In the original interceptor pattern [SSRB00], which is also adhered by the OMG [OMG01], the proxy is then responsible of calling each of its configured interceptors in succession, which requires the interceptors to return control immediately. In the case of CORBA interceptors, this leaves no possibility for interceptors to alter the general proceeding of call processing.

JBoss Interceptors. Another implementation variant realizes interceptors closer to the scheme drawn in figure 2.20 on the preceding page. For instance, the J2EE (see section 2.2.4 on page 22) application server JBoss [JBo04, FR03] features an interceptor variant that closely resembles the behavioral design pattern *Chain of Responsibility* as introduced by Gamma *et al.* [GHJV94].

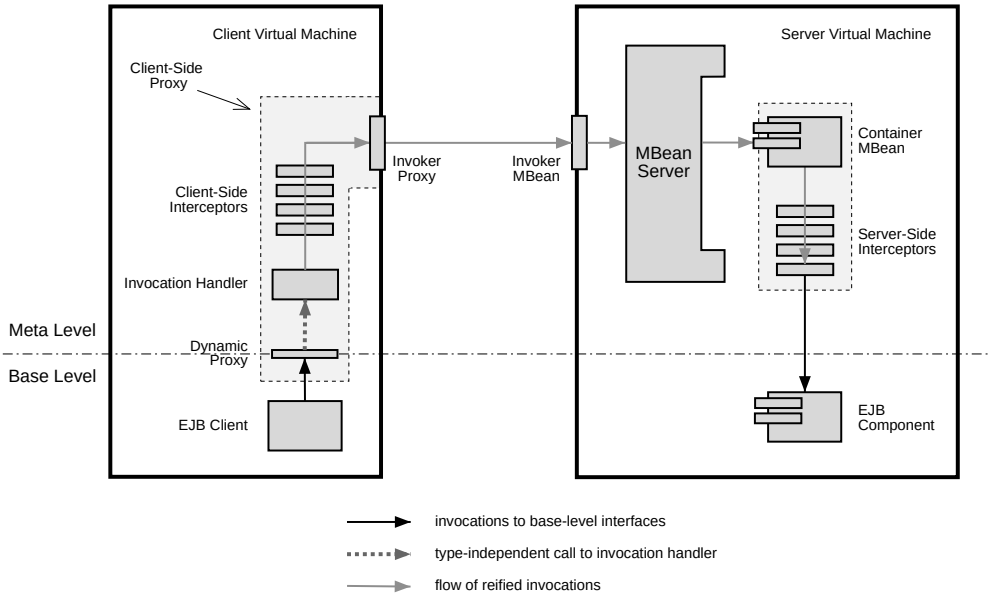

Fig. 2.21: JBoss metalevel architecture [FR03]

Figure 2.21 depicts this interceptor variant as described by Schaefer [Sch02], and Fleury and Reverbel [FR03]. The transition from base level (M1) to metalevel (M2) is achieved by `DynamicProxy` objects, a concept of the Java Reflection API. A chain of client- and server-side interceptors is configured for each deployed component. Each interceptor in this chain is responsible for invoking its successor. The `InvokerProxy` is the terminal client-side interceptor, which eventually calls the potentially remote `InvokerMBean`. This is the entry point to the server side, where the `MBeanServer`, i.e., the "message

dispatcher" of JBoss' JMX[37] implementation, selects the correct component container. The container in turn traverses a configured chain of server-side interceptors until the bean implementation and thus the actual business logic is finally invoked. Method results are then passed backwards along this invocation chain until they reach the invoking client. As mentioned introductorily, the typical use case assumes that client interceptors set service context information, which is then evaluated by server-side interceptors to decide about permissibility of invocations, e.g., based on security or transaction contexts.

Idea for integration of caching

However, any interceptor may as well skip this step and return an arbitrary result object to its predecessor. This is the basic idea of integrating a middleware service for caching, which is discussed in detail in section 4 on page 107.

Several projects, like Open ORB / Open COM [BCRP98], Lasagne [TVJ+01], OIF [FBLL02], or Fractal [DL03], leverage interceptors for building frameworks that try to hide a part of the complexity of meta-programming or to add a higher level of communication abstraction. In contrast, our goal is rather to pinpoint possible uses of basic mechanisms that already exist in current component platforms.

2.5.6 Aspect-Oriented Software Engineering

While smart proxies and interceptors, the meta-programming approaches introduced in section 2.5.5 on page 72 and section 2.5.5 on page 73, provide only limited possibilities for *behavioral* reflection at few predefined execution points, they do not address *structural* reflection at all. This insufficiency was the main motivation behind Aspect-Oriented Programming. It technically provides a meta-programming mechanism but its approach goes far beyond, because it also proposes a new *paradigm* for programming.

Behavioral and structural reflection

The term *Aspect-Oriented Programming (AOP)* was coined by Kiczales *et al.* [KLM+97], who also created *AspectJ* [Asp01], a Java-based aspect-oriented programming language. AOP can be envisioned as the logical consequence of the *Separation of Concerns* principle as proposed by Harrison *et al.* in the initial paper on *Subject-Oriented Programming* [HO93]. It is a consequential extension of the principles of reflection (see section 2.5.5 on page 71), allowing more fine-grained control than reflective systems at both, base level and metalevel. AOP especially considers *cross-cutting concerns* that can be implemented separately and integrated into a cohesive system:

Separation of Concerns

Definition 2.18 (Czarnecki and Eisenecker [CE00])
A model is an aspect *of another model if it* cross-cuts *its structure.*

A popular example for such a cross-cutting concern is logging [KLM+97, Asp01]: Logging statements are spread throughout program's code, which

[37] Java Management eXtension, an upcoming specification for managing and monitoring of services.

makes them hard to maintain, for instance, if the need for changes arises. With AOP, the logging aspect is modularized as a separate program unit that is weaved into the application at the desired points before execution. This process is supported by three concepts:

1. The *join point model*, which allows to potentially select any execution point of a system;
2. The ability to add *aspectual behavior* at the execution points identified by using the join point model; and
3. The support for *structural amendments* for modifying the static structure of a system.

The "reference implementation" AspectJ [Asp01] supports these concepts by the following constructs, which are modeled by extensions of the Java language. A preprocessor performs the *aspect weaving*, i.e., pre-compilation of "aspectized" code into compilable Java code containing the weaved-in aspects.

Aspects encapsulate cross-cutting concerns;

Join points are points in the code that can be modified;

Pointcuts describe the execution contexts in which an aspect should be activated; they refer to specific *join points*;

Advices implement the aspect functionality. They provide support for adding aspectual behavior at specific points in the execution flow of a system;

Introductions provide for structural amendments.

Aspect-Oriented Modeling

The broader field of *Aspect-Oriented Software Development (AOSD)* additionally comprises the remaining stages of software engineering, including analysis, modeling, and design. Since AOP is not only a meta-programming mechanism but also a new paradigm, it has implications on the whole software development cycle. For instance, *Aspect-Oriented Modeling* as proposed by Elrad *et al.* [EAB02, Ald03] is a burgeoning research area that addresses the lack of support for modeling of separate concerns as sub-models in UML (see section 2.5.1 on page 58). This resulted in the initial proposal for an *UML Profile (see section 2.5.1 on page 61) for AOP* by Aldawud *et al.* [AEB03], which essentially introduces stereotypes for Aspects as special Classes, Cross-Cuts as Associations, and Preactivation and Postactivation Operations. This approach closely resembles the *UML notation for AOSD* by Pawlak *et al.* [PDF+02]. The ultimate goal of these endeavors is to enable model transformation and code generation for AOSD by largely reusing concepts and practices of the *Model Driven Architecture* (see section 2.5.2 on page 65).

Caching as an Aspect

Caching as an aspect

Among the various application scenarios for aspect-orientation, caching is sometimes used as an example aspect to demonstrate concepts of certain AOP approaches. For instance, Pawlak *et al.* [PDF+02] model the use of a simple software cache that stores queried objects as an aspectual concern of an existing software system. Ségura-Devillechaise *et al.* [SDMML03] even model Web cache prefetching (cf. section 3.3.3 on page 102) as an aspect. Their motivation for employing AOP is that the prefetching concern cross-cuts the structure of cache systems.

Convergence of Middleware, CBSE, and AOSD

Another tendency that can be observed is the convergence of middleware, Component-based Software Engineering (CBSE), and AOSD. Many application server providers have started integrating AOP frameworks into their products, e.g., BEA and JBoss [BB03]. Sun has realized this development and adopted these ideas for the upcoming EJB 3.0 [DeM04]. All these examples have in common that they try to minimize the number of mandatory programming artifacts for components. Aspect-orientation helps to capture meta-data concerning service usage of components, allowing to program components like "Plain Old Java Objects" (POJOs).

One reason for this development is the realization that interceptors (cf. section 2.5.5 on page 73), which are typically used to embed middleware service usage in component-based applications, provide only a subset of AOP's capabilities while AOP can in turn be used to model equivalent interceptor behavior, as we explained introductorily. For the example of EJB, the set of mandatory middleware services could be modeled as *one possible* interceptor chain configuration. Interceptor chains form only *one possible* behavioral extension mechanism of the middleware platform that could as well be modeled by AOP mechanisms. Hence, AOP provides a superset of possible and required functionality, which enables the most flexible solutions.

Dynamic aspect weaving

The trend also moves from static aspect weaving as with AspectJ [Asp01] towards dynamic solutions, e.g., AOP framework of JBoss 4 [BB03], which use byte code manipulation and similar techniques at runtime to dynamically weave cross-cutting concerns, like middleware services, into existing application code. Thus, no modification of existing code and/or explicit recompilation is needed. Aspects are declared *descriptively*, which parallels the concept of component deployment descriptors.

Relevance for this Work

In the context of this work, aspect-orientation plays only a marginal role since interceptors turned out to provide a sufficient degree of flexibility for the as-

pired goals. However, interceptors provide only a subset of the modeling possibilities of aspect-oriented approaches as we have shown above. Hence, our middleware extensions proposed in section 4 on page 107 could as well be modeled as aspects, although this would add only little scientific value. The additional opportunities for caching services offered by behavioral and structural reflection at arbitrary point of execution have not yet been investigated.

2.6 Conclusion

What we have seen in this chapter is a general convergence of development trends from different fields. Component-based Software Engineering (CBSE) (see section 2.2.1 on page 15) has influenced the area of middleware (see section 2.2 on page 14), which led to component-based middleware platforms that allow specifying non-functional requirements of components and component-based applications in a descriptive manner. Middleware platforms need meta-programming mechanisms like reflection (see section 2.5.5 on page 71) or interceptors (see section 2.5.5 on page 73) to augment applications according to the requirements of a concrete deployment. Aspect-Oriented Software Development (AOSD) (see section 2.5.6 on page 75) is a closely related approach that allows to capture non-functional properties as cross-cutting concerns, which can be weaved into applications at compile time, at deployment time, or even at runtime. Hence, today's component-oriented middleware platforms start supporting AOSD in their frameworks, e.g., see [BB03].

On the other hand, Meta-programming and AOSD form a basic mechanism to build adaptive systems (see section 2.4 on page 53), since they both allow altering the structure and behavior of systems by manipulating their interpretative environment. Middleware platforms can be used again to bridge the gap between adaptable components and adaptive systems built from them.

Model-driven approaches (see section 2.5.2 on page 65) are related to all of the above mentioned fields, as they allow capturing additional information about non-functional properties, adaptability, and other cross-cutting concerns early at design time, to refine this information subsequently, and to reuse and transform it between different levels of detail of models, culminating in code generation. Attribute-oriented programming (see section 2.5.3 on page 66) represents the continuation of this trend towards enriching source code with meta-data to capture refined model information and to make this information readily available to the runtime environment.

Management of distributed data (see section 2.3 on page 46) has been researched by the database community for decades. Concepts like TP monitors have been the archetypes of today's component-based middleware platforms. Moreover, data replication provides many fundamental concepts for caching (see section 2.1 on page 9), which itself can be seen again as a mechanism for adapting the parameters of communication between components by buffering queried results.

In the course of this work, we will seize concepts from all of these fields to come up with an integrated solution for caching of method results of components as a middleware service (section 4 on page 107) that can be configured at development time and deployment time (section 5 on page 131), and which can adapt to changing cacheability parameters at runtime (section 4.2 on page 111).

Now that we have gathered the basic "tools" needed in the later course of this work, we are ready to take a closer look in section 3 on page 81 at concepts and approaches directly related to our ideas.

The wise speak only of what they know.

Gandalf to Grima "Wormtongue" in *Lord of the Rings—The Two Towers*
by John Ronald Reuel Tolkien (∗1892–†1973), English fantasy author
and philologist.

3 Related Work

While section 2 on page 9 introduced the state of the art concerning fundamental concepts used in this work, this chapter concentrates on research more closely related to the contribution of this work. Section 3.1 resumes the general discussion of caching from section 2.1 on page 9 and examines various approaches for caching in distributed systems. Section 3.2 on page 99 introduces other alternatives for reducing call latency by communication restructuring and section 3.3 on page 100 elaborates on prefetching as mechanism for loading probably needed data in advance.

3.1 Caching in Distributed Systems

It has been pointed out in section 2.1 on page 9 that locality of reference is an important principle of data access, which should be observed when building any kind of applications. This implies that data and the logic operating upon it should be moved as close together as possible. A general observation in distributed applications is the need of clients to manipulate data stored on servers. To maintain efficiency, application architectures have to be designed in such a way that either data is transferred to logic, closer to clients, or logic has to be implemented near data, closer to servers. A trend in *multi-tiered architectures* (see section 2.2.3 on page 21), which has been observed by C. Mohan [Moh01], is to move interactions as close to clients as possible. Caching is a key technique to reduce costs and improve user-perceived response times in this respect.

Multi-tiered architectures

Looking at the example of J2EE / Enterprise JavaBeans (cf. section 2.2.4 on page 22) as a typical multi-tiered architecture, several starting points for caching become obvious. Considerable effort has gone into development of

efficient data access paths on the server side as depicted in figure 3.1.

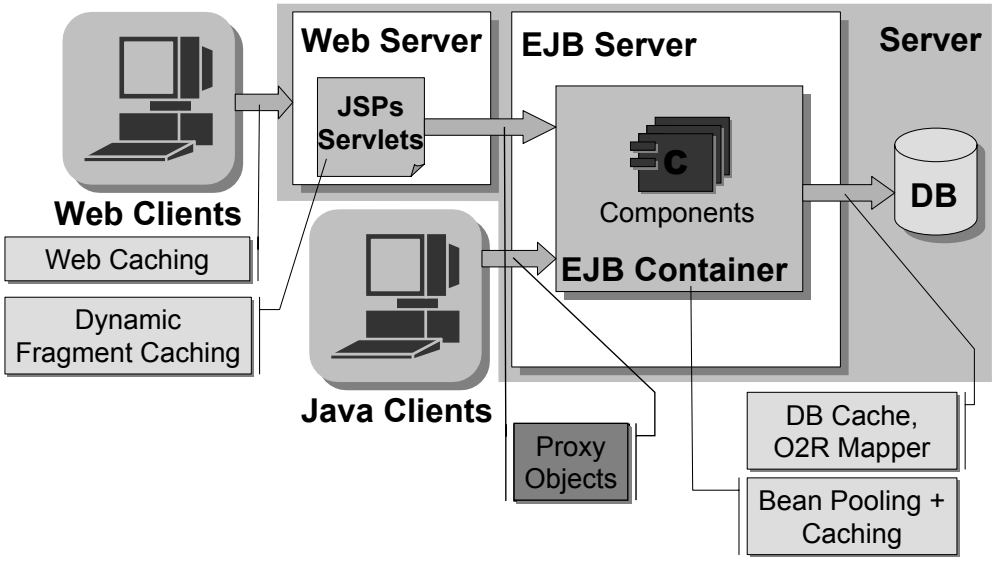

Fig. 3.1: Existing Approaches for Caching in Enterprise JavaBeans

DB Caches hook in below and above the database connectivity (JDBC) layer. Their main goal is to eliminate unnecessary database queries by caching the results of previously executed SQL statements. 0bject-relational (02R) mappers translate tuples of database relations and entities to objects. The results of this process may also be subject of caching. But also more sophisticated front-end database caches are deployable at this point. See section 3.1.3 on page 87 for more examples.

Bean Caches and Pooling are container-level mechanisms for performance optimization. Usually, containers keep a number of bean instances ready in a pool of configurable size. In case of subsequent accesses to the same bean, the state doesn't have to be reloaded on every call because it is readily available from the pool due to temporal locality of reference. But even for access to different instances, the overhead for instantiation can be saved by reusing instances from the pool.

Pooling vs. caching

It is noteworthy that despite various parallels and similarities, *pooling* is a mechanism for exclusive assignment of shared resources, while *caching* is a technique to speed up (concurrent) access to read-only or mostly-read data.

Entity beans can be passivated and stored to a temporal memory by the container based on a certain replacement strategy to limit memory consumption. Later reactivation avoids the overhead of reloading entity state from the database and object-relational mapping of query results. Additional add-on software extends these built-in caching capabilities,

e.g., by introducing persistent bean caches that store the state of passivated entities in an object-oriented database. However, the application server has to make use of consistency protocols like the ones known from replication control (see section 2.3.2 on page 51) to ensure coherence between its bean instances and the underlying persistent storage.

Clustering

Application servers may also be grouped in *clusters*, which poses a form of replication targeted at fault-tolerance and scalability. *Homogeneous clusters* replicate the same data evenly across all participating servers. *Heterogeneous clusters* split servers into groups responsible for serving subsets of a distributed application. Clustering may also be applied for Web servers at the presentation tier. Again, replication control is required to maintain coherence of replicas in the cluster.

Fragment Caches are used by Web servers (the presentation tier) to save the effort for repeatedly constructing the same dynamic page[1] or fragments thereof when several requests refer to the same Uniform Resource Identifier (URI) [BLFM98]. Some application servers, e.g., BEA WebLogic [BEA00], use additional tags for controlling cacheability in the JSPs themselves, which may prevent parts of a JSP from being retransformed or recalculated. Rabinovich and Spatscheck called this "caching the uncacheable" [RS02, Chap. 13]. Servlets have to be programmed explicitly to reuse cached data, e.g., by leveraging object caching frameworks like those we will briefly discuss in section 3.1.5 on page 95. But since web servers are not necessarily deployed on the same network node or within the same process as the EJB container, the whole presentation tier will also benefit from caching at the proxy layer (see below).

Web Caches include Web proxies and browser caches like they can be found in almost every Web browser. Both of them retain HTTP response data mapped to the corresponding URIs. This strategy is only partially effective in its original form in many of today's scenarios, due to dynamic, personalized contents. More details of Web caching are discussed in section 3.1.1 on the next page.

Proxy Objects are used by distributed middleware platforms to provide distribution transparency, as we have discussed in section 2.2 on page 14. Both client logic and presentation logic use this abstraction level for programming. This opens up two possibilities for caching implementations: Either on top of this abstraction as an extension of external component interfaces of the business tier or below as an additional middleware service. Approaches to the former alternative will be introduced in section 3.1.4 on page 91, examples for the latter one in section 3.1.5 on page 94.

[1] i.e., Java ServerPages (JSP) or Servlets

3.1.1 Web Caching

Caching of Web documents is doubtlessly the most prominent example of caching in distributed systems, which has received the greatest attention during the last decade due to the tremendous growth of the World Wide Web (WWW). Again, major goals comprise reduced server load and bandwidth consumption as well as improved user-perceived responsiveness. The Hypertext Transfer Protocol (HTTP) [FGM+99] forms the basis for transfer of data objects on the WWW. Every data object is globally locatable by a Uniform Resource Identifier (URI) [BLFM98]. Users typically have browsers as (graphical) user interfaces for downloading and displaying data objects, i.e., hypertext documents and embedded multimedia files.

Proxy Caches

Web caching is technically accomplished by buffering responses to HTTP requests along their path from server to client. This can either be done by caches directly in front of the servers, by relaying proxy caches anywhere on the network, or by browser caches on the client machines. Server-side caches, also known as *reverse caches*, primarily aim at reducing server load. Client-side caches run by Internet service providers (ISPs) or local intranet providers are also called *forward caches*; their goal is to reduce latency for their local user population. HTTP specifies the conditions and details for caching of requests [FGM+99, Sect. 13]: Protocol headers may essentially control cacheability and give a "time to live" (TTL), i.e., a cache expiration time.

HTTP: Cacheability per object

Content Delivery Networks

The idea of cooperating proxy caches and so-called surrogate servers led to the development of *Content Delivery Networks (CDN)* [DMP+02], [RS02, Chap. 15]. One goal is to exploit geographical locality of reference to a higher degree by propagating data over an infrastructure closer to its consumers. Selection of surrogates can be accomplished, e.g., by dynamically manipulating server name entries in the Directory Name Service (DNS)[2] [Moc87] or by URI rewriting in gateway servers[3]. Apart from mere content caching CDNs also provide functionality for authentication, authorization, and accounting (AAA) of content access. Although standardization of interfaces for content distribution interoperability by the Internet Engineering Task Force is still in progress, CDN providers like *Akamai* already provide proprietary commercial solutions. They try to reduce cost for redundant excess capacity by reusing their infrastructure for multiple customers. Since CDNs do not only cache data upon request but may also proactively propagate data to surrogate servers, part of their functionality can also be categorized as *prefetching*. We will therefore resume this discussion in section 3.3.3 on page 102

Replacement strategies

A considerable amount of cache replacement strategies (cf. section 2.1.2 on page 12) have been proposed especially for Web caching. Podlipnig and Böszörmenyi [PB03] give a comprehensive survey of the current state of the

[2] Clients access servers by symbolic names, which are resolved via DNS. The assignment of names to physical servers can be manipulated by the CDN according to current server load.

[3] All requests initially go to the same (cluster of) Web server(s), which redirect(s) them to different URIs. These refer to some server performing the actual work.

art in this field. Adaptive approaches, like ACME presented by Ari *et al.* [AAG+02], try to self-optimize caching by dynamically exchanging replacement strategies according to current workload. More examples are discussed in [PB03, Sect. 5.1].

Consistency

For managing consistency (cf. section 2.1.1 on page 11) of Web caches, plain *client validation* is the prevailing strategy instead of *server invalidation*, which is only used for informing mirror servers and surrogates in CDNs [RS02, Chap. 10]. However, a number of research approaches implemented piggyback schemes that closely resemble our approach presented in section 4.2 on page 111. *Piggyback Cache Validation (PCV)* [KW97] batches validation requests and transmits them using normal HTTP traffic. *Piggyback Server Invalidation (PSI)* [KW98] follows the coarse-grained concept of *volumes*, i.e., groups of correlated objects, which are treated equally with respect to consistency. Volumes have version numbers that are incremented upon every modification of a contained object. Client validation requests are augmented to contain also the volume version number of every requested object, which enables the server to piggyback a list of updated objects of the corresponding volume. The server does not need to keep a list of its clients. A combination of PCV and PSI turned out to produce the least stale deliveries [KW98].

Fragment Caching

We have already mentioned above that per-object caching as defined by HTTP becomes increasingly inefficient for today's highly interactive and personalized Web applications because of the decreasing percentage of static content. However, the realization that fragments of dynamic Web pages can be categorized into static (cacheable) and dynamic (volatile) content again enabled caching at a lower level. This enables *Fragment Caching* for Web servers as depicted in figure 3.1 on page 82.

Edge Servers

But since HTTP has no knowledge about the inner structure of transferred data objects, the bottleneck between Web servers and clients persists. This led to the proposal of so-called *Edge Servers*, i.e., the infrastructure of Web servers, application servers, and database servers is moved closer to data centers on the "edge" to clients[4]. Since Web applications typically make state accessible that is maintained by a (or few) central databases, reliable caching solutions are in turn required to bridge the gaps between the architecture tiers (cf. section 2.2.3 on page 21). The bandwidth of possible approaches has already been mentioned introductorily. Database caches, which we will examine closer in section 3.1.3 on page 87, address the bottleneck between application servers and databases by introducing front-end caches in the back-end of the business tier. Application servers may themselves make use of a variety of caching techniques, like the ones we already hinted at above. Business logic running on application servers may also leverage caching services provided by their cluster middleware. In the context of this work, we will concentrate on proxy-based solutions at the back-end of Web servers or other clients of the business tier, i.e., approaches for caching the data flow between presentation tier and

[4] Cf. [RS02, Sect. 17.7] about distributed Web applications. CDNs currently try to embrace this trend of Edge Servers in their development, cf. [DMP+02].

business tier.

A brief overview of current trends in Web caching is given by C. Mohan [Moh01]. More in-depth background information can be obtained from Rabinovich and Spatscheck's book [RS02].

3.1.2 Adaptive Caching

As we have seen in table 2.5 on page 56, caching already represents a mechanism for adaptation itself. However, the term "adaptive caching" is slightly overloaded. We will briefly discuss a few exemplary approaches that try make various aspects of caching adaptive.

Adaptive replacement strategies

For instance, the existence of adaptive replacement strategies [PB03, Sect. 5.1] has already been mentioned in the context of Web caching. *ACME* (Adaptive Caching Using Multiple Experts) by Ari *et al.* [AAG+02] uses machine-learning algorithms to dynamically weight cache replacement strategies according to their success, which provides better performance for proxy cascades in Web / hyper-media scenarios.

Adaptive TTL

Web proxies usually employ a heuristic for cache validation called *adaptive TTL* based on the assumption that Web objects tend to remain unchanged if they have not been modified for a while [RS02, Sect. 10.1.2] The adaptive time-to-live t_{TTL} is computed as a fraction of the time between download t_{sent} and last modification $t_{modified}$ at the originating server: $t_{TTL} = min\{k \times (t_{sent} - t_{modified}), t_{threshold}\}$, where $k = [0.1 \ldots 0.2]$ is a constant and $t_{threshold}$ ensures that very old objects are occasionally validated. This heuristic closely resembles our approach for adaptive invalidation, which will be presented in section 4.2 on page 111.

The close relation between replicated databases (see section 2.3.2 on page 50) and distributed caching has already been outlined before. Adaptive approaches also exist in this field.

Adaptive locking

For instance, many cache consistency protocols for client-side database caches (see section 3.1.3 on the next page) take an adaptive approach to lock escalation. They dynamically adapt the granulate of locks from page level to object level if other locks already exist on the same page.

Adaptive replication

Lenz [Len97] proposes *adaptive replication*, which aims at a rather static adaptation of platform-provided data replication consistency to application requirements by allowing a flexible specification of consistency requirements and controllable, application-specific inconsistencies.

Divergence Caching

Huang *et al.* 's *Divergence Caching* [HSW94] tries to reduce response time and bandwidth consumption of access to online databases by permitting transactions to read stale (*divergent*) data. Each read has to be associated by its tolerance to divergence. A refresh rate denotes intervals between automatic cache refreshes. The authors start with a static approach of fixed refresh rates

and extend this to Dynamic Divergence Caching, which adapts refresh rates to current intensities by monitoring a sliding window of read-write requests. This aspect of Divergence Caching is closely related to the above mentioned adaptive TTL and thus slightly comparable to our approach presented in section 4.2 on page 111.

However, no approaches are known so far that explicitly aim at adaptive determination of cacheability.

3.1.3 Database Caching

Database caching addresses the connection between business tier and database tier. Scalability is intended to be improved by reducing the workload on back-end database servers. This domain is not directly related to our work since we address a higher layer of enterprise application architectures. However, many approaches, e.g., for cache consistency, can be used as archetypes for similar problems at higher layers. Apart from that, observations in section 3.1.1 on page 84 already showed a general convergence of caching approaches from different architectural tiers.

OODBMS

Much research has already been done in the early 1990s in the context of Object-Oriented Database Management Systems (OODBMS), which already provided solutions for non-persistent caching in their client-side object buffers. For instance, Adaptive Call-Back Locking (ACBL) [FC94], which will be discussed below, was developed in this context and is still the most important protocol in this application area. In fact, many concepts for database caching originate from the background of database buffer management in general and client-side object buffers in particular.

In analogy to homogeneous, replicated databases (see section 2.3.2 on page 50), the simplest approach would be to clone entire back-end tables in front-end caches. However, the required amount of (largely useless) data transfer upon updates jeopardizes the potential benefits of this strategy. This realization led to a number of improvements, as illustrated in surveys by C. Mohan [Moh01] or Härder and Bümann [HB04, BH04]. We will give a brief overview in the following paragraphs.

Materialized Views

Materialized views are a method for query result caching. Listing 3.1 on the next page gives an impression how to define a materialized view `germancat` for limiting `catalogue` entries to the countries Germany, Austria, and Switzerland. The result will be cached by the front-end database, which will use its contents for answering subsequent queries. A major problem of materialized views are overlapping, redundant data sets from different queries. This introduces replication to the cache, which in turn requires strategies for keeping replicas consistent.

Quasi-copies

A similar but older approach are *Quasi-copies* [ABGMA88], which have also been reviewed in [Len97, Sect. 4.4.3, p. 113]. Database clients use read-only

Listing 3.1: Example for a materialized view

```
1 CREATE SUMMARY TABLE germancat AS
2 (SELECT * FROM catalogue
3 WHERE catalogue.state IN ('DE','AT','CH'));
```

replicas[5] for which they specify a defined degree of deviation from the master copy (i.e., the server). Coherence conditions include *delay* for temporal deviation, *version* for deviation in terms of a number of write operations, and a *arithmetic* condition for deviation of value differences. Continuous *alive messages* from server to clients communicate the new data value, version number, or an invalidation request to let clients decide about violation of their local coherence predicates.

If database caches are not only expected to answer queries that have been identically issued before but also ones whose answer is a subset or union of previous queries, the cache needs query processing capabilities and the challenge of *query containment* arises.

Semantic caching

One solution for this problem is *semantic data caching* for client-server relational database environments [KB94, DFJ+96]. It has been shown that semantic caching outperforms page and tuple caching because of the smaller amount of result data that has to be transferred. This is achieved by issuing potentially smaller remainder queries after intersecting a client's current cache content with the expected query result. The principle is depicted in figure 3.2 on the next page: Let Q be a query $Q = x > x_1 \wedge y \leq y_3$. The current cache content can be described by $V = x < x_2 \vee (y \geq y_1 \wedge y \leq y_2)$. Hence, Q can be split into *probe query* $\mathcal{P}$ and *remainder query* $\mathcal{R}$: $\mathcal{P}(Q,V) = P_1 \vee P_2 = (x > x_1 \wedge x < x_2 \wedge y \leq y_3) \vee (x > x_2 \wedge y \geq y_1 \wedge y \leq y_2$ and $\mathcal{R}(Q,V) = R_1 \vee R_2 = (x \geq x_2 \wedge y < y_1) \vee (x \geq x_2 \wedge y > y_2 \wedge y \leq y_3)$. Semantic caching exploits semantic locality rather than spatial or temporal locality of reference for its cache management meta-data. Individual cache entries are represented by *semantic regions* containing constraint queries as references and lists of data tuples as content. Apart from relational databases, semantic caching has also been successfully applied to other application areas, e.g., code archives [PTS05].

Cache Groups

Another solution for the challenge of query containment is *constraint-based database caching*: Based on an approach by TimesTen [Tea02] called *Cache Groups*, various extending concepts have been developed. Cache groups are collections of cache tables, whose interrelations are specified by parameterized cache constraints: *Cache keys* trigger loading tuples into the cache; *referential cache constraints (RCC)* take care of loading additionally required tuples according to data relationships between different columns of tables. The concept of *domain completeness (DC)* ensures that all tuples required to answer a query are contained in the cache. Although the approach proved to be effective, the

[5] Similar to the concept of *Snapshots*, which was discussed in section 2.3.2 on page 50.

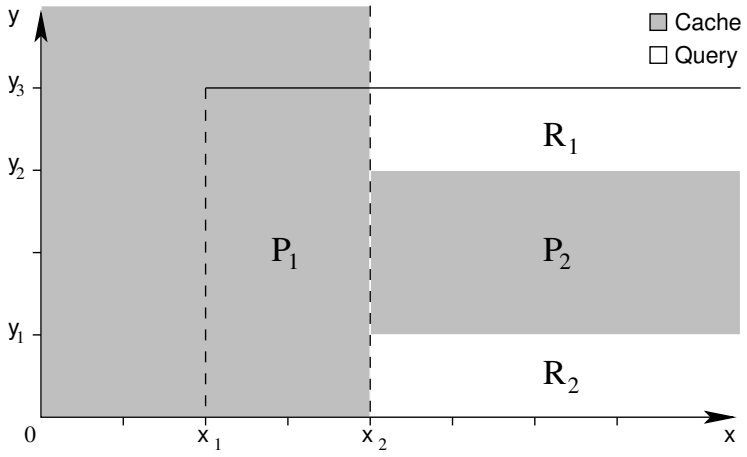

Fig. 3.2: Example of Semantic caching

complexity of specifying cache constraints is still a daunting task for administrators, which requires thorough planning to avoid degradation to full-table caching if RCCs are used for eager loading. Härder and Bümann [HB04, BH04] give a comprehensive overview of this trend.

Intertransactional cache consistency protocols

Protocols for *intertransactional cache consistency* of caches aim at providing transactional properties for cached queries, i.e., cached data from one transaction can be reused by another transaction while maintaining ACID (see section 2.3.1 on page 46) properties with special focus on serializability and recoverability. Franklin *et al.* [FCL97] have compiled a classification of existing approaches by differentiating between *avoidance-based* algorithms that prevent access to stale data within a transaction and *detection-based* ones that initially permit access but may roll back transactions after commit if collisions are detected. A second dimension is given by the point in time when the server is informed about write operations: *Synchronous* methods block clients until they receive a response to their lock escalation messages from the server; *asynchronous* methods permit clients to optimistically continue their work without waiting for the response; and *deferred* methods postpone informing the server until commit. Gruber [Gru97] furthermore distinguishes avoiding protocols into eager and lazy proactive, and detecting protocols into eager/lazy reactive and passive, depending on the efforts taken to keep cache contents up-to-date. Pfeifer [Pfe04b] also gives a good overview from a more contemporary point of view and mentions the existence of nine important protocols. The quality of cache consistency algorithms is typically measured in terms of throughput of successfully committed transactions and the abort rate of transactions that had to be rolled back due to conflicts or deadlock resolution. In the following, we shortly discuss a few exemplary protocols: *Adaptive Call-Back Locking (ACBL)* [FC94], *Adaptive Optimistic Concurrency Control (AOCC)* [AGLM95], *Asynchronous Avoidance-based Cache Consistency (AACC)* [ÖVU98]. Older approaches, e.g., *Caching Two-Phase Locking (C2PL)*, simply extend pessimistic concurrency control schemes known from section 2.3.1 on page 46 for

Avoidance vs. detection

Synchronous, asynchronous, and deferred validation

Throughput vs. abort rate

use in client-side caches, which reduces the size of required messages but not the number in comparison to no-caching solutions.

ACBL is the prevailing cache consistency protocol for client-side object buffers of OODBMS. It follows a synchronous, avoidance-based strategy, i.e., locks have to be obtained before write access is granted, and clients block until the response to their lock escalation message arrives. Read locks are obtained implicitly with data access. They are retained by clients across transactions. The server uses *call-backs* to ask other clients to release or loosen locks for required pages. The protocol is *adaptive* because of automatic refinement of lock granularity from page level to object level.

AOCC is a deferred detection-based protocol that avoids read locks and uses BOCC (see section 2.3.1 on page 49) for synchronization of write access. Invalidation messages are send to other clients after commit by the server, *piggybacked* on other messages. These invalidation messages may require the other clients to roll back started transactions. This leads to a better performance than ACBL due to less messages required but also to the penalty of a higher abort rate in high-contention scenarios.

AACC is an asynchronous, avoidance-based protocol that achieves a high performance *and* a low abort rate. Locks are managed collaboratively be clients and server at both page and object level. Lock escalation messages are send asynchronously (in a non-blocking manner) by clients. A client blocks at commit if its write operations affect another client's cache contents (i.e., collision avoidance). *Piggybacking* is only used for deferred lock escalation and invalidation messages from clients to the server.

Schaller [Sch03] developed a concept for caching client-side front-end databases, which further allows the migration of transactions between different front-end databases in mobile scenarios. The fundamental part of his work constitutes of an intertransactional cache consistency protocol based on Multiversion Concurrency Control (MCC) (see section 2.3.1 on page 49) for shielding read transactions from synchronization, in combination with Forward-oriented Optimistic Concurrency Control (FOCC) (see section 2.3.1 on page 49) for write transactions. It can be classified as a deferred, passively detecting protocol.

Pfeifer [Pfe04b] takes a similar approach for caching data access from presentation tier to business tier, at a higher architectural level, by caching results of business method invocations. We will thus elaborate on his approach in the context of section 3.1.5 on page 98. The architectural integration into the transaction management infrastructure of an EJB-based application server is provided by interposing an "*m*-scheduler" for method operations, which can execute three different *optimistic* concurrency control protocols: The first protocol is based on *2PL certification* as described in [BHG87, Sect. 4.4] (and also on BOCC (see section 2.3.1 on page 49)); it aborts transactions trying to read method results that have been invalidated by other transactions. The second

protocol is a *"freshness-based" timestamp* protocol derived from the TO concepts in [BHG87, Sect. 4.2]; it processes transactions in the sequence of their timestamps. The third, *"fitting-based" timestamp* protocol allows reading stale cache contents to some degree, similar to MCC (see section 2.3.1 on page 49). It tries to "fit" transactions into non-cyclic serialization graphs although they might contradict to the timestamp sequence. All three protocols can be classified as deferred, lazy reactively detecting, since detection of invalid accesses is deferred until commit and piggybacking is used for sending invalidation messages.

Not being bound to the limitations of the possible network communication scenarios we defined in section 1 on page 1, most database caching methods and consistency protocols rely on call-backs from servers to clients, which disqualifies them as feasible solutions. However, piggyback approaches pose an interesting option for deferred client notification, which we will discuss in section 4.2 on page 111.

3.1.4 Application Level Solutions

In this section, we will concentrate on solutions for caching data access to the business tier above the level of proxy objects (cf. figure 3.1 on page 82), within the accessing applications themselves.

Caching implementations based on the abstraction level of component interfaces have to introduce a proxy layer for encapsulating caching logic for other client application modules. The closer this proxy layer follows the original component's interfaces, the less modifications to existing client modules are needed. Design patterns (see section 2.5.4 on page 70) help with this task. *Chain of Responsibility* [GHJV94] calls the remote implementation upon local cache misses. *Decorator* [GHJV94] stores results in the cache after receiving them from the server and before passing them to the client. This approach is not novel and has been followed by many others before, e.g., *Orca* [BKTJ92], *Shadows* [CPS93], *OORPC* [ZC96], and *MinORB* [MCC99]. But full access transparency is not achievable due to the need for explicit invocation and/or creation of the proxy objects—server and client programmers have to be fully aware of these changes. The latter is especially cumbersome because changes in client code are typically hard to deploy and maintain; versioning is required etc.

Design patterns

Transparency

The described caching layer can simply try to keep copies of query results but usually several queries are triggered in sequence, e.g., because more than one attribute of a component is needed by client applications to do their work. In modern middleware as introduced in section 2.2 on page 14, there exist a number of similar patterns and approaches, which are all more or less related to the patterns *State Object* and *Session Façades*. Hardly any of them are supported by design tools or deployment utilities of containers.

State Object Pattern

The State Object pattern is not to be confused with the behavioral State pattern as introduced in [GHJV94]. It is rather a structural extension of the *Proxy* pattern. Our assumption is shared that a component's attributes / properties are exhibited by accessor and mutator methods on its interface (cf. JavaBeans [Sun02b]). The main part of the pattern is formed by a simple holder object that is used as a "virtual attribute" for bundled state transfer between an abstract client server pair, i.e., it is exhibited by an additional pair of accessor/mutator methods. This strategy makes granularity of data access coarser, thus saving a number of network round-trips as several attributes are queried together instead of being fetched individually. Other names for the same concept are *Value Object* [ACM01] and *Data Array*. But similar strategies have also already been used by *Orca*'s shared data-object model [BKTJ92]. Eberhard and Tripathi [ET01] call this concept *Reduced Object* in their RMI-based caching service (see section 3.1.5 on page 95).

Value Object

This pattern uses a fixed data structure for bulk state transfer in its original form but an extended, more flexible version called *Dynamic Property* enables clients to query a variable set of business object properties instead of the bulk of attributes. Different use cases become possible at runtime without changes to the component's code.

Dynamic Property

In common e-business applications, a substantial part of a business object's attributes is usually needed all at once, e.g., when displaying customer information in a table. To avoid costly network traffic, a State Object would be used to access the entireness of customer's attributes in a single remote call as shown in figure 3.3 instead of querying every attribute separately.

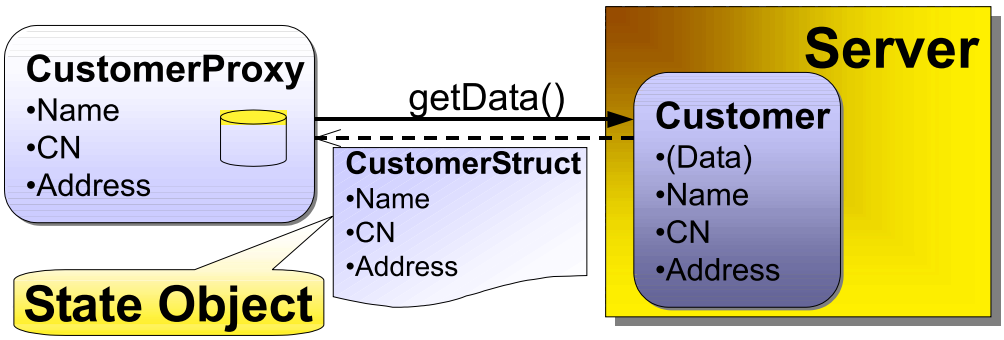

Fig. 3.3: State Object Pattern by example

Although tools like XDoclet (see section 2.5.3 on page 66) provide help for automatic generation of (multiple) state objects, this pattern still features a lack of access transparency. The client programmer has to be well-aware of how to use these mechanisms; the server-side programming efforts are non-trivial.

The caching layer could use state objects to transfer component state to the manipulating (client) process in a coarse-grained manner, which reduces

the overall number of network round-trips. This bulk state transfer can either be triggered by the client application after some modifications or even transparently by the caching layer itself, which would add a simple prefetching functionality since data is (partially) loaded in advance.

Access Beans

A similar solution called *Access Beans* is integrated in IBM's Enterprise JavaBeans product line [ST00]. Plain (non-remote) JavaBean components [Sun02b] are used to wrap EJBs. These JavaBeans facilitate easier integration into visual builder tools and encapsulate most of the typical tasks when accessing an EJB, e.g., home look-up in naming contexts and remote instance creation, in the Beans' constructors. Extended versions, called *Copy Helpers* and *Rowsets*, add caching functionality for single and multiple EJB instances in one Access Bean.

Wizards for Access Bean creation and selection of EJB data attributes for Copy Helpers are part of IBM's Visual Age for Java. Unfortunately, this is the only supported builder tool and deployment of generated Access Beans is cumbersome on application servers other than IBM WebSphere.

Astral Clones

Instead of creating additional State Object classes, *Astral Clones*, as proposed by Res [Res01], try to reuse (EJB) bean implementations out of a container's scope for this purpose after some minor modifications. These modified beans are called *Astral Clones*. Serialized instances containing the component's state can be transferred to clients by additional methods. Since these classes already contain all necessary data manipulation logic, they can be used by clients just like their remote equivalents. Special precautions have to be taken to ensure serializability, i.e., (un)marshalling, and to distinguish between the two now possible execution contexts—container and client side—in the bean's implementation. Thread safety becomes another important issue as this is normally handled by the container, which is now missing on the client side.

Only EJB 1.x

This concept works perfectly well with EJB 1.0 and 1.1 [MH98, MH99] but it is jeopardized by the Container-Managed Persistence (CMP) mechanism introduced in EJB 2.0 [DYK01]: The container now handles persistent state management by polymorphic inheritance instead of delegation. Bean implementations defined by the bean provider have to be abstract. They access their persistent state by calling abstract accessor / mutator (get-/set) methods, which are implemented by the container's persistence manager. This in turn implies that the persistence manager's concrete bean implementations have to be serializable and their behavior has to be portable outside the container's boundaries, which cannot be assumed inherently.

Session Bean Wrapper

An EJB's implementation is not always accessible for later modification when the need for caching arises. This small ancillary condition shipwrecks all the hitherto introduced patterns as they strongly rely on server-side modifications of bean implementation classes. Though it is still possible to use proxy objects with caching functionality, they would be limited to caching already-queried attributes only—no bulk state transfer could be arranged due to fixed business interfaces.

If client code modification is feasible, an additional Session Bean could be used to parenthesize (Entity) Bean access and add support for querying an object's state. The clients use the new Session Bean's interface instead and the Session Bean's implementation handles State Object construction on the server side (in the same container) thus avoiding network traffic when separately querying the Bean's attributes.

It is quite a common pattern to use Session Beans for executing operations that involve numerous interactions with server components (EJBs) on the same network node to reduce the application's distribution cross section. This pattern is also known as *Session Bean Wraps Entity Beans* [Bro01] or *Session Façade* [ACM01].

Session Façade

3.1.5 Middleware-based Concepts

As mentioned introductorily, all the concepts presented in section 3.1.4 on page 91 require a considerable amount of recoding on client and server side, which poses a violation of access transparency (and distribution transparency) that might not be feasible in a lot of today's business scenarios. This section introduces alternative ways to accomplish transparent caching at the level of middleware. Since caching is a technically motivated aspect, its implementation should be the middleware vendor's responsibility.

Smart Proxies and Interceptors

It has already been outlined that *Smart Proxies* (see section 2.5.5 on page 72) and *Interceptors* (see section 2.5.5 on page 73) can be used in CORBA-based middleware to add user-defined behavior to remote object proxies. Caching functionality is only one possible application for these concepts. For instance, *OIF* [FBLL02] and *Fractal* [DL03] use client-side caching as an example aspect for their interceptor frameworks. The principle is simple: Every intercepted invocation is first checked against a local cache of invocation results on client side; server-side implementations are only invoked upon cache misses.

Stub Annotation

Java's RMI [Sun02a] and Java-IDL don't have any providences for this concept. Nevertheless, proposals have been made how to fill this gap: *Smart Stubs* [Lot00] is a simple proof-of-concept whose basic idea relies on renaming default RMI remote stubs and replacing them by derived, augmented classes that wrap functionality for caching and performance monitoring.

Smart Stubs

Since it's often undesirable to modify existing code—neither on clients, nor on server side—the stub classes themselves (i.e., the underlying wire protocol's proxy components for marshalling of remote calls) can be adapted and modified. Without altering the business interfaces or violating the wire protocol, the system's generated default stubs—in the case of Java RMI usually results of `rmic`—can be replaced by user-defined implementations. *Smart Stubs* wrap system-generated default stubs in especially adapted subclasses that add functionality for caching and performance monitoring. Command line tools are provided to assist the creation of these adapted stubs.

However, attribute caching is performed in a far too static way. No attention whatsoever is paid to the diversity in attribute usage. Every attribute is cached upon first usage and discarded uniformly in fixed intervals—a crude way to ensure update synchronization. We tried to implement a first prototype of our concepts as an extension of the Smart Stub approach, which is introduced in section 6.1 on page 147.

Caching Services

We already argued that caching is a technically motivated aspect, which is never directly related to functional application requirements. It rather emerges out of performance considerations taken at application runtime. Therefore it's only equitable to disburden application programmers of the challenge to implement caching logic by themselves. The caching aspect should be provided as a middleware service, which can either be used explicitly by application programmers or be configured descriptively during deployment without recoding[6].

In the CORBA world, middleware services are usually implemented as so-called *Common Object Services (COS)*, e.g., Naming, Trading, Transactions, etc. The need for a caching service led to projects like *Flex* [KAD96], a CORBA-based framework for client-side caching, and *Cascade* [CDFV00, Vit01], a CORBA service for object caching in Wide Area Networks (WAN).

Efficient RMI A similar tendency can be observed in the context of Java RMI [Sun02a]. For instance, Krishnaswamy *et al.* [KWB+98] implemented an efficient RMI implementation including a caching service. Later versions of the

[6] Cf. section 2.2 on page 14 for the discussion of *explicit* vs. *implicit / descriptive* middleware

prototype [KGDA00, KRB01] were implemented on top of BBN technology's *Quality Objects* (QuO) [ZBS97] framework instead to enable the fulfillment of Quality of Service (QoS) requirements regarding response times by using a caching service, which was integrated in a transparent manner via QuO proxies[7]. A *time-sensitive* consistency model allows to specify time-constrained consistency levels for different sites in a staggered fashion.

Eberhard and Tripathi [ET01] also present a middleware solution for transparent object caching in RMI-based distributed applications. Flexibility is provided with respect to consistency: A *client policy* is configured for client consistency managers with an *ActionsList* and other metadata. ActionsLists configure sequences of actions that are executed by the client consistency manager, e.g., local and remote method invocation, state update, etc. but also retrieval of further ActionsLists from the server consistency manager. Metadata includes a cacheability categorization into none, immutable, read-only, read-write, and server-only, which basically determines the behavior of the client policy. So-called *reduced objects* (cf. section 3.1.4 on page 92) are used for bundled transfer of object attributes, which saves network round-trips. Transparent integration is provided by byte-code manipulation of RMI stubs (cf. section 6.1 on page 147). Although the approach provides many interesting suggestions for our work, its major flaw is the ultimate dependence of its invalidation mechanism on client callbacks, which renders it impracticable for NAT / firewall scenarios.

General Purpose Software Cache Starting with *Data Update Propagation* [IC98], an algorithm relying on specified data dependencies between replicas and their origins for optimized updates of distributed caches from the background of IBM's DynamicWeb Cache system, Iyengar *et al.* developed a general-purpose software cache [Iye99], which can also be used out of the context of a specific middleware in other application domains, e.g., databases.

A similar endeavor has been started in the Java community with Java Specification Request (JSR) #107 [Bor01], incited by Oracles *Object Caching Service for Java (OCS4J)*. Meanwhile, a number of prototypical implementations evolved. Penchikala has compiled a comparison of recent implementations of this JSR in the context of J2EE (see section 2.2.4 on page 22) application servers [Pen04]. The relation to application servers is justified by their need to consistently cache data in the business tier and web tier across clustered servers. Some of the examined implementations also incorporate consistency protocols for transactional cache access. However, most prototypes belong to the category of *explicit* middleware, i.e., component/bean developers or Servlet programmers have to use these services explicitly in their code. Only the upcoming JBoss Cache could be used *implicitly* by interpreting an adequate description with JBoss' AOP framework [BB03].

While most of these approaches more or less exclusively address *explicit*

[7] QuO proxies are comparable to *Smart Proxies*, see above.

middleware, our goal is the *implicit* usage of such middleware services for caching, in accordance with a description provided by either application programmer, component developer, or application assembler / deployer.

Distributed Shared Objects

Quite a number of scientific publications in the past decade elaborated on *Distributed Shared Objects (DSO)*, a concept for transparent replication of objects. In contrast to traditional middleware like CORBA and DCOM (cf. section 2.2 on page 14), DSOs are physically distributed entities that encompass all replicas and proxies belonging to a logical object. In this respect, DSO models go beyond the scope of the above presented caching services, although many caching services already provide similar functionality.

Javanaise

Although it could be used as a *Caching Service* (cf. section 3.1.5 on page 95), *Javanaise* by Hagimont *et al.* [HL98, HB01] implies different semantics than this class of middleware services. It is based on Java RMI [Sun02a] and provides client-side caching (replication) of objects. Interdependent objects are grouped in so-called *clusters* for improved performance of updates. Javanaise provides the same interface as Java RMI, while extending its functionality. However, full access transparency can not be achieved on client side. The issues of managing multiple references to single objects are also discussed in [HB01]: A proxy-out on client side manages backward-reference parameter passing; a proxy-in on server side manages onward-reference parameter passing. All references to a certain cluster within another cluster point to the same proxy-out object. This concept closely resembles the scheme of stubs and skeletons, or client-side and server-side interceptors, respectively, although at the higher abstraction level of object clusters.

Globe

Globe by Bakker *et al.* [vSHT97, BAB+00] also builds on *Distributed Shared Objects* for controlling data replication on a per-object basis. A prototype called the *Globe Distribution Network* based on this middleware implements a service for global (wide area) distribution of data, e.g., software packages. *Globe* is adaptive to changes in data popularity and update patterns.

AspectIX

AspectIX by Hauck *et al.* [HBG+98], an aspect-oriented (see section 2.5.6 on page 75) and CORBA-compliant ORB architecture, introduces a concept named *Fragmented Objects*, which roughly resembles *Globe*'s concept of DSO. Both functional and non-functional properties (i.e., aspects) of objects can be configured by clients using a generic interface. Local fragment implementations of objects can be replaced transparently to fit configuration requirements.

In the strict sense, the *shared data-object model* behind the *Orca* programming language [BKTJ92] can also be categorized as a DSO model.

Unbound by the constraints of particular middleware or component platforms, most of these proposals managed to build efficient solutions for distributed applications. Unfortunately, hardly any of them managed to gain a broader base of acceptance apart from academic acknowledgment. However,

interesting cross-references can be drawn to the contribution of this work.

Client-side Caching of Method Results

Although some of the above mentioned proposals for caching services already tried to cache results of method invocations at the client side, e.g., [KWB+98, ET01], this section will concentrate on an approach that probably resembles our work most closely, Peifer's *Method-based Caching* [PJ03, Pfe04a, Pfe04b], which transparently caches method results of business interfaces in multi-tiered, J2EE-based applications.

The initial prototype [PJ03] integrates the caching logic at client side via generated proxy objects, which are transparently made accessible to clients over a manipulated naming service. Local cache consistency is preserved by checking *cache models* for method dependencies (i.e., invalidation relationships between different read/write methods) at runtime. Concurrent modifications of multiple clients are synchronized by invalidation messages, which the server propagates to clients using a callback interface. Although this solution is feasible for scenarios with only a few web servers as application server clients, it scales badly with for a growing number of clients of the business tier. An experimental evaluation based on the *RUBiS*[8] revealed promising results for unlimited cache size and using LRU as replacement strategy.

Cache models

An algebra for the introduced *cache models* [Pfe04a] formalizes their structure, correctness, and precision. Cache models are necessary to prevent the cache from serving invalid results. They help to identify read-only methods, i.e., those that do not alter to server's state, by capturing all read-write dependencies between business methods. Thus, specification of cache models is a tedious, non-trivial task for developers. In contrast, our approach in section 4.1 on page 107 follows the JavaBeans concept [Sun02b], i.e., component state is exposed by accessor / mutator method pairs, which can be recognized by their `get/set` naming pattern; additional invalidation dependencies can be captured as metadata.

Consistency

However, *cache models* alone can neither guarantee consistency of cached method results, nor convergence of different replicas. To provide a stronger level of consistency than the initial invalidation scheme, transactional method-caching was proposed in [Pfe04b], which was already mentioned in the context of section 3.1.3 on page 87. Apart from the three consistency protocols, an architecture is presented that implements the integration of the *m-scheduler*—a transaction scheduler for method operations—into the transaction management architecture of the JBoss open-source application server [FR03] on both client and server side. A theory for serializable *method cache histories*[9] is pro-

[8] *R*ice *U*niversity *Bi*dding *S*ystem [CMZ02, CCE+03], an *eBay*-like auction benchmark for performance measuring of multi-tiered Web application architectures. RUBiS is comparable to but different in focus than *TPC-W* [Smi01], a Web shop benchmark that tendentially overloads the database tier.

[9] i.e., representations of concurrent executions of transactions

posed, which considers cached results of method invocations in addition to common read/write operations of data elements, based on the concepts and terminology of Bernstein *et al.* [BHG87].

Due to complications with integrating the scheduler with the JDBC data manager, explicit `read()/write()` calls are necessary to inform the scheduler about access and manipulation of data. This poses a major violation of transparency since it requires either bean developers or container-managed persistence providers to explicitly use this interface.

The implementation hooks statically into the JBoss middleware architecture (cf. section 2.5.5 on page 73) with hard-coded modifications of the `RemoteMethodInvoker` instead of some self-defined Interceptor. The `RemoteMethodInvoker` is the last interceptor in the client chain, which finally transmits invocations to the server. Interceptors can be programmed and inserted descriptively by third parties or by application developers themselves for arbitrary middleware service functionality. Pfeifer's approach is hence less configurable in this respect than the solution presented in this work.

3.2 Communication Restructuring

Caching is only one solution to reduce the latency of distributed programs. As explained in section 2.1 on page 9, the basic principle relies on temporal and spatial locality of reference in data access program structures to reduce subsequent idempotent remote calls. Another approach is to execute remote calls *asynchronously*, i.e., without blocking the client, or to *defer* them at client side and execute them in a bundle as soon as one result is actually needed for further processing. These options have already been mentioned in section 3.1.3 on page 87 in the context of cache consistency protocols.

Asynchronous RPC

Liskov and Shrira introduced a language extension called *Promises* [LS88] for asynchronous invocation in Remote Procedure Calls (RPC) (see section 2.2 on page 14): RPCs do not block anymore, the runtime environment immediately returns a *promise* instead, i.e., a kind of proxy for the actual result. The client code can proceed while the result is computed and finally returned asynchronously by the server. Blocking only occurs if the client tries to actually use the result behind a *promise*; in this case it has to wait until the result arrives. Walker *et al.* proposed a similar concept called *Futures* [WFN90]. The *Rover* system by Joseph *et al.* [JdT+95] extends these concepts for asynchronous RPC in the context of mobile environments. Temporary disconnections can be handled by persistently queueing RPCs in log files for later execution. However, these concepts are not access transparent, i.e., they have to be used explicitly by application programmers.

Boxcarring

The above mentioned concept for deferring and bundling several remote invocations into a single one is also referred to as *batching* or *boxcarring*. It can be employed to reduce the number of required network round trips. Microsoft DCOM [MSd, MS DCOM technical overview] uses this technique extensively,

e.g., for object lookup, remote instantiation, or querying object functionality, among others. To reduce the impact on the programming model, batching code can be injected in custom proxies.

Yeung *et al.* [YK03, Yeu04] applied this scheme of call aggregation to Java RMI [Sun02a] in a fully automated fashion. They provided an augmented Java Virtual Machine (JVM) called *Veneer* that uses byte-code manipulation at load-time to intercept RMI calls and delay them as long as their results are not needed locally. In addition to that, the server may cache sequences of calls to speed up future invocations. This approach poses an interesting complementation of our work. A conceptual integration of these concepts at interceptor level might help to further improve the benefits of client-side caching of method results.

3.3 Prefetching

While the discussion of *caching* in section 3.1 on page 81 merely addressed the buffering of queried data for later reuse and the strategies for *communication restructuring* in section 3.2 on the preceding page tried to delay and bundle data queries, this section tackles the challenge of how and when to fetch data in advance, i.e., *before* it is actually queried. Since fetched data has to be stored locally to make it immediately available to future queries, *prefetching* can be seen as an extension of caching. While cache replacement algorithms (cf. section 2.1.2 on page 12) were limited by Belady's optimal strategy [Bel66], prefetching tries to tackle this boundary from the opposite direction by anticipating future events.

Danger of reduced scalability

The goals are almost the same as for caching: reduced apparent latency and increased user-perceived performance. Although prefetching can achieve better results in regard to these goals, its benefits come at the cost of increased bandwidth consumption, even in comparison to the plain non-caching implementation of a certain application: Because loading data in advance always implies some uncertainty of future events, a significant percentage of the bandwidth and server load is wasted to load data in vain. This drawback may also result in reduced scalability and throughput. Important metrics to measure the efficiency of prefetching algorithms are *precision* (i.e., percentage of subsequently requested prefetched objects) and *recall* (i.e., percentage of client requests that were prefetched) [RS02, Sect. 12.1]. Both values limit the maximum network utilization up to which no latency degradation occurs.

Speculation

Prefetching includes techniques for pre-population of client caches, both "server push" and "client pull" style. It is based on the assumption that the time between subsequent data request of a client (i.e., data processing time or user think time) can be used to load data in advance. User-transparent prefetching is almost always *speculative*. Non-transparent approaches require either preanalysis of data processing code or externally provided information about prefetchable data on client and/or server. Speculative techniques auto-

matically adapt to given access patterns but need an initial "ramp-up" phase for learning. On the other hand, non-transparent approaches become immediately effective but may operate inefficiently if not configured carefully enough. A combination of both is discussed in section 4.4 on page 125. Speculative techniques always follow a similar pattern:

1. Observe and record access patterns;
2. Recognize patterns later;
3. Predict subsequent accesses;
4. Prefetch (hopefully) needed data.

Implementation

Implementation alternatives for speculative techniques include *Markov chains* [Mar06] and *data compression* like *Lempel-Ziv (LZ)* [LZ77]. Both variants take the probability of subsequent events into account. Approaches based on Markov chains consider the probability for the next accessed data item as a function of last n items to fetch the most likely next item. Compression-based approaches model the frequency distribution of access to specific data items for the same purpose. Examples for such *prediction algorithms* in the context of Web prefetching are introduced in [RS02, Sect. 12.8].

The following sections discuss various application domains for prefetching and related approaches.

3.3.1 Prefetching in Database Management Systems

Speculative, forecasting-based prefetching strategies have already been used quite early in database systems for efficient buffer management [RR76, Smi78]. Speculative prediction algorithms like the ones mentioned above were used to load pages into the buffer prior to requests to avoid page faults.

In contrast, Wedekind *et al.* [WZ86, KWZ90] proposed a different strategy for canned transactions in realtime databases. Their concept is based on *preanalyzing* future data operations to determine and load a superset of all potentially relevant candidate pages beforehand. Preanalysis helps to determine access patterns of transactions, which can be used to determine potential subsequent requests. Queries are expanded to so-called *superset queries* by developing a precedence structure of involved database calls, identifying free query parameters, and binding a reasonably large amount of them at execution time as they become known. The challenge is to find the optimal time for issuing a superset query to limit the amount of response data while remaining effective.

Superset queries

Cache Groups [Tea02, HB04, BH04] have been discussed in section 3.1.3 on page 87 as a concept for constraint-based database caching, which tackles the issue of query containment by ensuring domain completeness of cached tuples. *Referential Cache Constraints* in Cache Groups also realize a form of prefetching since they force loading data that is presently not directly queried.

Cache Groups

Our concept for prefetching presented in section 4.2 on page 111 follows rather the descriptive approach of specifying prefetching dependencies in advance than the automatic operation preanalysis strategy.

3.3.2 Prefetching in Distributed File Systems

Distributed file systems (DFS) manage sets of distributed storage devices on different machines and aim to provide a global view for its users. Transparency is a major design goal, i.e., a DFS should have the same interface as its local equivalent. Caching is often employed to increase the user-perceived performance of file access. Most systems assume a low percentage of write accesses and do not cater for full transactional capabilities, i.e., they provide durability but not isolation. The cache granularity is relatively large: DFS cache either blocks or whole files. Many systems also provide prefetching functionality, e.g., to allow *disconnected operation* of clients in mobile environments.

Disconnected operation

Hoarding

For instance, *Coda* [KS91] was one of the first DFS to allow disconnected operations. It introduced a concept called *hoarding* for prefetching files during times of strong connectivity according to a user-defined list of required data objects (the hoarding database). This is obviously a non-transparent, non-speculative technique that requires user interaction to provide additional metadata.

In contrast, Kuenning's *SEER* [Kue97] provides *predictive hoarding* of files based on the *semantic distance* of files. This allows determining groups of files belonging, e.g., to a project. SEER dynamically analyzes user behavior to predict the current working set.

3.3.3 Web Cache Prefetching

Although the issue of using the time between two HTTP requests to preload data and decrease user-perceived latency is close at hand, most approaches are still at research level. The simplest transparent solution would be to analyze the structure of HTML documents and perform a complete traversal of all contained links. But simple eager prefetching of all referenced objects is sub-optimal. Hence, user preferences and service characteristics have to be kept in mind, which has also been realized by [SDMML03], among others. Rabinovich and Spatscheck's survey [RS02, Chap. 12] revealed that commercial solutions operate only client side since most effective server-side approaches would require changes to the HTTP specification or at least the agreement on a least common denominator.

A possible client non-transparent solution requires the specification of a "hot list", similar to user bookmarks, whose actuality is always preserved. The major Web browsers, Microsoft Internet Explorer and Mozilla, provide features called "site-crawling" or "link prefetching", respectively. Both re-

Site crawling

quire the insertion of additional tags in HTML documents concerning the next prefetchable links, which qualifies both solutions as server non-transparent *prefetching hints*. The Microsoft solution allows the optional specification of a *prefetching depth*, i.e., the number of levels that will be prefetched in the tree spanned by the links of an HTML document. Unfortunately, both solutions are incompatible with each other, which further limits their usefulness.

Other server non-transparent approaches include according to [RS02, Sect. 12.5.2]:

- Gathered access statistics made available to clients to judge relative popularity of objects;
- Centralized aggregation of client-submitted usage reports;
- Active pushing of likely objects to clients.

Bestavros has been among the first to research Web cache prefetching issues. He proposed a speculative, non-transparent, actively pushing scheme called *Server dissemination* [Bes95, Bes96], which requires the server to attach the objects that are most likely requested next to client responses.

Content Delivery Networks [DMP+02, RS02] have already been mentioned in the context of Web caching (see section 3.1.1 on page 84). But since the propagation of data among surrogate servers usually takes place prior to client requests, this concept also poses a form of prefetching in the strict sense. *CDN*

3.3.4 Prefetching in Distributed Object-oriented Systems

Although some of the systems relying on restructuring of communication patterns in section 3.2 on page 99 technically perform prefetching in a strict sense, only few publications exist on prefetching in distributed object middleware.

Brügge and Vilsmeier [BV03] propose a mechanism to reduce the user-perceived latency of remote invocations in CORBA by caching and prefetching results of remote method invocations. Cacheability of method results is configured statically by special tags in IDL source code. The same is done with the expiration time (TTL) of cached results on interface level. Integration is accomplished by means of manipulated client proxies. A history of method calls is logged per proxy object in advance to construct a dependency matrix of interface methods, which delivers probabilities for the invocation of certain methods in dependency of the previous invocation of some other method. If the probability exceeds 75 %, the depending method will also be executed by the server and its result will be appended to the previous invocation for insertion into the cache by the client proxy.

The approach is rather simple but it shows some interesting parallels to our work, e.g., concentration on distributed objects that encapsulate centralized state of some (database) entity or specification of cacheability and TTL in the

source code. The calculation of a dependency matrix prior to execution poses an interesting alternative to completely static specification and continuous dynamic calculation of prefetching dependencies in section 4.4 on page 125.

3.4 Summary

In this chapter, a number of approaches for reducing access latency in different application domains have been presented.

Starting with the possibilities and limitations of Web caching, we have shown that the trend is shifting towards dynamic content, which is increasingly hard to cache by conventional architectures. Edge Servers have been introduced in Content Delivery Networks to move the business tier (i.e., the data manipulating application servers) closer to clients and end-users. Flexible caching support is needed for finer grained data caching below the level of whole HTML pages. Our work represents a possible solution to this challenge because it addresses exactly this interface. In addition, solutions for piggyback (in)validation that closely resemble concepts from section 4.2 on page 111 have been presented. These approaches were complemented by various adaptive schemes, including an adaptive TTL heuristic that is also relevant for our work.

A number of parallels have been drawn to the field of database caching, where a selection of cache consistency algorithms were presented. Again, optimistic schemes using piggyback messages for invalidation seemed to pose an interesting complementation of our work in section 4.2 on page 111, since these approaches do not require client callbacks and thus confirm to our initial assumptions of client accessibility.

Various patterns and concepts at application level more or less require the application programmer's conscience of caching issues, which is why they have only been surveyed for possibly reusable aspects like bundled state transfer with State Objects.

We have presented a number of middleware mechanisms for transparent integration of services like caching, followed by numerous caching services, both explicitly programmable and configurable integrated ones. However, only few solutions in the context of multi-tiered architectures explicitly address caching at the interface between the business tier and presentation tier.

An approach for method-based caching in multi-tiered applications, which explicitly aimed at this gap, was presented next. Although the concept is supported by an established serializability theory for cached access to method results, transparency is partially abandoned at the level of component code. The configurability is inferior to our solution and the integration into the development cycle, which would enable a better end-to-end treatment of caching-related metadata, is also missing.

Communication restructuring and prefetching have been introduced as

two closely related alternative concepts for reducing access latency in a way more detached or independent from actual requests. Both concepts provide interesting prospects for further extending our solution. However, no solutions are known that directly combine predictive and specification-based prefetching. Section 4.4 on page 125 will address this issue.

We can thus summarize our survey of related work as follows:

- Caching at the level of proxy objects at the interface of business components running on application servers has hardly been addressed at all.
- Application-level solutions based on design patterns require massive code modifications on both client and server side and thus cannot provide our the required degree of transparency and flexibility.
- The majority of middleware-based solutions addresses only specific platforms, like RMI and CORBA. A general, platform-independent approach is missing.
- Reconfigurability of caching properties is addressed only by few (mostly aspect-oriented) prototypes. These in turn provide no continuous integration of caching issues into the software development cycle.
- Adapting cacheability properties at runtime has only been tackled by more remote examples, e.g., adaptive TTL in Web caching. However, this concept has not yet been applied to our abstraction level.

This summary augments the objectives and requirements from section 1.3 on page 3 and section 1.4 on page 5 with respect to the functionality we will design in section 4 on page 107 and implement in section 6 on page 147. A more in-depth comparison of our adaptive approach to related work is discussed in section 4.2.4 on page 119.

Adaptability is not imitation. It means power of resistance and assimilation.

Mahatma Gandhi (∗1869–†1948), Indian spiritual and political leader.

4

Design of an Adaptive Middleware Service for Caching

In this chapter, we will explain the general design of our middleware service for caching. For the static solution in section 4.1, which allows altering cacheability parameters only before deployment, component attributes or method results once considered cacheable remain in that state. We will show how we gradually enhanced this concept with respect to adaptivity and prefetching functionality.

To respect the demanded *validity probability* for cache invalidation and to consider changing access characteristics of component attributes and method results, heuristic combination of *adaptive expiration times* and *piggyback transfer* of metadata about invalidation and cacheability categorization will be introduced in section 4.2 on page 111.

The remainder of the chapter will furthermore introduce an implementation of *static prefetching* in section 4.3 on page 120, which is built on top of the adaptive caching solution. We will then elaborate an approach for *dynamic prefetching* in section 4.4 on page 125.

4.1 Static Caching

The original starting point for this thesis in general was an idea for automatic generation of caching logic in client-side stubs of Enterprise JavaBeans in combination with a mechanism for notification of clients upon updates, which was described in [NPF99]. The first prototype [PS02] used the concept of *Stub Annotation* (see section 3.1.5 on page 95) to integrate caching logic in client-side proxy objects. However, later conceptual extensions were based on the as-

sumption of an *interceptor*-enabled middleware platform (see section 2.5.5 on page 73). Details of these two alternatives for implementation will be discussed in section 6 on page 147.

Explicit middleware

The major goal is the explicit, separate handling of the orthogonal non-functional aspect "caching" throughout a component's lifecycle. This includes the tight integration into the software development cycle, which will be examined closer in section 5 on page 131.

Platform independence

Another goal is the design of a platform-independent solution, which differentiates our concept from various other related approaches presented in section 3.1.5 on page 94, e.g., [KWB+98, ET01, BV03] that focus on RMI or CORBA, respectively. Instead, we aim to provide a solution that is practicable for any request/response-based middleware with a notion of application components (cf. section 2.2 on page 14), as long as it provides a similar concept for interceptors or reflective access to the metalevel (cf. section 2.5.5 on page 71) by dynamic proxies or similar mechanisms.

Application specific consistency

An additional motivation for static configurability of the caching aspect was the realization that it allows for capturing and utilizing available knowledge of developers concerning the applicability of caching to component attributes and methods as well as the application's tolerance for inconsistencies at an early stage of software development, as already outlined in section 1 on page 1. Later publications [Poh03, PS03] complemented this static solution with adaptivity, which is presented in section 4.2 on page 111, and prefetching (see section 4.3 on page 120 and section 4.4 on page 125).

We have shown in section 2.5.1 on page 61 that software development is often an iterative process that reuses test results again for further loops of analysis, design, and implementation. Hence, adaptively determined configuration results can in turn represent initial values for a new iteration of the software engineering process. This notion is also shared by the Model-Driven Architecture (see section 2.5.2 on page 65), which introduced the challenge of "backward" transformations that allow for reverse engineering and model augmentation as a result of modifications to the implementation of an application.

4.1.1 Architectural Integration

In general, static caching is performed as depicted in figure 4.1 on the next page, based on the general architecture for additional services in component-oriented middleware in figure 2.13 on page 45:

1. A container-generated dynamic proxy implementing the desired component's home and remote interface is called from somewhere within the client application code.

   The proxy creates an `Invocation` object and passes this through the client-side interceptor chain where a `CachingClientInterceptor` is

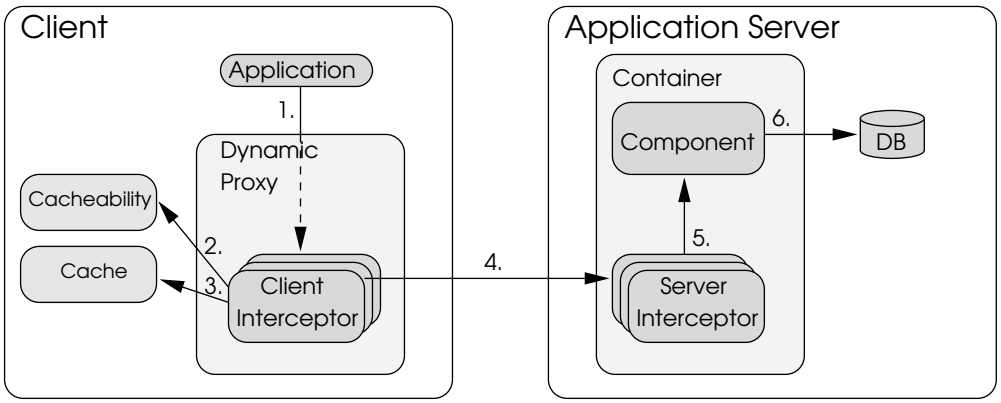

Fig. 4.1: Schema of static caching

installed to quickly answer invocations whose results it can anticipate from its cache contents.

2. The `CachingClientInterceptor` first checks its *cacheability database* for entries matching the current invocation. The cacheability database is a preconfigured client-local singleton[1] If the current invocation is `not-cacheable`, the invocation proceeds like every normal method call with step 4.

3. For `cacheable` or **`const`** invocations the *cache* is eventually checked for cached method results. The validity of existing entries is then checked using the `TTL` property currently configured in the cacheability database for the invocation. Invocations with `invalidates` entries, e.g., mutator methods of attributes, lead to the invalidation of cached results that possibly exist in the cache for their referenced methods.

4. If the invocation is not cacheable for some reason or if its result was not found in the cache, it will finally be transmitted to the server.

5. The invocation is handled by additionally configured server-side interceptors until it reaches the actual component instance (the *Bean*).

6. The component instance may in turn contact a back-end database to retrieve business data, depending on component type and state. For instance, in the case of EJB *CMP Entity Beans* the container transparently performs this step as needed before the actual business logic is executed.

The `CacheabilityDB` essentially stores the following data:

Cacheability. Categorization into `cacheable`, `constant` (read-only), and `not-cacheable`;

[1] Cf. section 5.3.3 on page 143 for the generation of the preconfigured `caching.xml`.

Validity. Maximum time-to-live (TTL) of `cacheable` entries; and

Invalidation. List of method signatures invalidated by the current method

Consistency

The TTL value is used by the `CachePolicy` to automatically invalidate a `CacheEntry` after the given amount of time to ensure weak consistency. Invalidation dependencies cause the interceptor to invalidate the specified other method results in the cache, similar to Eberhard's *ActionsList* [ET01] and Pfeifer's *cache models* [Pfe04a] introduced in section 3.1.5 on page 95 and page 98. Calls to `remove()` methods require the interceptor's special attention because they imply the removal of all cache entries for keys $(i_r, m, \{p\})$ with a given component identity i_r, method signature m and parameter list $\{p\}$.

4.1.2 Static Prefetching

Prefetching has been introduced in section 3.3 on page 100 as an extension of caching technologies that aims at further reducing user-perceived latency of data access by prepopulating client caches with data, which will probably be queried next in the near future. We consequentially applied this concept to our service for caching method results.

First considerations in this direction were presented in [PS02] and [PS03]. *Prefetching dependencies* have to be specified between methods with a close temporal locality of reference to allow additional method results to be fetched by the middleware service in advance. The combination with automatically generated *State Objects* (cf. section 3.1.4 on page 92) for fetching multiple attributes at once presented a promising approach.

However, the first fully functional prototype [AH03, Sect 2.3] was implemented on top of the dynamic, adaptive caching solution. The reason for this decision was the necessity for special considerations in the access statistics code of adaptive caching. It will thus be presented in section 4.3 on page 120.

4.1.3 Conclusion

Three questions remained unanswered in this section:

1. How can application designers *specify caching-related metadata*, i.e., concerning cacheability, consistency, and prefetchability of method results, in advance to allow continuous integration by model-driven approaches and generative programming mechanisms?

2. How can this *metadata be adapted at runtime* to "heal" misconfigurations and to gradually obtain optimal configurations of the middleware service?

3. How can the simple static time-to-live consistency mechanism be enhanced to *reduce anomalies* (cf. section 2.3.1 on page 46)?

The answer to the first question is given in section 5 on page 131, where the integration of this solution into the software development cycle is elaborated. The latter two questions are subject of section 4.2, which introduces a concept for adaptive reconfiguration of our caching service in combination with a heuristic for improved (but still *weak*) consistency based on piggyback validation/invalidation and adaptive expiration.

4.2 Adaptive Caching

So far, we have investigated the challenges of building a *statically configurable* middleware service for caching component attributes and method results at client side in section 4.1 on page 107. However, a number of open issues have been isolated, motivating the need for adaptivity of the presented middleware service at runtime: Up to now, the service does not explicitly respect the demanded *validity probability* for cache invalidation, nor does it consider changing access characteristics of component attributes and method results. These two challenges will be tackled in this section by implementing a heuristic combination of *adaptive expiration times* and *piggyback transfer* of metadata about invalidation and cacheability categorization.

4.2.1 Goals

An obvious disadvantage of the static approach described in section 4.1 on page 107 is necessity for component developers and deployers to precisely describe a component's cacheability properties before deployment without any chance of later interference. This drawback gave the motivation for our endeavors to extend the framework to dynamically adapt cacheability status of component attributes at runtime, i.e., whether a certain attribute or method result should be considered for caching or not. The primary goals of this approach are:

Adaptive cacheability categorization

- *Automatic correction of misconfigurations.* Component providers and cache advisors (cf. section 5.3.4 on page 145) (among others) may not find optimal configurations in advance or they can even make mistakes. These misconfigurations should be corrected automatically; the service should *heal* itself (cf. section 2.4 on page 53).
- *Automatic determination of optimal configuration.* Closely related to the aforementioned self-healing property is the goal of *self-optimization*.
- *Adaptation to gradually changing client access behavior at runtime.* Optimal configurations are not static; they tend to be subject to changing access

characteristics at runtime. If an attribute gets written or a method result gets invalidated more often than originally considered, caching may become inappropriate. Conversely, less frequent modifications may render caching useful again. Self-healing and self-optimization should also apply in this respect.

Validity probability

Another disadvantage of the static solution is the missing consideration of demanded application-specific validity probabilities, the specification of which was allowed by our UML Profile for Caching in section 5.2.2 on page 136. Current access patterns in terms of read/write ratio should be used to calculate an *adaptive expiration time* that reflects the two input parameters—validity probability and read/write ratio—for each cached data item. This adaptive time-to-live should be used to invalidate cache contents close to their actual modification time, estimated from past modification cycles. The above mentioned property of self-healing / self-optimization is thus extended to the TTL parameter, as well.

Adaptive TTL

In terms of our definition 2.13 on page 55, we can summarize these goals: While the static solution presented in section 4.1 on page 107 can be seen as an *adaptable system*, the dynamic extensions of this section aim at building an *adaptive system*, i.e., a system that is able to manage parts of its service functionality by itself.

4.2.2 Architectural Extensions

The key to achieve the above defined goals is to capture client-side data access behavior to derive information about *cacheability* and *invalidation* of component attributes and method results as we will explain later. We have developed a *distributed access statistic* for this purpose, whose details are subject of section 4.2.3 on page 114.

In the following, we will explain how this concept has been integrated into the static solution from section 4.1 on page 107 to build a dynamic, adaptive caching service.

It has been anticipated in section 2.5.5 on page 73 and figure 2.13 on page 45 that interceptors are also available at server side in quite a similar fashion, which allows exchanging arbitrary payload information between client and server by attaching the concerning data to regular method invocations. Pairs of interceptors on client and server side can thus be used to implement virtually any type of middleware service. Following this line of thought, our distributed access statistic described in section 4.2.3 on page 114 was also implemented in such a piggyback manner by chaining additional `CachingServerInterceptor`s in the component container's interceptor stack. How the gathered information is evaluated and eventually used to dynamically adapt the behavior of `CachingClientInterceptor`s behavior is shown in figure 4.2 on the next page as presented first in [Poh03, PS03].

Piggyback exchange

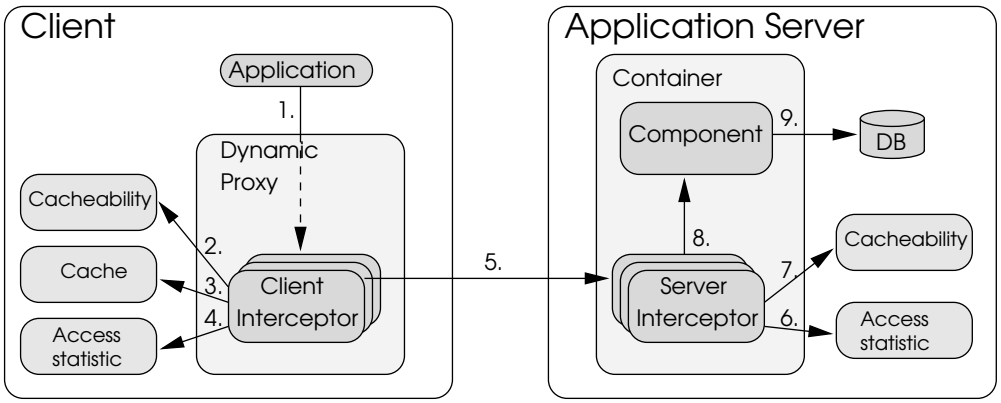

Fig. 4.2: Adaptive Caching Approach

1. A container-generated dynamic proxy implementing the desired component's home and remote interface is called from somewhere within the client application code.

2. The `CachingClientInterceptor` first checks its *cacheability database* for entries matching the current invocation. The cacheability database is a preconfigured client-local singleton[2] and continuously updated by dynamic adaptation as described below. If the current invocation is `not-cacheable`, the invocation proceeds like every normal method call with step 5.

3. For `cacheable` or **`const`** invocations the *cache* is eventually checked for cached method results. The validity of existing entries is then checked using the `TTL` property currently configured in the cacheability database for the invocation. Invocations with `invalidates` entries, e.g., mutator methods of attributes, lead to the invalidation of cached results that possibly exist in the cache for their referenced methods.

4. *Cache hits*, i.e., method results served from the client-local cache, are logged in the client-side *access statistic*.

5. If the invocation is not cacheable for some reason or if its result was not found in the cache, it will finally be transmitted to the server. Additional metadata, e.g., about access statistics, may be added to the payload of the invocation. On the server side, it will pass another interceptor chain containing the `CachingServerInterceptor`, which will evaluate this metadata. Submitted access data can be removed from the client's statistic after transmission.

6. The invocation and its possibly attached information about client-side cache hits are added to the server-side *access statistic*.

[2] So far, there are no differences to the static solution from section 4.1 on page 107; cf. section 5.3.3 on page 143 for the generation of the preconfigured `caching.xml`.

Adapting cacheability

7. Changes in the access statistic concerning the read/write ratio of attributes or method results, or their average time between modifications may necessitate adapting the TTL and hence the *cacheability categorization* of this type of invocations. The current TTL and cacheability setting of an invocation is always attached to the payload of its returning result.

8. The invocation is then handed off to the additionally configured interceptors of the server-side chain until it reaches the actual component instance (the *Bean*).

9. The component instance may in turn contact a back-end database to retrieve business data, depending on component type and state. For instance, in the case of EJB *CMP Entity Beans* the container transparently performs this step as needed before the actual business logic is executed.

For concision and better readability, the return path of invocations has not been explicitly marked in figure 4.2 on the preceding page. It basically follows the numbers in reverse order. Apart from the actual result object, additional information may be added to the payload of the `InvocationResponse`, such as adapted TTL values or cacheability categorizations. Back at the client side, this information will first be stripped off the response. The result is then stored in the client-side cache if the corresponding type of invocations is categorized as `cacheable` or **`const`**.

4.2.3 Distributed Access Statistic

Capturing client-side data access behavior has been identified as the key to derive information about *cacheability* and *invalidation* of component attributes and method results. We have developed a *distributed access statistic* for this purpose, i.e., a service that monitors access behavior at all clients and aggregates the gathered data at server side.

A pure client-side solution would not reflect the behavior of the whole user group. It would furthermore take more time to accumulate a significant amount of data and to react to changing access patterns. We opted for a distributed version where all clients collaborate with the server. The general procedure can be described as follows:

1. Capture access patterns, i.e., gather access statistics (cache hits, read-/write ratio, average time between modifications) at clients;

2. Accumulate and synchronize statistics at server;

3. Use in conjunction with application-specific required validity probability

   - for invalidation of client-side caches and
   - to determine changes of cacheability (categorization and TTL).

This implies a data flow as depicted in figure 4.3:

- In addition to normal data transfer between client and server (i.e., method invocations and responses), clients regularly inform the originating server of local *cache hits*, i.e., component attribute accesses and method results that have been successfully served by the client-side cache. The server needs this information for the correct calculation of read/write ratios. Write accesses (i.e., attribute modifications and invalidating method invocations) are never served by the `CachingClientInterceptor`; they are always transmitted to the server and thus do not need to be considered explicitly.
- The server in turn accumulates the submitted metadata to calculate the *average validity time* $\overline{t_{val}}$ of cached results for each potentially cacheable data item as the average time between modifications to this data item. Using the configured, application-specific *demanded validity probability* p_{val}, the *time-to-live* t_{TTL} can be adaptively derived for each data item. The current cacheability categorization can then also be determined by comparing t_{TTL} to the preconfigured threshold value ttlMin from section 5.2.2 on page 136.
- Clients may also send *validation* requests to the server and the server may *invalidate* cached data items as needed. We will elaborate this below.

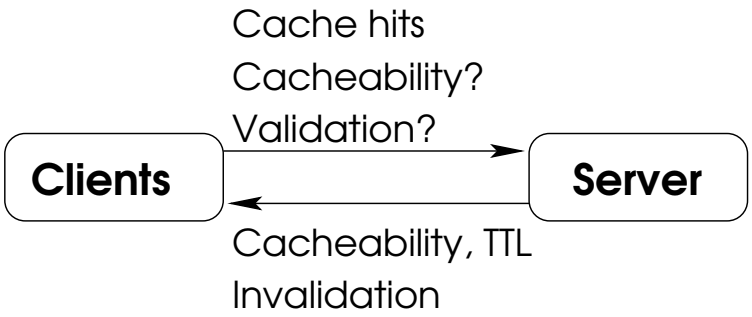

Fig. 4.3: General data flow of adaptive caching

Assuming a continuous, even distribution of validity times $\overline{t_{val}}$ and an invalidity probability given by $p_{inv} = 1 - p_{val}$, an estimation for t_{TTL} can be calculated by:

$$t_{TTL} = p_{inv} \times 2\overline{t_{val}}$$

which follows from p_{inv} as given by the ratio of the area of the rectangular triangle with the edges $t_{TTL} = \overline{t_{val}} - (\overline{t_{val}} - t_{TTL})$ and 1 to the area of the rectangle $(\overline{t_{val}}, 1)$ as depicted in figure 4.4 on the next page:

$$p_{inv} = \frac{t_{TTL}/2}{\overline{t_{val}}}$$

The demanded validity probability p_{val} can thus also be used to directly control the cacheability of method results: $p_{val} = 0$ is equivalent to the categorization **`const`** and $p_{val} = 1$ corresponds to `not-cacheable`.

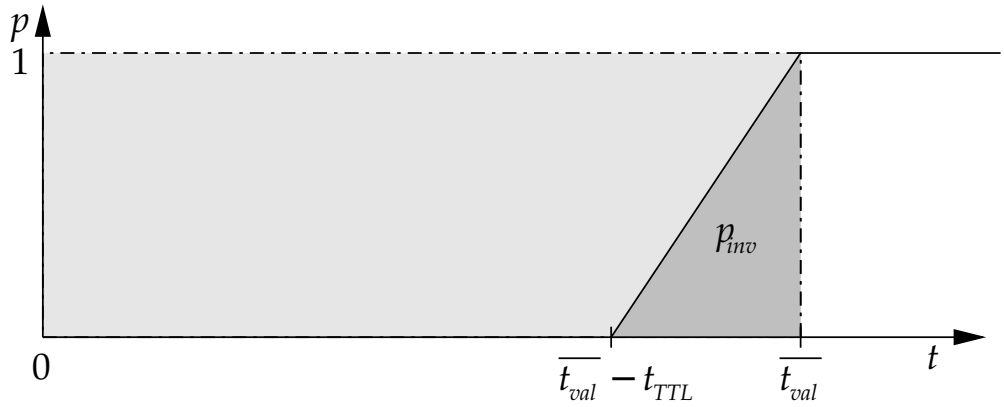

Fig. 4.4: Graphical interpretation of validity probability

Note that the exact determination of t_{TTL} would require an estimation of a confidence interval for the distribution of t_{val} and the demanded p_{val}. However, our heuristic neglects this approach because it would

- impose a much higher overhead for calculations and
- require exact knowledge of the actual statistical distribution of t_{val}, which is even harder to determine.

Optimization: Interval-based Access Statistic at Component Level

Instance-independent statistic

Based on the assumption of approximately equal access characteristics of all used component instances, we implemented an optimization that reduces the book-keeping effort at server side. Instead of keeping track of every read and write access to attributes or method results of each individual component instance, all accesses are aggregated at the metalevel, i.e., for every method invocation of the same type[3]. Time intervals have furthermore been introduced as a basis of access statistic collection instead of considering all accesses since the last server start. This is where the component-level tagged values intervals and intervalLength from section 5.2.2 on page 136 come into play:

[3] Cf. section 2.5.1 on page 58 for metalevels and component forms.

intervals denotes the number n_{int} of intervals to be taken into account for determining access characteristics;

intervalLength denotes the length t_{int} of these measuring intervals in seconds.

The *average validity time* $\overline{t_{val}}$ of cached results, which is needed to adapt the *time-to-live* t_{TTL} as explained above, can then be calculated from the number of involved component instances I and the number of write accesses m_{write} within a given measuring interval of the length t_{int}:

$$\overline{t_{val}} = t_{int} \times \frac{I}{m_{write}}$$

Hence, the access statistic governs a data structure per interval for every potentially cacheable method invocation, which comprises the number of write accesses m_{write} and a set of identities of invoked instances $\{i\}$ whose cardinality determines the number of involved instances: $I = |\{i\}|$. Lists of cache hits transmitted piggyback by clients are needed to effectively determine the set of globally used instances within an interval. At server side, the periodical evaluation of intervals and subsequent TTL adaptations are triggered asynchronously by a `Thread`.

Additionally Required Data Structures

The *access statistic* has been implemented as an extension of the *cacheability database* at client side and server side. In addition to method signature, cacheability, and TTL, a list of *cache hit counters* was inserted at client side. Counters are removed from the list after transmission to the server. At server side, method signature, cacheability, and TTL are augmented by the demanded *validity probability*, and a list of *interval statistics* whose entries consist of a set of component *instance identities* accessed during the corresponding interval and a *write access counter*.

Invalidation

One question remains unclear with respect to the above presented procedure: How are clients notified about server-side updates concerning component state and cacheability?

Earlier experiments with event-based publish-subscribe middleware in the context of the solution presented in section 6.1 on page 147 [NPF99] scaled poorly for increasing numbers of clients due to the tremendous amount of status data and connections the server had to govern.

Therefore, the decision was made for a client-driven "pull" strategy that relieves the server from the burden of direct "push" update propagation. To

avoid polling or "busy wait" of clients, the expiration time t_{TTL} is continuously adapted to the current average validity time $\overline{t_{val}}$ of cached results of a certain method. In this regard, the approach closely resembles the scheme of *adaptive TTL*, which was presented in section 3.1.1 on page 84 as a heuristic for cache expiration in Web proxies. However, our concept additionally takes the demanded validity probability of cached data into account.

Revalidation

The original strategy of our approach as presented in [Poh03, PS03] incorporated the feature of *revalidation*. It can be described as follows with reference to figure 4.2 on page 113:

- In addition to the average time between modifications $\overline{t_{val}}$ the server-side access statistic also keeps track of attribute and/or method result modification times t_{mod} by means of `invalidates` dependencies.

- When a cacheable method is invoked, the `CachingServer-Interceptor` attaches the time of last modification t_{mod} and the expiration time t_{TTL} according to the attribute's current average time span between changes $\overline{t_{val}}$ as additional payload to the returned `InvocationResponse` object, taken from step (8) and (9) of figure 4.2 on page 113. The current cacheability setting as explained in step (7) is also attached, accordingly.

- Back on client side, the `CachingClientInterceptor` updates the method result's cacheability categorization if necessary and schedules a `java.util.TimerTask` with the given expiration time. This TimerTask will enqueue the method results identity (i.e., signature, component instance identity, and method parameter values if applicable) together with its last modification time t_{mod} in a list of expired objects.

- The next remote call passing a `CachingClientInterceptor` picks up the value pairs from this expiration list and attaches them to the Invocation, thus preventing additional network traffic by this piggyback strategy.

- Unmarshaled at server side, the expiration list is compared with the last modifications in the access statistic, resulting in the creation of a positive list containing a bit mask for changed cacheability categorizations and value updates, which is transferred on the invocation's way back to the client.

- Modified attributes and method results are then discarded (*invalidated*) from the client-side cache, implicating a normal retrieval upon next access that causes the described procedure to start over again. The remaining expiration candidates are *revalidated*, i.e., reinserted into the cache. If an attribute turns out to be `not-cacheable` from now on, any of its possibly existing cache entries will also be discarded.

However, this strategy implies a tremendous book-keeping overhead since the last modification times have to be managed for every instance of a component attribute or method result. The above described instance-independent optimization of an interval-based access statistic at component level was thus introduced in [AH03] as a feasible trade-off between performance and consistency.

Assuming a small deviation of t_{val}, the amount of method results additionally invalidated by the *revalidation* scheme should remain rather small. At the same time, the savings achieved through preventing premature invalidations by revalidation will hardly compensate the higher costs for data management at server side, especially for fine-grained component attributes and method results.

4.2.4 Conclusion and Comparison

It has already been mentioned that the adaptive caching approach presented in this section closely resembles the *adaptive TTL* concept known from section 3.1.1 on page 84 in the context of Web proxy cache expiration. The success of this concept for caching data between client and presentation tier suggests its applicability between presentation and business tier as well for applications with weaker consistency constraints. Our approach additionally considers a demanded *validity probability* in combination with the current average time between modifications to dynamically adapt the time-to-live value.

However, more parallels can also be drawn to *Piggyback Cache Validation* (PCV) and *Piggyback Server Invalidation* (PSI) [KW97, KW98], which have also been successfully applied in the context of Web caching [RS02, Sect. 10.1.4, 10.2.6]. Furthermore, *Adaptive Optimistic Concurrency Control* (AOCC) [AGLM95] also uses piggybacking for cache invalidation messages in the context of database caching (see section 3.1.3 on page 87). Our approach uses piggybacking to transmit adapted TTL values and cacheability categorizations for cached method results and component attributes. Due to the usually smaller granulate of method invocations in comparison to Web page requests or database queries, piggybacking seems to be inappropriate for validation requests of clients as well as invalidation messages of server. The round-trip of a validation request takes nearly as much time as the corresponding method invocation for data retrieval because of the high marshaling overhead. Invalidation messages furthermore imply a book-keeping overhead at server side that seriously impacts scalability for growing numbers of clients, attributes, and component instances. These considerations were the driving force behind our aforementioned optimizations towards an interval-based access statistic at component level.

Although the achieved level of consistency is weaker than those of the database cache consistency protocols compared by Gruber *et al.* [Gru97], this classification can also be applied to categorize our solution as:

deferred because of the postponed piggyback messages;

lazy reactive because of eventual invalidation; and

avoiding because stale cache content is tried to be avoided in an best-effort manner by means of adapted TTL values.

In this respect, it is also comparable to AOCC, which has a similar categorization. The aspect of conflict *detection* is missing, because the solution was originally designed for application scenarios with less stringent consistency requirements. A fully transaction-aware solution could be built by adapting the concepts proposed by Pfeifer [Pfe04b]. This would however bear only little scientific value; it rather represents a pure engineering effort.

4.3 Static Prefetching

Prefetching has been introduced briefly as a possible extension of *static caching* in section 4.1.2 on page 110. Earlier experiments in this direction have already been presented in [PS02, PS03]. But we also mentioned that the first prototype [AH03, Sect. 2.3] was instead built on top of the dynamic, adaptive solution introduced in this section because of synergies with the implementation of the distributed access statistic presented above.

The major goal of prefetching is to speculatively load that data into the cache in advance, which might probably be needed in the near future. Hence, it tries to exploit the temporal locality of data accesses to an even higher degree than caching, which primarily considers spatial locality (see section 2.1 on page 9).

Prefetching dependencies

Speculative loading is performed in a transparent manner, i.e., without the client's notice, as we already realized in section 3.3 on page 100. The goal of the approach presented in this section is to reduce the amount of necessary speculation by preconfiguring *prefetching dependencies*, which reduce the initial ramp-up phase the service would otherwise need to decide which cacheable method results are particularly worth being loaded in advance.

Prefetching can furthermore be conducted in a *recursive* manner by traversing multiple levels of components. This is especially interesting if method results of a hierarchical network of component instances are to be prefetched. In this case, method results of previously prefetched component references are prefetched immediately after, e.g., in a first step all of a customer's orders and in a second step all product items contained therein. However, limiting the recursion depth is important to avoid request bursts that might have serious impacts on responsiveness and overall performance due to high load peaks. Prefetching requests should thus be deferred to relax these negative impacts.

In analogy to the extension of static caching (see section 4.1 on page 107) to the adaptive solution presented in section 4.2 on page 111, an approach for

dynamic prefetching will be presented in section 4.4 on page 125 as an extension of static prefetching.

4.3.1 Architectural Integration

In contrast to caching, prefetching necessitates the execution of data requests (i.e., method invocations) independent from the actual client application's control flow. While intercepting and augmenting client-initiated communications was sufficient for caching, prefetching requires additional methods to be issued by the middleware service.

In all these considerations, the requirements and design goals of section 1.4 on page 5 still have to kept in mind: Client server relationships must not be violated; piggybacking should be used as far as possible for data transfer to preserve the original control flow as far as possible.

If prefetching is implemented as an extension of *adaptive caching*, the access statistic should not register prefetching requests, which would otherwise distort actual access patterns. Therefore, prefetching requests must be marked as such. This also serves the purpose of limiting the recursion depth as explained above.

With these goals in mind, several alternatives for architectural integration are possible:

Server-side initiation. The `CachingServerInterceptor` automatically initiates invocations for method results to be prefetched and attaches their results to the initiating invocation's `InvocationResponse`. However, the server has no knowledge about already cached data of the concerning client, so far.

A naive implementation would transfer prefetchable requests with every invocation. To avoid this, the server could either try to keep track of cached data of all clients or clients could submit a list of their currently cached prefetchable method signatures in addition to each request. The immense memory consumption of the first approach would seriously limit scalability, while the second approach imposes a higher bandwidth consumption.

Client-side initiation. In contrast, the `CachingClientInterceptor` can also detect the necessity of prefetching with the help of preconfigured prefetching dependencies. It has naturally a much better overview of the contents of its client-local cache, allowing to skip those prefetching requests whose results are already cached.

To provide for a low user-perceived latency despite the higher amount of data to be transferred, prefetching requests are processed asynchronously by a Thread. Once invoked, such method results are marked as "in prefetching" in the local cache. When the client application tries to

access such data, it simply has to wait for the result to arrive or for a timeout to expire, which would then throw a `RemoteException`. This approach can be compared to the concepts of *Futures* [WFN90] and *Promises* [LS88] introduced in section 3.2 on page 99.

In turn, two alternatives exist for actually triggering the necessary asynchronous invocations:

Access via component interface. The least invasive possibility for issuing additional requests is to invoke the desired methods via the normal Home/Remote interface of the component. However, it is hard for the caching/prefetching service to distinguish between regular client invocations and additional prefetching requests since there is no possibility to mark requests as "in prefetching" at this layer. It is furthermore impossible to efficiently bundle prefetching requests in a simple way for the same reasons.

Insertion into interceptor chain. Instead of issuing all prefetching invocations at the dynamic proxy of the desired component, the `CachingClientInterceptor` can as well create `Invocation` objects by itself and insert them into the interceptor chain for regular processing. The process of method invocation is thus abbreviated for prefetching requests. It is furthermore possible to explicitly mark prefetching requests as such, easing the additionally required processing.

For reasons obvious from the discussion above, *inserting invocations into the interceptor chain* was chosen as the best alternative for implementation. The general principle of this approach is depicted in figure 4.5, which is actually an extension of figure 4.2 on page 113.

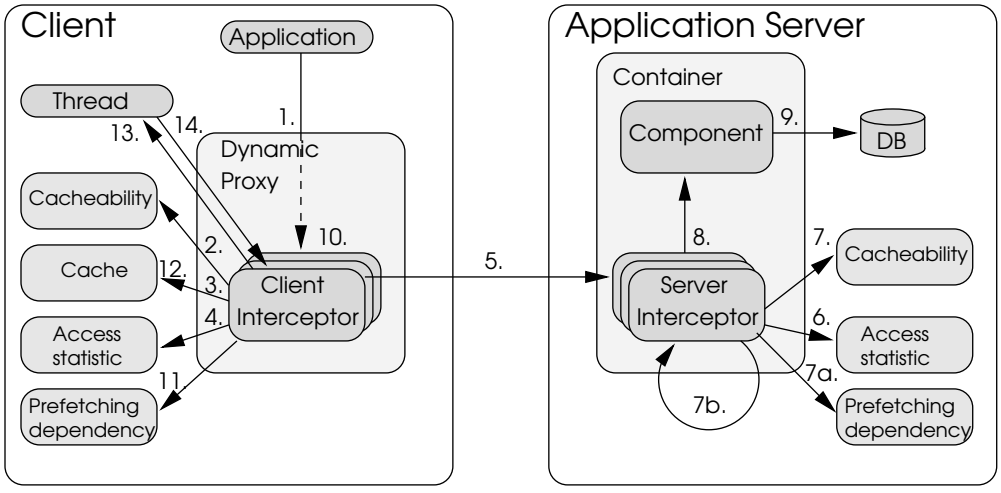

Fig. 4.5: Static Prefetching

The following description of the steps in figure 4.5 on the facing page just considers the *extensions* to the description in section 4.2 on page 111:

2. If an `Invocation` is marked as "in prefetching", proceed with step 5.

6. If an `Invocation` is marked as "in prefetching", proceed with step 7.

7a. If the client issued requests for *prefetching* dependencies, the corresponding reply is inserted into the `InvocationResponse` on the invocation's way back.

7b. If the invocation was issued for prefetching purposes, parameters for prefetching requests will be extracted from its payload. For each entry, an `Invocation` is created, submitted to the next interceptor in chain, and the returning result is inserted into the `InvocationResponse`.

11. After the invocation has returned to the client and its result data has been inserted into the cache, additional invocations are determined for another level of prefetching. The bail-out condition of this recursion is defined by the maximum depth of recursion.

12. Invocations determined as prefetching candidates are then checked for existing results in cache, in case of which the corresponding invocations are dropped. For the remaining invocations, a `CacheEntry` is created as a placeholder that marks them as "in prefetching" and counts down the number of recursions yet to be executed.

13. A Thread is created and supplied with a list of necessary information for generating invocations, i.e., `CachingClientInterceptor`, method, and parameters. An `Invocation` is created only for the first request to be prefetched; the remaining list is simply inserted into its payload.

    To enable a thread for reuse for multiple subsequent prefetching invocations, it gets returned a list of parameters for the next run of prefetching list determination, instead of the actual results of its invocations.

14. The Thread issues a single invocation, which is marked as "in prefetching". This invocation starts with step 2.

The details of the involved interactions on client side and server side are also depicted in the sequence diagrams in figure 4.6 on the following page and figure 4.7 on page 125.

Data Structures

In addition to the data structures of the adaptive caching solution, the cacheability database at client side and server side is extended to contain a *list of prefetching dependencies* for every cacheability database entry (i.e., for every registered method). Such prefetching dependencies consist of a *method signature*,

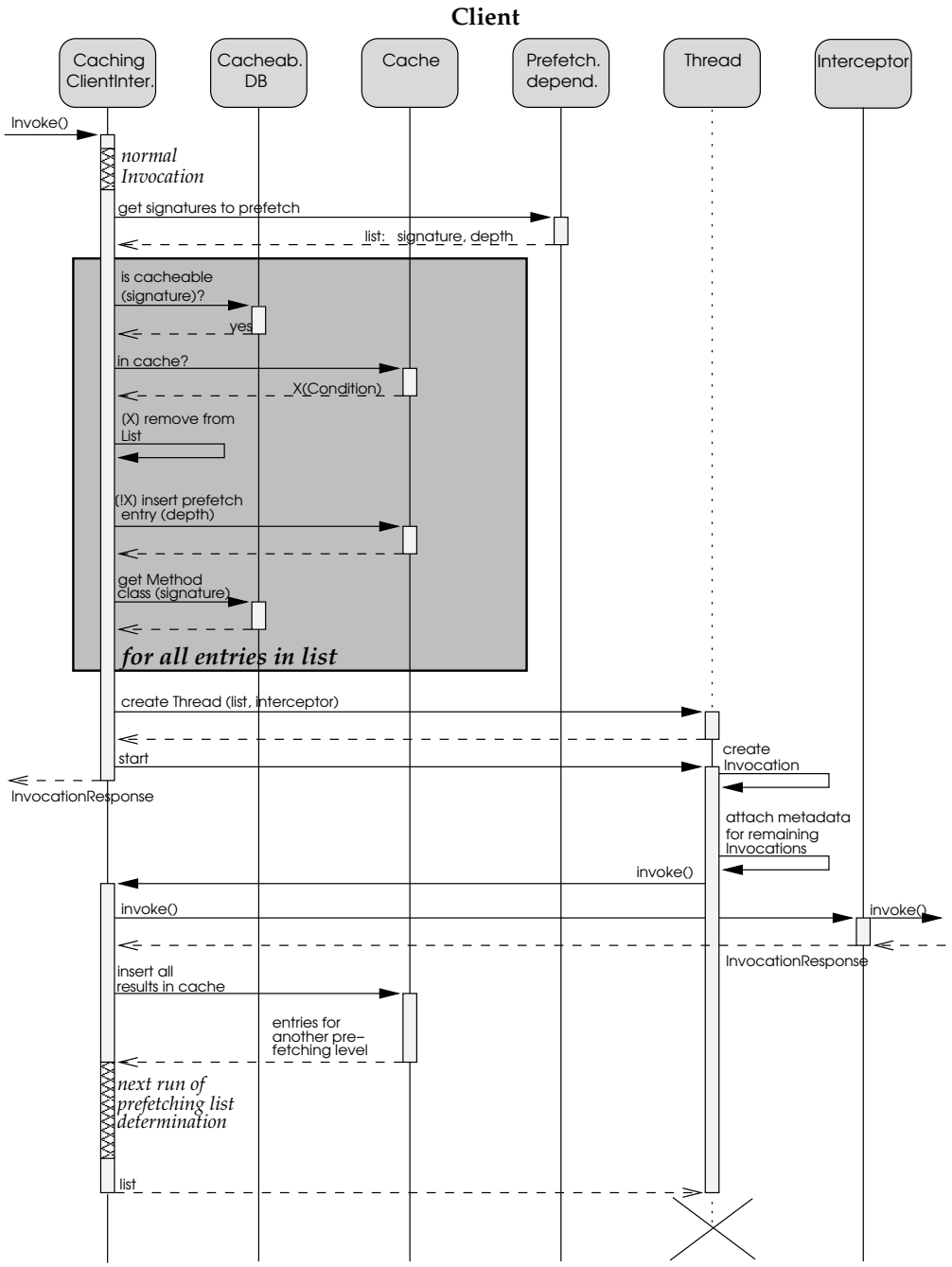

Fig. 4.6: Sequence diagram of Static Prefetching at client side

maximum prefetching recursion depth, and a *list of parameter specifications*, which in turn comprises a number of method signatures.

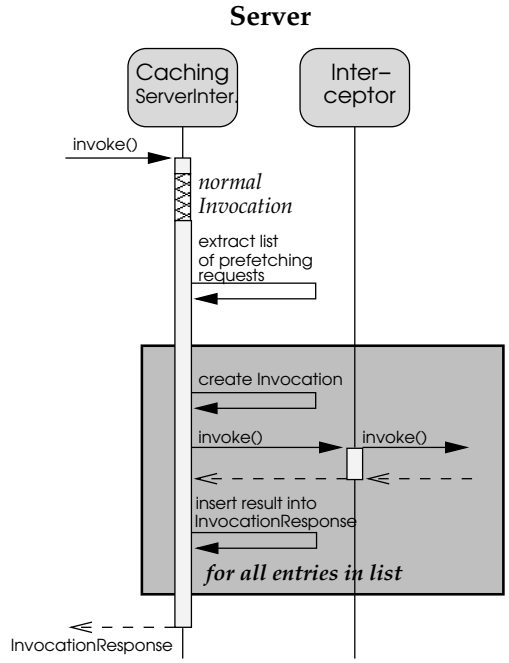

Fig. 4.7: Sequence diagram of Static Prefetching at server side

4.3.2 Conclusion

We have demonstrated in this section how prefetching can be implemented as an extension of our middleware service for caching. Prefetching dependencies between method invocations can be modeled in a fashion that closely resembles the capturing of cacheability properties at design time. The necessary modeling extensions will be presented in section 5 on page 131. Again, our experiments with the prototypical implementation of the framework's prefetching extensions showed the general feasibility of the conceived approach.

Prefetching dependencies may however be influenced by changing access characteristics at runtime. Misconfigurations may also occur during the modeling phase. Hence, concepts are needed to *heal misconfigurations*, *find optimal configurations*, and to *adapt to changing access characteristics*. These issues will be discussed in the next section.

4.4 Dynamic Determination of Prefetching Dependencies

In section 4.2 on page 111 we argued that *cacheability properties* are dynamic features of components, which are subject to changes at runtime. This gave the motivation for *adaptation* of these properties to current access characteristics. In turn, the same dynamic also applies to the *prefetching dependencies* introduced in section 4.3 on page 120 as a means for static modeling of tem-

porally correlated access dependencies between method invocations. The process of modeling these dependencies is also prone to misconfigurations and dependencies themselves may gradually change with access characteristics at runtime. First considerations in this direction were presented in [PS03]. A first design at a more concrete level was drafted in [AH03]. The general goal is to *dynamically determine prefetching dependencies* at runtime.

Adaptive prefetching

Much in the same way that *static prefetching* (section 4.3 on page 120) was an extension of *adaptive caching* (section 4.2 on page 111), *adaptive prefetching* will be presented in this section as a direct extension of static prefetching. However, while the concepts and approaches have already been implemented prototypically, *adaptive prefetching* exists only as a conceptual design at the time of writing.

4.4.1 Architectural Integration

Client-dependent, instance-independent statistic

The first issue to be solved is the extent of data to be recorded for an optimal coverage of prefetching dependencies. To be able to recognize temporal dependencies between subsequent method invocations of a single client, *client-dependent data recording* is mandatory[4]. Hence, embedding the necessary statistic into the `CachingClientInterceptor` seems to be the most appropriate design decision.

In section 4.2.3 on page 116, an optimization has been introduced for limiting the amount of required memory and computing time by implementing the access statistic independent of component instances, based on the assumption that access behavior is similar among different instances of the same component. It is sensible to extend this optimization to the aspired prefetching statistic, as well.

To further limit the amount of data to be stored, recorded information should be retained only as long as required. Therefore, it is necessary to give a threshold value t_{pref} for the maximum time between two method invocations to be considered for prefetching. A good heuristic seems to be $t_{pref} \leq t_{TTL}$.

Finally, a threshold value p_{pref_min} needs to be given for the probability $p_{pref}(m_1, m_2)$ that the *dependent* method m_2 is invoked after the *initiating* method m_1 within t_{pref}. If $p_{pref}(m_1, m_2) \geq p_{pref\_min}$, m_2 should be prefetched when m_1 gets invoked. We thus need to monitor p_{pref} for all combinations of method invocations at runtime. To limit the complexity, the current prototype considers only dependencies between invocations of the same component type.

The general architecture depicted in figure 4.8 on the facing page is a consequential extension of figure 4.1 on page 109, figure 4.2 on page 113, and figure 4.5 on page 122. This implies an implementation based on the concept of inserting additional invocations into the interceptor chain. Just like before,

[4] in contrast to unified, client-independent treatment of all accesses

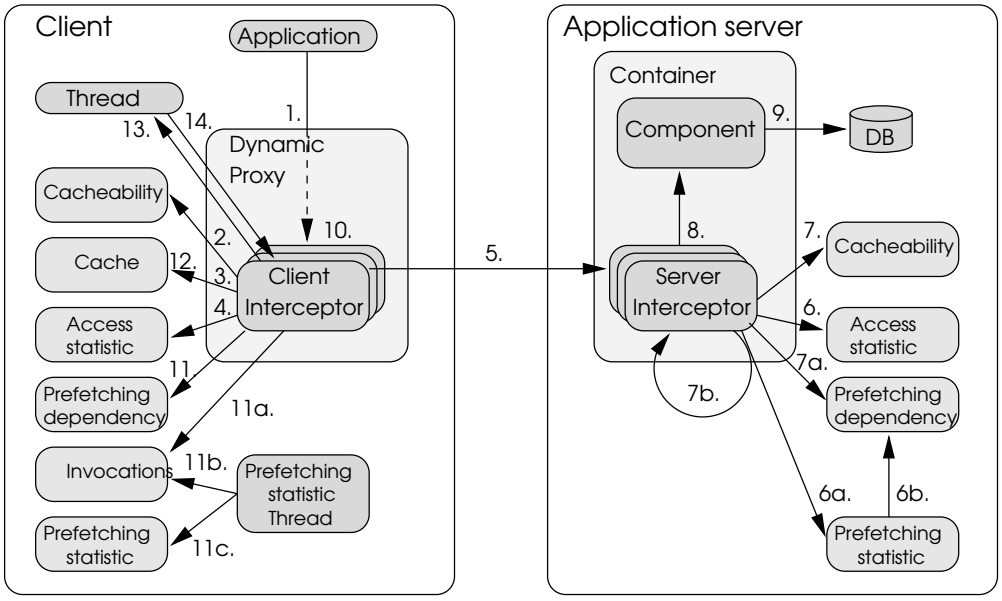

Fig. 4.8: Adaptive Prefetching

we will only highlight the extensions added in this section:

5. New values of the *prefetching statistic* are periodically inserted into invocations as piggyback payload for updating *prefetching dependencies* at server side.

6a. Values from client-side prefetching statistics are inserted into the server-side prefetching statistic.

6b. Prefetching dependencies are continuously updated with changing prefetching statistics.

11a. After an invocation has returned to the client, it is inserted together with its result (for traceability purposes) into the list of *Invocations*.

11b. The *prefetching statistic thread* periodically queries the list of *Invocations* for new entries and analyzes them accordingly.

11c. Processed entries are inserted into the *prefetching statistic* and *prefetching dependencies* are updated if required. The prefetching statistic stores a list of subsequent invocations together with a frequency counter for every possible invocation.

The above described procedure can be compared to the approach of Brügge and Vilsmeier [BV03], which was discussed in section 3.3.4 on page 103:

1. Every invocation is logged together with its time of occurrence.

2. The counter of the *initiating* invocation's signature is incremented.
3. It is checked whether another *initiating signature* accepts the invocation as *dependent signature*. Therefore, a reverse map from *dependent* to *initiating* is appropriate.
4. It is checked for all found signatures if the current invocation could have been prefetched from them. In this case, the corresponding counter of the *dependent signature* is incremented at the *initiating signature*

However, this approach is not yet capable of detecting prefetching dependencies across multiple levels of invocations. This would require tree-like data structures and a much more sophisticated processing logic for the prefetching statistic.

To integrate the conceived concepts for adaptive prefetching into the software development cycle, additional *tagged values* are needed for the maximum inspection time t_{pref} and the minimum prefetching probability p_{pref_min}. Consequentially, additional *XDoclet* tags need to be introduced as well as new elements of the `caching.xml` file.

4.4.2 Performance Considerations

The adaptive prefetching approach presented in this section has not been fully implemented. However, an initial analysis of the required data structures and processing logic suggests a comparatively high overhead in terms of memory consumption and processing time. Hence, it seems only sensible to use this concept rather during a *learning phase* to find optimal static configuration of prefetching dependencies (cf. section 4.3 on page 120). The results of this initial *learning phase* should then be exploited to configure the actual runtime environment, based on the assumption that access patterns are comparatively stable with respect to prefetching dependencies.

4.5 Conclusion

In this chapter, we have presented a number of extensions to the initial concept of a statically configurable caching service in section 4.1 on page 107.

Adaptive caching in section 4.2 on page 111 aimed at healing misconfigurations, finding optimal configurations, and adapting to changing access characteristics with respect to cacheability properties by keeping track of average validity times of cached method results. These values were considered for adaptively configuring cache expiration times and cacheability categorizations based on demanded validity probabilities.

This solution for adaptive caching was extended in section 4.3 on page 120 to allow *static prefetching*. Prefetching dependencies were statically configured

between methods and attribute accessors. Recursive prefetching was enabled by mapping input and output parameters to invocations of the next level.

Finally, considerations towards *adaptive prefetching* have been discussed in section 4.4 on page 125. In analogy to adaptive caching, a prefetching statistic is needed to dynamically keep track of dependencies between method invocations. The two delimiting parameters, maximum inspection time t_{pref} and minimum prefetching probability p_{pref_min} need to be adjusted carefully for optimal operation of the adaptive prefetching service.

Besides the complete implementation of the adaptive prefetching approach, the thorough quantitative evaluation of the implemented service functionality is still left as an open issue. Furthermore, the software engineering support needs to be optimized with respect to the configuration of prefetching metadata.

The engineer's first problem in any design situation is to discover what the problem really is.

5

Software Development Cycle Integration

The static configurability of caching aspects has been assumed for the middleware service for caching proposed in section 4 on page 107. However, means for capturing the corresponding metadata have been isolated as an open issue. Application developers can naturally enrich application models with valuable information at design time and thus can descriptively configure the middleware service in advance. This is because they already have a fair notion about the consistency requirements as well as cacheability properties of their application components. This knowledge should be used as far as possible to avoid the waste of resources during an initial learning phase for determining cacheability[1] on one hand as well as to exploit known tolerances to a limited amount of inconsistency during application design and at runtime, on the other hand. These observations led to the requirement for software engineering support for caching.

First ideas for integrating the management of caching-related metadata into the software development cycle have already been presented in [PS02]. A concept was proposed for representing cacheability categorizations of component attributes by using *Stereotypes* as standard UML extension mechanisms during application design with UML. This general approach still forms the fundamental concept behind the proposed extensions in this chapter.

The remainder of this chapter will explain the integration using state-of-the-art modeling and design techniques from section 2.5 on page 58. The overall development process is explained in the context of a model-driven tool chain in section 5.1 on the next page. The lightweight UML extensions necessary for controlling model transformations are summarized as an *UML Profile*

[1] Similar to the mechanism for continuous adaptation towards an optimal cacheability configuration for current access patterns, which is presented in section 4.2 on page 111.

for Caching in section 5.2 on the facing page. The application of this profile in the context of the proposed development process is discussed with the help of examples in section 5.3 on page 139.

5.1 Model-driven Development

The *Model-Driven Architecture (MDA)* introduced in section 2.5.2 on page 65 proposes the iterative application of model transformers to generate more detailed target models from abstract source models for further refinement. We have extended and adapted this process as depicted in figure 5.1 and explained below.

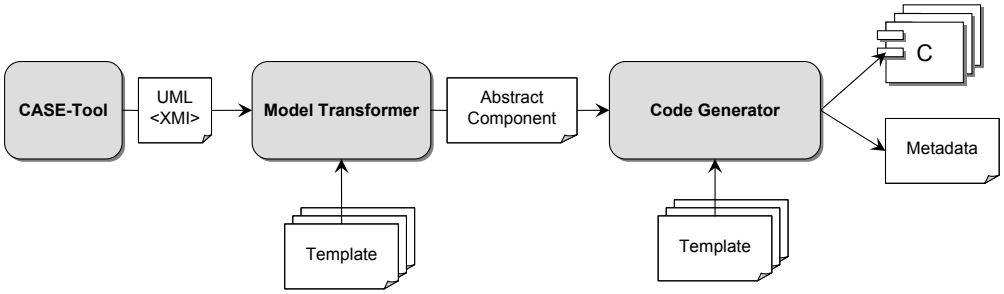

Fig. 5.1: Model-driven tool chain

A substantial part of application design is typically done using UML (see section 2.5.1 on page 58) models that capture the functionality, structure, and behavior of application elements, e.g., components. Tools for *Computer-Aided Software Engineering (CASE)* help design UML models in a graphical manner. *Stereotypes*, *Tagged Values*, and *Constraints* can be used as lightweight extensions of the UML metalevel to capture additional metadata within for certain model elements. A set of such extension mechanisms for a specific purpose can be summarized as a *UML Profile* like the one presented in section 5.2 on the next page for caching. Profiles can be used to describe constraints for valid *Platform-Independent Models (PIM)* and *Platform-Specific Models (PSM)*. Most CASE tools (like *Poseidon*, *ArgoUML*, *Together*, or *Rose*) allow to export UML models using the *XML Metadata Interchange (XMI)* format [OMG03e], which can be read by other tools.

PIM/PSM

In our case, the model transformer tool *AndroMDA* [Boh04] processes XMI files containing appropriately annotated UML models to produce *abstract*[2] component implementation classes according to selected target templates, which compose *cartridges* in AndroMDA. Cartridges are available for various component platforms like EJB or *Hibernate*. Their contained templates determine the predefined frame of how target component implementations look like corresponding to the source model. An AndroMDA template basically contains the

AndroMDA

[2] i.e., not yet compilable

code frame of a specific component type, e.g., an EJB CMP Entity Bean, intermingled with special code of a proprietary, macro-based language for dynamically inserting component segments according to the state of certain source model elements. We have extended the templates of AndroMDA's EJB cartridge to additionally process our profile extensions from section 5.2.

XDoclet

For most templates of AndroMDA's cartridges, the mentioned *abstract components* contain source code comments readable by the code generator *XDoclet*. The approach of generating all necessary interfaces and auxiliary classes from the single source of an abstract component class has been explained in section 2.5.3 on page 67. XDoclet's extension mechanisms have also been introduced in this context: Extending *XDoclet Template (XDT)* files, inserting additional static code snippets at specific *merge points* of XDT files, and programming custom *tasks*. We have developed a custom XDT file for producing the additional `caching.xml` file required for statically configuring the middleware service of section 4 on page 107 or the initial configuration for the adaptive middleware service presented in section 4.2 on page 111, respectively. Other alternative abstract component target formats for *attribute-oriented programming* are imaginable. For instance, a representation based on Java 1.5 annotations might obsolete the step of code generation altogether.

The overall integration of the mentioned tools is provided by means of the *Ant* build tool as explained, e.g., in [HOV04]. In the following, we will summarize our lightweight UML extensions for modeling caching properties of components as a UML profile in section 5.2, followed by a step-by-step explanation of the sketched development process in section 5.3 on page 139.

5.2 UML Profiles

We have already outlined that *UML Profiles* can be used to constrain valid *Platform-Independent Models (PIM)* and *Platform-Specific Models (PSM)*. Since caching method results of Enterprise JavaBeans clearly is a platform-specific matter, we need to provide a PSM for describing correct models that capture caching-related metadata. Figure 5.2 on the following page shows possible ways towards such a profile.

The UML standard facilities for modeling classes and components [OMG03c] are adapted to the specific needs of component-oriented business applications by the *UML Profile for Enterprise Distributed Object Computing (EDOC)* [OMG02b], which represents a PIM itself. Alternatives for a PSM for EJB have already been discussed in section 2.5.1 on page 64, including the dilemma of discontinued support for the EDOC to EJB binding in [Gre01].

AndroMDA follows a slightly different approach: A more general EDOC-like profile, which will be discussed in section 5.2.1 on page 135, is used to directly generate abstract component code for the selected target platform as explained in section 5.1 on the preceding page. This *abstract component* can be envisioned as a kind of *PSM* although it has already passed the transition

PSM

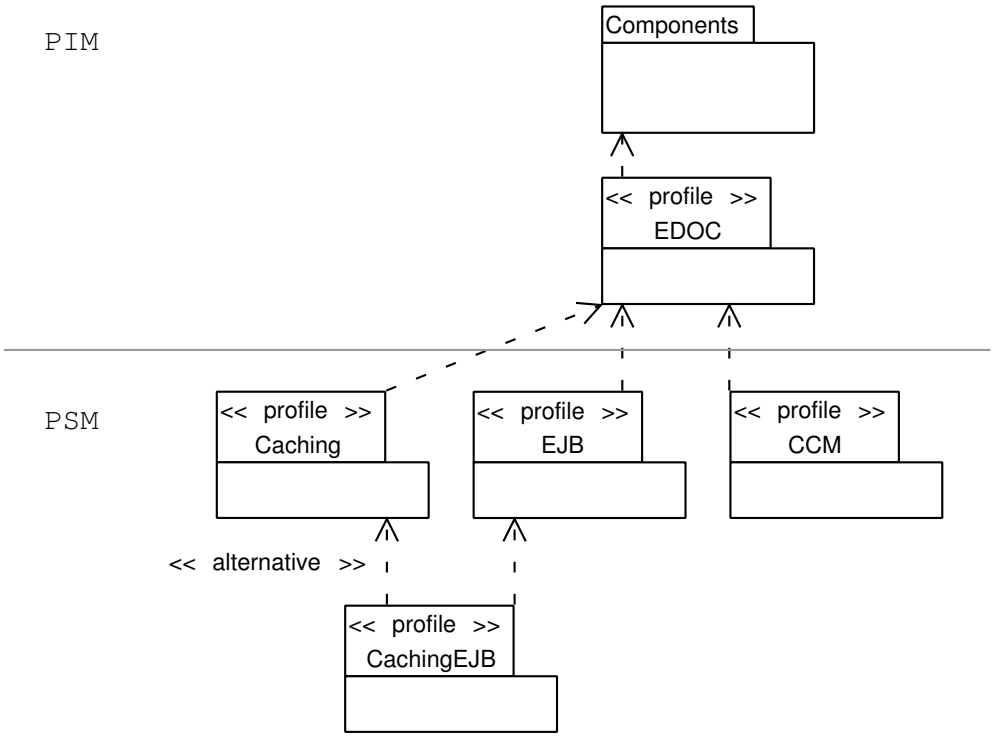

Fig. 5.2: Possible relationships between profiles

from model to code. However, this is not a conflict since code is one possible representation of an application's model[3].

Since the situation of UML Profiles for EJB is somewhat ambiguous for the reasons explained in section 2.5.1 on page 64, the pragmatic decision was to extend AndroMDA's generic profile. These extensions will be explained in section 5.2.2 on page 136. A positive side-effect of this decision is the potentially greater code reuse of cartridges and templates. With respect to figure 5.2, we thus opted for the left alternative. The ≪profile≫ CachingEJB only exists indirectly: It is the metamodel of all *abstract components* appropriately annotated with caching metadata. This is an intermediary representation, which is not meant to be modified by developers. Hence, we refrained from specifying a profile for it.

Representation and exchange of UML Profiles between different models and tools is still an open issue. At the time of writing, profile elements have to be inserted into models by typing their names into modeling tools like *Poseidon* prior to later assignment to model elements. A standardized exchange format would help to increase robustness of our development process due to the better type-checking support.

[3] For instance, tools like *Together/J* generally used to map all model elements 1 : 1 to code segments.

5.2.1 *AndroMDA* Profile

The philosophy of *AndroMDA* [Boh04] is to save one step of model transformation by directly mapping the results of PIM to PSM transformation to its representation in source code. It has been mentioned that a EDOC-like profile is used to constrain valid source models from which target implementation frames, e.g., for EJB, can be generated. The key elements of this profile are listed in table 5.1. Such a tabular notation has become accepted for describing profiles according to common examples [Gre01, OMG03c].

Simplified EDOC Profile

Tab. 5.1: Stereotypes of AndroMDA PSM according to [Boh04]

Stereotype	*Base Class*	*Description*
≪Entity≫	Class	business object or domain object that knows much and does little
≪Service≫	Class	dynamic element or activity that does something to the entities
≪PrimaryKey≫	Attribute	mark attribute of entity class as primary key so that the entity can be found in a database system
≪FinderMethod≫	Operation	associate a method of an entity class with a database query to find instances of this entity
≪EntityRef≫, ≪ServiceRef≫	Dependency	make class A depend on entity/service B if a template/cartridge can process this information
≪Exception≫	Dependency	make class B an exception thrown by all business methods of entity/service class A

In addition to this simple profile, AndroMDA defines three constraints that are dictated by the implementation of the default templates:

1. Stereotypes drive code generation. The Stereotypes from table 5.1 have to be used as "labels" that can be attached to application model elements to control the generation of *abstract components* and parts thereof.
2. Primary keys of entity components must be of the type `String`. This is because AndroMDA follows the philosophy of generating artificial primary keys, which allows for more robustness of the data model based on the assumption that requirements for natural keys often change during development.
3. One component has one `Exception`. This exception class will be thrown by every business method of the component. Other exceptions should be wrapped by this component-specific one.

As part of the profile, a mapping is defined from stereotypes of table 5.1 to

Possible transformations

elements of the metamodels of Enterprise JavaBeans, Hibernate, BPM4Struts, and Web Services, among others. Such mappings convey a significant part of the actual semantics of UML Profiles. Mapping AndroMDA stereotypes to EJB is rather self-explaining: Entity Beans are generated from classes tagged as ≪Entity≫, Session Beans from ≪Service≫, respectively. The reason why AndroMDA uses classes instead of components as base classes of its component stereotypes is the tight relationship to XDoclet's one-source concept, i.e., all additionally necessary auxiliary classes and interfaces of a component are generated from its main class.

5.2.2 UML Profile for Caching

Our UML Profile for Caching has been specified as a direct extension of the *AndroMDA* profile in section 5.2.1 on the preceding page for the reasons explained above. Although it has only been validated for the Enterprise JavaBeans platform, the generic nature of its parent profile would also allow transformation to other target platforms like *Hibernate*. Table 5.2 on the next page lists used *stereotypes*, which extend the metamodel, and table 5.3 on the facing page further defines *tagged values* that are useable in connection with these stereotypes. *Constraints* have only been specified implicitly within the descriptions on stereotypes and tagged value definitions.

The stereotype ≪caching≫ was just introduced to tag components as generally appropriate for caching and above that, to allow specifying default values for caching-related properties of component attributes and methods.

While the use of the stereotypes is rather self-explaining, the tagged values in table 5.3 on the next page might require some further remarks:

- ttl is short for time-to-live and represents the maximum time a result should remain in cache, i.e., its expiration time until validation (and possibly invalidation) is necessary.
- validity denotes the demand for a specific probability for validity of cached results, which results in adapting the ttl of the corresponding item depending on its access characteristics. This concept has not been used in the static approach presented in section 4 on page 107. It is part of the dynamic/adaptive solution, which will be introduced in section 4.2 on page 111. If ttl exceeds ttlMin, the corresponding attribute or method result is considered to be ≪cacheable≫, or ≪not-cacheable≫ otherwise.
- intervals and intervalLength are also related to the dynamic/adaptive solution. They can be used to specify the number of measuring intervals and their length in seconds on a per-component basis. Access statistics for dynamic (re)configuration of cacheability categorizations periodically check read/write ratios of component attributes and method results within the boundaries of these two tagged values.

Tab. 5.2: Stereotypes for caching

Stereotype	*Base Class*	*Description*	*Tagged Values*
≪caching≫	≪Entity≫, ≪Service≫	tags a component as appropriate for caching and allows to specify default values for cacheability properties of cacheable component attributes using tagged values	ttl, validity, intervals, intervalLength
≪const≫	Attribute, Operation	results of this component method or component attribute's `get` method practically never change (idempotent operation); to be cached upon first access	invalidates
≪cacheable≫	Attribute, Operation	results of this component method or component attribute's `get` method change rarely; should be invalidated every ttl seconds to ensure the required validity probability	ttl, validity, invalidates
≪not-cacheable≫	Attribute, Operation	volatile attribute or method that is subject to frequent changes or that should only be accessed in a transactional context	invalidates

Tab. 5.3: Tagged values for caching

Tagged value	*Stereotype*	*Description*
invalidates	≪const≫, ≪cacheable≫, ≪not-cacheable≫	list of signatures of method results invalidated by the corresponding method
ttl	≪caching≫, ≪cacheable≫	maximum time until a cached result expires and must be validated
ttlMin	≪caching≫, ≪cacheable≫	minimum ttl for an item to be considered as ≪cacheable≫
validity	≪caching≫, ≪cacheable≫	required probability for validity of cached results
intervals	≪caching≫	number of intervals to use for the access static for dynamically determining cacheability
intervalLength	≪caching≫	length of the interval for recording access statics

A few special cases have been considered: `setXyz` methods are automatically considered to be ≪not-cacheable≫ and to have a tagged value invalidates set to `getXyz`. `create/remove` methods are also ≪not-cacheable≫ and have invalidates set to all ≪FinderMethods≫.

Profile Extensions for Prefetching

To allow *modeling of prefetching dependencies* in a similar manner like cacheability properties, extensions to our UML Profile are necessary, especially to the *stereotypes* in table 5.2 on the preceding page and the *tagged values* in table 5.3 on the page before. Hence, we introduced yet another stereotype ≪prefetch≫ (table 5.4), which has three tagged values: signatures, depths, and parameters (table 5.5).

Tab. 5.4: Stereotypes for prefetching

Stereotype	*Base Class*	*Description*	*Tagged Values*
≪prefetch≫	Attribute, Operation	tags a component method or component attribute's `get` method as having one or more *prefetching dependencies*, which are further specified by this stereotype's tagged values	signatures, depths, parameters

Tab. 5.5: Tagged values for prefetching

Tagged value	*Stereotype*	*Description*
signatures	≪prefetch≫	a list of method signatures to be prefetched upon invocation of the corresponding method
depths	≪prefetch≫	a list of maximum prefetching recursion depth at which the process of prefetching should stop
parameters	≪prefetch≫	an optional list of parameter mappings for the prefetchable method's input parameters; numbers refer to parameter values of the initiating method (starting with 0 at the **`return`** value); signatures may refer to other attribute accessor methods of the same component instance

The stereotype's three tagged values—signatures, depths, and parameters—can be compared to the tagged value invalidates in table 5.3 on the preceding page; all of them denote lists of values for a number of dependencies. Their handling is rather uncomfortable because multi-valued tagged

values are not properly supported by the current UML specification, which simply provides string (textual) values. Furthermore, signatures, depths, and parameters represent tuples that refer to the same dependencies. The workaround is that all lists are required to have the same number of elements and that all elements at the same position relate to each other as members of a dependency tuple. Since elements of the parameters list may also contain multiple values (for individual parameters of the same dependency), two different delimiters have to be used to distinguish between individual parameters and parameter groups belonging to dependencies.

Problem: multi-valued tags

5.3 Development Process

It has already been outlined that most caching concepts lack integration with visual design (CASE) and builder tools. Although UML provides the means for capturing additional caching-related metadata, a common solution is still to store caching properties externally in proprietary formats using simple text editors and similar tools. The purpose of this section is to demonstrate how the UML Profile for Caching from section 5.2.2 on page 136 can be used within the context of the tool chain presented in section 5.1 on page 132.

We will proceed as follows: Section 5.3.1 will demonstrate how to apply our UML Profile during *component design*. The result of *model transformation* is an abstract component that can be used as a skeleton for further refined component implementations as shown in section 5.3.2 on page 141. This abstract component contains metadata, which is to be processed by *code generators* to create the final implementation classes and deployment descriptors as explained in section 5.3.3 on page 143. We will finally contemplate implications responsibilities and additionally required roles in section 5.3.4 on page 145.

5.3.1 Component Design

Considering the standard software development process, developers already get a fair notion about a component's usage scenarios, corresponding data flows, and application-specific consistency requirements at design time, immediately after thorough analysis. Attributes of components are the most suitable candidates for caching as they contain their actual data. We thus allow to specify caching properties for component attributes using the stereotypes introduced in table 5.2 on page 137:

constant Attributes marked as ≪const≫ practically never change. They are to be cached upon first access;

cacheable Attributes marked as ≪cacheable≫ change rarely[4], they are

[4] The definition of "rarely" is application-specific. Hints about the tolerance of an application

also to be cached upon first access. An appropriate invalidation / update propagation protocol is necessary to maintain the required level of consistency;

not cacheable Attributes marked as ≪not-cacheable≫ should not be cached for various reasons. They might either be subject to frequent changes, or they should only be accessed in a transactional context, or they might contain sensitive data that should not remain in cache for security reasons, just to mention a few examples.

A minimal example for the use of caching stereotypes from section 5.2 on page 133 is depicted in figure 5.3. Note that *tagged values* have no graphical representation in UML by default, which is why they are not included in the figure[5].

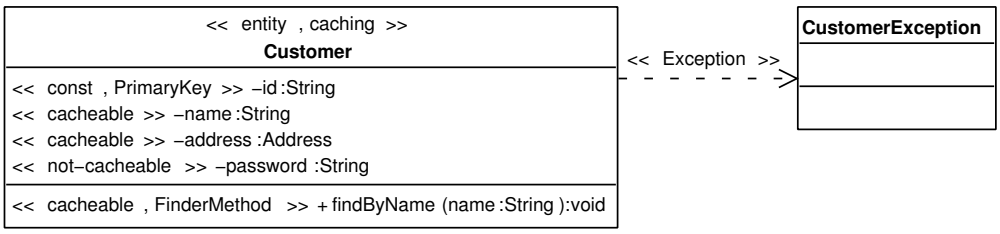

Fig. 5.3: Example for use of caching stereotypes

Figure 5.3 also demonstrates the combination of the *AndroMDA*'s EDOC-like profile for components (cf. section 5.2.1 on page 135) with our UML Profile for Caching (cf. section 5.2.2 on page 136): Customer is tagged as an ≪entity≫ but it is also stereotyped as a ≪caching≫ component, which means that it is generally appropriate for caching and that default values for caching properties may be configured. Another combination of stereotypes can also be seen at the definition of the id attribute as ≪const≫ and ≪PrimaryKey≫ or at the method findByName as ≪cacheable≫ and ≪FinderMethod≫. A CustomerException will be thrown by each business method of the Customer component, including its attribute accessor methods.

Although at the time of writing not many tools support such combinations of stereotypes, the UML and XMI specifications permit the assignment of multiple stereotypes to model elements in the same way that a class may have inheritance relationships to multiple parents. For instance, the CASE tool *Poseidon* provides an appropriate user interface for modeling such relationships.

---

to inconsistencies can be configured using the ttl and validity tagged values to control expiration time and required validity probability. This additional metadata is used by the dynamic/adaptive solution in section 4.2 on page 111

[5] The UML 2.0 Superstructure specification [OMG03c] suggests using associated *comments* to visualize tagged values. However, hardly any tool supports this optional feature at the time of writing.

5.3.2 Component Implementation

In this section we will discover how an abstract component is generated by the model transformer *AndroMDA* as an intermediary representation.

There are currently several ways to trigger model transformation with AndroMDA: Either as an *Ant* task, as a *Maven* plug-in, or as a *Poseidon* plug-in. Irrespective of the actual starting procedure, *AndroMDA* searches the *cartridge* selected for model transformation, e.g., EJB, for available *templates* that match encountered stereotypes. *Abstract components* are successively constructed by applying and processing these templates as already hinted at in section 5.1 on page 132.

Modified templates

We controlled the output by providing modified component *templates* for our model extensions. An AndroMDA template is essentially (Java) pseudo code with mixed in special comments for controlling the transformation process in a proprietary script language.

Listing 5.1 on the next page gives an impression of the way an exported abstract `Customer` component might look like.

Obviously, stereotypes and tagged values have been mapped to special *JavaDoc* comments. As mentioned before in section 2.5.3 on page 67, this scheme of using special tagged comments to convey additional metadata in source code was introduced with Java 1.2. These comments can be evaluated by compiler-independent parsers and tools, as we will see in section 5.3.3 on page 143. Thus, a stereotype ≪cacheable≫ for a component attribute becomes a `/** @caching.method type="cacheable" */` comment above the corresponding accessor method, or a component-level constraint caching.ttl=600 translates to `/** @caching.ttl = 600 */` above the Bean class. The dotted notation of these tags allows for simple namespacing.

Following the JavaBeans pattern of naming accessor/mutator method pairs of component attributes `get/setXyz` by convention, stereotypes of attributes are automatically mapped to the corresponding accessor methods. Theoretically, the invalidates tagged value of the corresponding mutator method would have to be set to the signature of the accessor method. However, since this procedure applies to every attribute, which would only bloat the generated code, we implicitly assume the presence of this tagged value for every attribute access as an optimization. But mutator methods may also have other invalidates relationships, as shown with the example of `setName`.

Since *finder* methods do not have to be implemented directly by the main (EJB) component implementation class, the ≪FinderMethod≫ from figure 5.3 on the facing page is mapped to a special class-level `@ejb.finder` tag and its categorization as ≪cacheable≫ to another tag `@caching.finder`.

The tag `@jboss.container-configuration` is responsible for selecting the appropriate configuration of the interceptor chain, which is required at

Listing 5.1: Example for use of caching tags

```
/** @caching.bean
 *     ttl=600
 *     validity=0.95
 *     intervals=5
 *     intervalLength=1200
 *  @ejb.finder
 *     signature="java.util.Collection findByName(java.
   lang.String name)"
 *     query="SELECT OBJECT(o) FROM customers AS o WHERE
   o.name=?1"
 *  @caching.finder
 *     signature="java.util.Collection findByName(java.
   lang.String name)"
 *     type="cacheable"
 *     validity=0.9
 *  @jboss.container-configuration
 *     name="Caching CMP EntityBean"
 */
public abstract CustomerBean
implements javax.ejb.EntityBean {
  /** @caching.method type="const"
   *  @ejb.pk-field */
  public String getId() throws CustomerException;
  /** @caching.method type="cacheable" ttl=3600 */
  public String getName() throws CustomerException;
  /** @caching.method type="not-cacheable" invalidates="
    java.util.Collection findByName(java.lang.String)" */
  public void setName(String name) throws
    CustomerException;
  /** @caching.method type="cacheable" */
  public Address getAdress() throws CustomerException;
  /** @caching.method type="not-cacheable" */
  public String getPassword() throws CustomerException;
  // (...)
}
```

deployment time and runtime as explained in section 6.2.1 on page 156.

Extensions for Prefetching

Referring to some fictive `Order` component, listing 5.2 illustrates the use of the corresponding *XDoclet* tags for our prefetching stereotypes and tagged values from table 5.4 on page 138 and table 5.5 on page 138.

Listing 5.2: Example for use of prefetching tags

```
/** @caching.bean
 *  @ejb.finder
 *     signature="java.util.Collection
   findByCustomerAndYear(java.lang.String customer, int
   year)"
 *     query="SELECT OBJECT(o) FROM customers AS o WHERE
   o.customer=?1" AND o.year=?2
 */
public abstract OrderBean
implements javax.ejb.EntityBean {
  /** @caching.prefetch
   *     signature="java.util.Collection OrderHome.
     findByCustomerAndYear(java.lang.String,int) throws
     javax.ejb.FinderException"
   *     depth="1"
   *     parameter="0"
   *     parameter="public abstract int Order.getYear()
     throws OrderException"
   */
  public abstract String getCustomer() throws
    OrderException;
  // (...)
}
```

5.3.3 Code Generation

Originally intended to bridge the disconnection between bean implementations and interfaces that often tend to get out of sync, *XDoclet* (see section 2.5.3 on page 67) generates interfaces, deployment descriptors, and auxiliary classes from Bean classes. It allows the construction of arbitrary code segments depending on special *JavaDoc* comments at class / method level and special template files that actually control the code generation process.

XDoclet is usually also triggered as an *Ant* task to process abstract components like the one presented in listing 5.1 on the preceding page. The EJB module of XDoclet uses the included tags to generate by default the concrete Bean class, remote, home, and local interfaces (complete with business and finder methods), primary key and utility classes as necessary, standard and vendor-specific deployment descriptors, i.e., in our case basically `ejb-jar.xml`, `jboss.xml`, and `jbosscmp-jdbc.xml`.

New template

Now the challenge was to get XDoclet to process the additional tags presented in section 5.3.2 on page 141. We have thus created a special XDoclet template `caching.xdt` that is used to generate a separate `caching.xml` file containing component-specific cacheability configuration data for deployment

as depicted in listing 5.3. In contrast to AndroMDA, XDoclet templates are XML files themselves with special XML tags as commands for controlling the transformation process. This template language is not very powerful but nevertheless sufficient for most transformation purposes.

The structure of the `caching.xml` file in listing 5.3 is rather self-explanatory: It sequentially captures cacheability properties of method signatures. The additionally inserted `<invalidates>` tags from mutator methods to accessor methods are also shown. Furthermore, the generated `jboss.xml` is adapted to include our `CachingClientInterceptor` in the client-side interceptor chain.

Listing 5.3: Exemplary caching.xml file

```
1  <caching>
2    <ttl>600</ttl>
3    <validity>0.95</validity>
4    <intervals>5</intervals>
5    <intervallength>1200</intervallength>
6    <method signature="public␣abstract␣java.lang.String␣
       Customer.getId()␣throws␣CustomerException"
7           cacheability="const" />
8    <method signature="public␣abstract␣java.lang.String␣
       Customer.getName()␣throws␣CustomerException"
9           cacheability="cacheable"
10          ttl="3600" />
11   <method signature="public␣abstract␣void␣Customer.
       setName(java.lang.String)␣throws␣CustomerException"
12          cacheability="not-cacheable">
13     <invalidates signature="public␣abstract␣Customer␣
         CustomerHome.findByName(java.lang.String)␣throws␣
         javax.ejb.FinderException" />
14     <invalidates signature="public␣abstract␣java.lang.
         String␣Customer.getName()␣throws␣CustomerException"
          />
15   </method>
16   <!-- (...) -->
17 </caching>
```

After code generation, the application assembler / deployer is given the opportunity to make certain manual adjustments to given descriptors, e.g., changing cacheability of certain attributes, altering expiration times etc.

Extensions for Prefetching

Listing 5.4 displays the `caching.xml` extensions corresponding to listing 5.2.

Listing 5.4: Exemplary caching.xml file with prefetching dependencies

```
1  <caching>
2    <!-- (...) -->
3    <method signature="public␣abstract␣java.lang.String␣
       Order.getCustomer()␣throws␣OrderException">
4      <prefetch signature="public␣abstract␣java.util.
         Collection␣OrderHome.findByCustomerAndYear(java.
         lang.String,int)␣throws␣javax.ejb.FinderException"
         depth="1">
5        <directparameter num="0"/>
6        <parameter signature="public␣abstract␣int␣Order.
           getYear()␣throws␣OrderException"/>
7      </prefetch>
8      <!-- (...) -->
9    </method>
10   <!-- (...) -->
11 </caching>
```

5.3.4 Roles and Responsibilities

Up to now, we have assumed that capturing cacheability metadata is performed on-the-fly by application / component designers. However, appropriately configuring the caching service can be a daunting task, especially with respect to consistency requirements. The adaptive solution presented in section 4.2 on page 111 can help to "heal" cacheability misconfigurations in terms of access behavior and change rates at runtime, but it can not cure the consequences of falsely specified tolerances to inconsistencies.

If none of the developer roles in the component life-cycle (cf. section 2.2.4 on page 24), namely *component provider*, *application assembler*, *deployer*, or *system administrator*, have sufficient information and/or expertise to perform this task, the introduction of an additional role may become necessary. We call this role *Cache Advisor*, inspired by a role with the same name in the context of *Cache Groups* (see section 3.1.3 on page 87) [HB04] whose responsibility it was to specify cache keys and referential cache constraints.

Cache Advisor

Our *Cache Advisor*'s responsibility is to annotate the component-based application model with additional caching-related metadata at various development stages. He may either assist the component provider at design time by annotating the UML model with cacheability categorizations and consistency constraints of component attributes and method results at this early stage. Or he may add and alter caching-related tags in source code after model transformation prior to code generation. But he may also assist application assemblers, deployers, and administrators by adapting `caching.xml` deployment descriptors before (re)deployment. Additional, yet to be built tools might help him with this task.

5.4 Conclusion

This chapter concentrated on how to capture cacheability metadata as early as possible during component development. On one hand, hints about application-specific data access characteristics can help to reach an optimal configuration of the caching service in a timely manner. On the other hand, knowledge about the tolerance of an application to (controlled) inconsistencies of cached attribute values and method results are crucial for preserving the application's semantics.

In contrast to simply specifying caching attributes, e.g., at EJB level, the UML integration allows visual modelling and a higher level of abstraction that potentially enables the transformation of models to other target platforms. Although this has not been implemented so far, the design of our middleware service (cf. section 4 on page 107) could also be applied to other component-oriented middleware platforms (see section 2.2 on page 14 and figure 2.13 on page 45).

A model-driven tool chain was presented in section 5.1 on page 132, which provided the seamless integration of this aspect into the development process. A UML Profile for Caching was constructed in section 5.2 on page 133 to summarize light-weight UML extension elements necessary for adding caching-related metadata to UML application models. The use of these metadata throughout the development process implied by the tool chain was then explained with the help of various examples in section 5.3 on page 139.

However, the presented solution alone is not robust to changes of access characteristics at runtime. It can neither cope with major misconfigurations of cacheability categorizations. These shortcomings motivate the need for an adaptive, self-healing solution like the one that was presented in section 4.2 on page 111. Future work may include *round-trip engineering*, i.e., reversely integrating adaptively determined configurations into the application model.

Many things difficult to design prove easy to performance.
Samuel Johnson (∗1709–†1784), English poet, lexicographer, and critic.

6 Implementation of the Adaptive Middleware Service

In this chapter, we will present the major steps towards the integration of our orthogonal middleware service for caching designed in section 4 on page 107 into the architecture of Enterprise JavaBeans (see section 2.2.4 on page 22) as an exemplary component-based middleware platform. The first approach in section 6.1 is based on the concept of *Stub Annotation* presented in section 3.1.5 on page 95. It can be seen as an early feasibility study that provided directions for further progress. The second approach in section 6.2 on page 155 provides more configurability based on the concept of *Interceptors* presented in section 2.5.5 on page 73 and section 3.1.5 on page 94, which forms the basis for the adaptive extensions that have been designed in section 4.2 on page 111. Its title "Descriptive Point-cutting" emphasizes the relation to Aspect-Orientation (see section 2.5.6 on page 75) and descriptive middleware (see section 2.2 on page 14).

6.1 Stub Modification

The original starting point for this thesis in general was an idea for automatic generation of caching logic in client-side stubs of Enterprise JavaBeans in combination with a mechanism for notification of clients upon updates, which was described in [NPF99]. The JavaBeans pattern for encapsulating component attributes by accessor/mutator methods—a basis for definition 2.8 on page 17—was already a fundamental assumption in the referenced paper. Another supposition was the generation of custom stubs by the EJB container at deployment time as the initial access point for modification: Generated stubs were

augmented by additional functionality for supporting three categories of consistency management mechanisms for component attributes, apart from *non-cacheable* attributes:

pull cacheable. Clients asynchronously update their caches by periodically querying the server.

pullWait cacheable. The server blocks and defers responses to client requests until modifications actually occur. A sort of "semi-synchronous" update is thus accomplished.

push cacheable. Following the *Observer* Pattern (cf. section 2.5.4 on page 70) from [GHJV94], clients register *EventListeners* as callback objects at the server. These are notified upon modifications of attributes to directly propagate updates.

The second mechanism is only feasible if using alternative Java RMI implementations as the underlying transport middleware. The reference implementation would leave ports open for every request, which turned out to be a bottleneck. To improve scalability, the third mechanism was additionally designed to provide hierarchical bundling of callback channels along the component containment hierarchy[1].

Bad scalability

However, the last two mechanisms proved to scale badly with increasing numbers of clients, components, and attributes in later feasibility studies. The reason for this was the considerable amount of necessary status information and potentially open connections between server and clients, which is why only the strategy of polling clients (*pull cacheable*) was implemented in later prototypes.

The first prototype, which is discussed in [PS02], was based on the assumption that most EJB container implementations use Java RMI Stubs as client-side proxies for Bean components. The concept of *Stub Annotation* (see section 3.1.5 on page 95) was used to integrate caching logic as a *client-side container* into these proxies, which can be compared to *aspect weaving* at *join points* in *Aspect-Oriented Programming* (cf. section 2.5.6 on page 75). The remainder of section 6.1 outlines this work before we introduce the interceptor-based solution in section 6.2 on page 155, which represents the basis of all further extensions.

6.1.1 Multiple References

Apart from component state caching in terms of results of attribute accessor methods, a middleware service for caching also has to take care of component identities and handling of multiple references to the same components:

[1] The approach was demonstrated with the example of the eLearning platform *JaTeK* [Neu03], which features a hierarchical structure of course material composed of courses, chapters, sub-chapters, and materials.

Application components are usually related to each other. These relations are typically exposed quite similar to attributes on their interfaces. This implies that there is often more than one way to acquire references to component instances. This leads to an often neglected problem we called *proliferation of stub objects*, which is shown in figure 6.1.

Proliferation of stubs

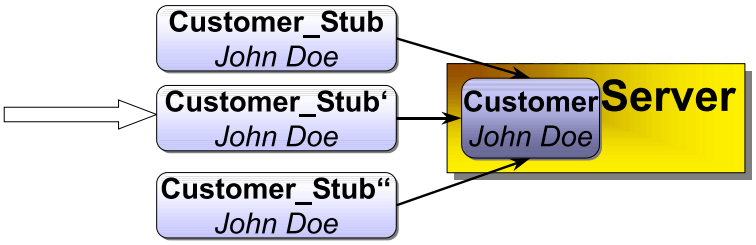

Fig. 6.1: Proliferation of stubs

Whenever remote operations return stubs (or proxies) as remote references, new proxies objects are being created in the client process upon every call. Memory consumption increases linearly with the number of object references. Every client has to check its received proxies for equality and discard identical copies if they reference the same server object, i.e., the same Bean component in our context. This gains even more significance if modified stubs may also contain cached data. If no precautions are taken, clients proverbially drown in cached attributes.

Memory consumption

This problem has been realized by others before but most proposed solutions have not proved satisfactory, due to either their lack of transparency or their complexity for integration. For instance, the Distributed Shared Object solution *Javanaise* (see section 3.1.5 on page 97) [HB01] uses a rather complicated scheme of proxy-in and proxy-out objects on server and client side to manage global object identities of transferred reference parameters. Our concept performs the same tasks with augmented stub objects as we will discuss below.

6.1.2 Client-side Containers

As a solution to avoid proliferation of stubs and to handle multiple references we suggested using a *Client-side Container*, i.e., a look-up table Singleton[2] in each client process, which is queried every time a new stub is received to ensure the identity of remote references.

The concept for integration into the middleware architecture is based on *Stub Annotation*—particularly on the "Smart Stubs" approach by Loton [Lot00]—as described in section 3.1.5 on page 95. This decision was a matter

[2] A pattern in [GHJV94] for client-static objects. More sophisticated implementation could use Java caching services like [Bor01]. Cf. section 3.1.5 on page 95.

of simplicity because it enables easier short-term integration than, e.g., a full-scale proxy generation solution. In other words, generated default RMI stubs are extended by self-defined subclasses as drawn schematically in figure 6.2.

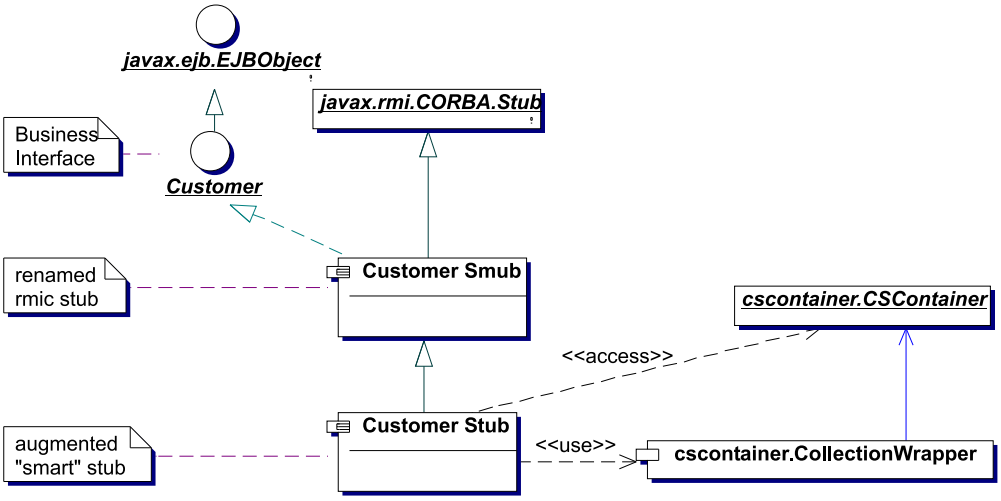

Fig. 6.2: Integration of modified "Smart stubs" with the *Client-side Container*

Equality of component stubs

These modified, tool-generated stubs provide adapted caching functionality for the components' attributes and check remote reference return values against the Client-side Container (see figure 6.3). If a remote reference turns out to equal another stub for the same component, which has already been stored in the Client-side Container's repository, it will be discarded and the equivalent reference from the repository will be returned. On the other hand, the Client-side Container transparently stores the new remote reference, if no equal stub can be found.

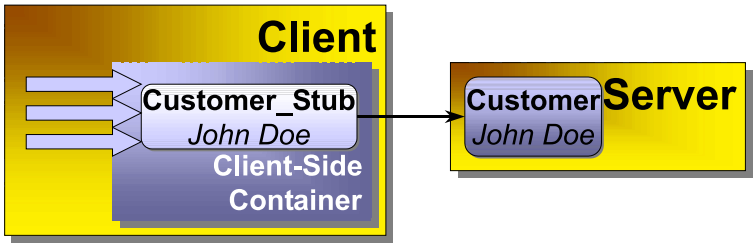

Fig. 6.3: Client-side Container

The actual Client-side Container interface is rather simple as depicted in listing 6.1 on the facing page.

Using the static factory method `getCSContainer()` a concrete instance of a Client-side Container can be obtained by the augmented stubs. This enables transparent exchange of various implementations. The `getStub()` method works in the above described manner.

Listing 6.1: Client-side Container Interface

```
1 public abstract class CSContainer {
2   public static CSContainer getCSContainer() {
3     // (...)
4   }
5   public abstract Object getStub(Object id);
6 }
```

6.1.3 Object Equality in Component-based Middleware Platforms

As mentioned introductorily, special attention must be paid to object identity. One might assume that stubs can simply be maintained in a `Hashtable` but there are certain obstacles that component models like EJB additionally introduce: The specifications [MH99, DYK01, Sect. 8.5] explicitly warn that "the Enterprise JavaBeans architecture does not specify object equality." This implies that the results of `hashCode()` and `equals()` are undefined. Unfortunately, these methods are crucial for proper storage in standard Java hash tables.

`Javax.ejb.EJBObject`s are derived from `java.rmi.Remote` which makes them accessible across network boundaries. But the very concept of EJB relies on dynamic redistribution of subsequent remote calls for a certain business object to different servants for pooling purposes. Hence, `hashCode()` and `equals()` behave as expected for one and the same `javax.ejb.EJBObject` servant but this could potentially be used for different entity identities by the container's pooling algorithms.

Handle instead of equals()

EntityBeans' primary keys are impractical for instance identification because they are only unique for a certain bean type in the context of a single deployment. A possible solution is to delegate the stubs' `hashCode()` and `equals()` implementations to `javax.ejb.Handle`, a long-lived identity object, which can be obtained from every `javax.ejb.EJBObject` as unique persistent reference[3]. Handles are cached as well in a lazy-evaluating way to avoid additional remote calls every time equality is tested.

Utilizing these prerequisites, a first simple Client-side Container prototype based on a single static hash table was implemented.

6.1.4 Integration into the Middleware Platform

To visualize the integration of Client-side Containers, imagine the following example: An `OrderBean` references a `CustomerBean`. A `getCustomer()` method is implemented to enable navigating access in an object-oriented man-

[3] Handles usually contain a primary key and information about the bean's JNDI-URL, among other container-specific information.

ner, returning a remote reference to a `CustomerBean`. A generator tool would realize this dependency, thus inserting code into the generated stub to perform the above mentioned look-up as shown in listing 6.2

Listing 6.2: Modified Stub

```
1  public class _Order_Stub extends _Order_Smub {
2    private static transient CSContainer cscontainer =
3      CSContainer.getCSContainer();
4    private Customer customerCache = null;
5    // (...)
6    public Customer getCustomer() throws java.rmi.
       RemoteException {
7      if (customerCache==null) {
8        Customer c = super.getCustomer();
9        customerCache = cscontainer.getStub(c);
10     }
11     return customerCache;
12   }
13 }
```

Stub renaming

The default stub `_Order_Stub`, which is generated by the standard RMI compiler, is renamed to `_Order_Smub` and the augmented implementation that takes its place is derived from this class. Stub instantiation in Java RMI is based on naming conventions, which makes renaming necessary[4]. On the other hand, default stubs should not be altered beyond simple renaming because their structure may change without notice due to Sun's internal modifications.

This example also demonstrates the integration of caching functionality: An Order's Customer is normally determined at creation time which makes this attribute a perfect candidate for caching. However, multivalued relationships between entities require special attention. We will elaborate on this problem in section 6.1.5 on the facing page

Note that no modifications are necessary on existing client or server code because all changes are entirely transparent to the component itself. Not even sources are needed for supplementary integration of caching. Stubs can be regenerated by `rmic` or a container's corresponding tool based on existing class files. These are in turn altered by the stub generator tool.

Deployment

Most EJB containers provide two archives (JAR files) as output of their deployment tools—one for Bean clients containing remote stubs and interfaces, and

[4] Cf. [Lot00].

another one for the server which additionally comprises the bean implementation itself, generated container glue code that actually implements the bean interface, and all necessary skeleton or tie classes.

The question arises how to get modified proxies into EJB archives. Unfortunately, most EJB servers do not even expose generated server JARs during Bean deployment. As long as there are no container-supported interfaces for deployment, client and server JARs have to be unpacked, modified and copied back to their container-specific locations.

Containers like JBoss [FR03] use dynamic proxies of the Java *Reflection* API instead of stubs (cf. figure 2.21 on page 74). The `InvocationHandler`s behind these proxies call a chain of *Interceptor*s, the last of which finally transmits the invocation using a generic RMI stub. This sophisticated architecture enables other possibilities for integration, which will be discussed in section 6.2 on page 155.

6.1.5 Returning Collections of Stubs

Multiple instance return values, i.e., multivalued relationships, pose an important issue, for instance:

- The `OrderBean` could also provide a method `getItems()` for querying the positions of a given order.
- EJB Home interfaces can provide finder methods that may return multiple instances.

Caching itself is not affected since the annotated stubs are automatically returned by the RMI subsystem. But mechanisms for preventing stub proliferation have to be adapted to ensure their effectiveness. The stubs of those remote objects returning multiple references for other remote objects have to query the Client-side Container for identical stubs before returning any references to their clients.

Different types

Unfortunately, numerous options exist for the purpose of returning multivalued results, e.g., `java.util.Enumeration`, `java.util.Vector`, `java.util.Collection` and it's subclasses, etc. On one hand it is hard to support at least the majority of these options and on the other hand the contents of every collection class are just polymorphic `java.lang.Object`s, which makes it difficult to detect one-to-many associations' corresponding accessor methods via introspection. Other object-oriented languages like C++ provide adaptable *Template* classes for this purpose but Java lacks such a concept until J2SE 5. Checking every return value with **`instanceof`** at runtime would be far too inefficient. A possible way to avoid this is special mark-up by design tools to make the type of an association prominent. This mark-up can in turn be transformed to special tags in source code comments, in analogy to the concepts presented in section 5 on page 131.

In [PS02] we concentrated on the features demanded by the EJB specification: Only `java.util.Collection`s are permissible return values—at least for multivalued `finder` methods. Java `Collection`s are usually accessed by `java.util.Iterator`s. These Iterators can be queried on an on-demand base. This allows fetching of contents as needed. However, this otherwise advantageous feature complicates a solution for the problem of returning collections of stubs: A simple wrapper for the returned Collection that initially converts its peer's content of Smart Stubs by checking them against the Client-side Container may have an impact on over-all performance because collections may potentially contain a vast number of members. Transferring all of them at once may result in undesirable access peaks at server side.

Lazy evaluation: wrap iterators

A far better solution is to encapsulate `Iterator`s as well. This ensures on-demand querying return stubs against the Client-side Container. The code segments in Listing 6.3 and 6.4 on the facing page show how to implement this:

Listing 6.3: CollectionWrapper

```
1  public class CollectionWrapper implements Collection {
2    private Collection peer;
3    private static transient CSContainer csc =
4      CSContainer.getCSContainer();
5    // (...)
6    public Iterator iterator(){
7      return new IteratorWrapper(peer.iterator());
8    }
9    public Object[] toArray(){
10     Object[] arr = peer.toArray();
11     if (arr!=null) {
12       for (int i=0;i<arr.length;i++) {
13         arr[i] = csc.getStub(arr[i]);
14       }
15     }
16     return arr;
17   }
18 }
```

6.1.6 Consistency

It has been outlined that changes made to replicated distributed objects always result in inconsistencies. Many applications can fortunately cope with these inconsistencies for a certain amount of time. For instance, expiration times in the form of simple TTL counters are common in Web caching (see section 3.1.1 on page 84). The approach presented in [PS02] supports only *dirty reads* (see section 2.3.1 on page 46). Cache invalidation via regular cache purges (i.e., TTL

Listing 6.4: IteratorWrapper

```
1  public class IteratorWrapper implements Iterator {
2    private Iterator peer;
3    private static transient CSContainer csc =
4      CSContainer.getCSContainer();
5    // (...)
6    public Object next(){
7      Object next = peer.next();
8      return csc.getStub(next);
9    }
10 }
```

expiration) can be enabled optionally. Client callback objects as described for the *push cacheable* mechanism above have been considered an option, which was abandoned later because of their bad scalability and their limitations in firewalled scenarios (cf. section 1.4 on page 5), although they would have been practicable for few Web servers as the only clients of the business tier. Cache consistency protocols from the database domain (see section 3.1.3 on page 87) were also taken into consideration but this direction was not pursued further for the initial prototype.

6.1.7 Conclusion

Although the initial prototype presented in [PS02] was based on a different integration technology, it nevertheless helped to isolate a number of problems in connection with caching method results of application components above the business tier, e.g., handling multiple references to single component instances, wrapping multivalued results, or managing consistency. These initial results formed the basis for extensions towards a different integration technology described below. First experiences were also gathered with respect to modeling issues and software development cycle integration, which have been presented in section 5 on page 131.

6.2 Descriptive Point-cutting

The concept of *Interceptors* has already been introduced as a flexible mechanism for integrating orthogonal middleware services in section 2.5.5 on page 73 and section 3.1.5 on page 94. On the other hand, incorporating our caching service into an existing middleware infrastructure using *Stub Annotation* proved to be complicated and inflexible, especially with respect to deployment, as described in section 6.1.4 on page 151. The combination of both—client-side caching of method results and integration via interceptors—was first presented in [PG03].

As mentioned introductorily, our second prototype [PG03] follows a static approach, i.e., cacheability of attributes and method results has to be declared at deployment time. Once considered cacheable, an attribute or method result remains in that state. A reference implementation based on Sun's Enterprise JavaBeans (EJB) platform and the open source EJB container JBoss [FR03] was developed to demonstrate the underlying concepts. JBoss was only chosen as an example platform for demonstration for pragmatic reasons: It is available open-source, comparatively well documented, and it is extensible and modularly built, not at least because of its interceptor framework.

Later implementations [Poh03, PS03] extended this prototype with respect to adaptive cacheability categorization as discussed in section 4.2 on page 111.

6.2.1 Integration into the Middleware Platform

JBoss Interceptors

Section 2.5.5 on page 73 already discussed the differences between CORBA Portable Interceptors [OMG01] and the variant implemented by JBoss [FR03]: The latter rather resemble a *Chain of Responsibility* and allow more flexibility for implementing services on top of this meta-programming mechanism.

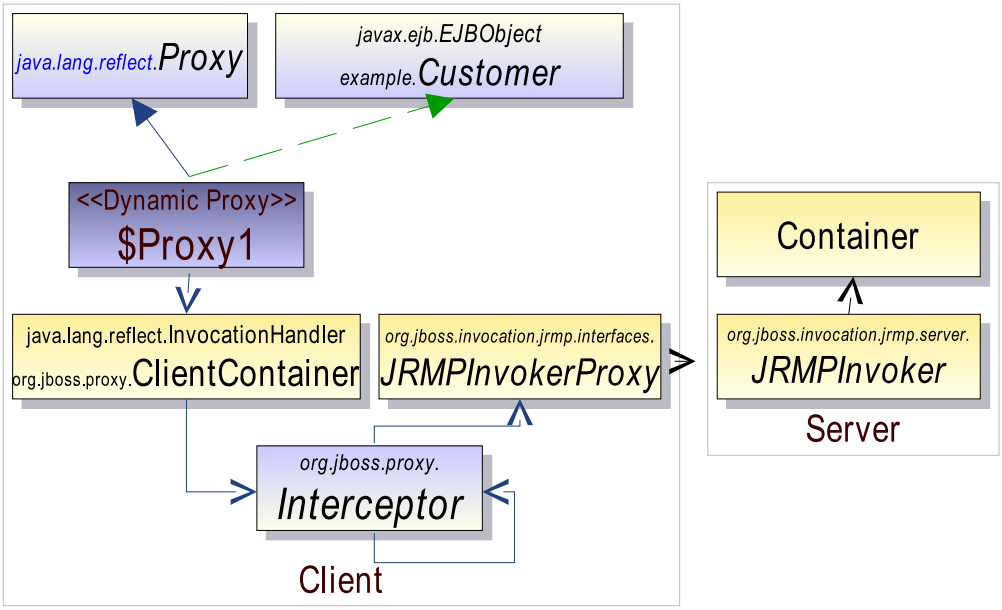

Fig. 6.4: Interceptors in JBoss Dynamic Proxies

Let us now take a closer look at the client-side integration of these interceptors as shown in figure 6.4 to better understand the integration of our caching service: Client-side component proxies are transparently instantiated at runtime using Java's Dynamic Reflection API, which allows them to dynamically implement arbitrary interfaces. The `Proxy` raises invocations to the metalevel by transforming them into `Invocation` objects, which are passed

to an `InvocationHandler`. A `ClientContainer`[5] passes each request as an `Invocation` metaobject through a chain of `Interceptor`s. The last interceptor always hands the request to a static `InvokerProxy` that finally transmits it to the server. A number of implementations exist for different transport protocols. The `JRMPInvokerProxy` uses the Java Remote Method Protocol, i.e., RMI, for this task. It actually holds a *Stub* object of the `JRMPInvoker`, which is an RMI servant object at server side that dispatches `Invocation` metaobjects to `Container`s.

Server-side interceptors are stacked in a similar fashion within `Container`s. However, they are omitted in figure 6.4 on the preceding page for brevity since the static solution only relies on client-side interceptors. When a response returns from the server, it passes through the same interceptor chain in reverse order.

Deployment

The default sequential order of both client-side and server-side interceptors is determined by the server administrator. It can be overridden by bean providers and/or application assembler for specific beans using the `jboss.xml` deployment descriptor. By overriding the default interceptor chain, we managed to integrate our `CachingClientInterceptor`, which is discussed below.

When a caching-enabled component is deployed, the interceptor chain is assembled along with a dynamic `Proxy` as shown in figure 6.4 on the facing page according to the configuration in its `jboss.xml` deployment descriptor. An additional `caching.xml` descriptor is evaluated by our modified deployer. This descriptor contains all necessary information about method cacheability, including cacheability, validity time, and invalidation dependencies of methods. The latter two aspects have been hidden in listing 6.5 on page 159 for brevity.

New deployer

The creation of the `caching.xml` descriptor and its contained cacheability information was discussed in section 5.3.3 on page 143 in the context of component design and implementation. We created a modified deployer infrastructure component for JBoss, which scans deployed archives for contained `caching.xml` descriptors. The deployer uses this information to preconfigure the static `CacheabilityDB`, which is a *Singleton* on both server and client side, referenced by `CachingClientInterceptor` objects. The whole dynamic proxy composition is registered with the JNDI (naming) service as the last step of deployment. Upon a client's first JNDI lookup of the Bean component, the proxy/stub for the Bean is marshalled and transferred. Interceptor instances are then created in the client JVM as needed by remote references.

[5] Not to be confused with the concept of *Client-side Containers* in section 6.1.2 on page 149!

Runtime

The implementation of the concept for static caching designed in section 4.1.1 on page 108 is depicted in figure 6.5, which corresponds to listing 6.5 on the next page.

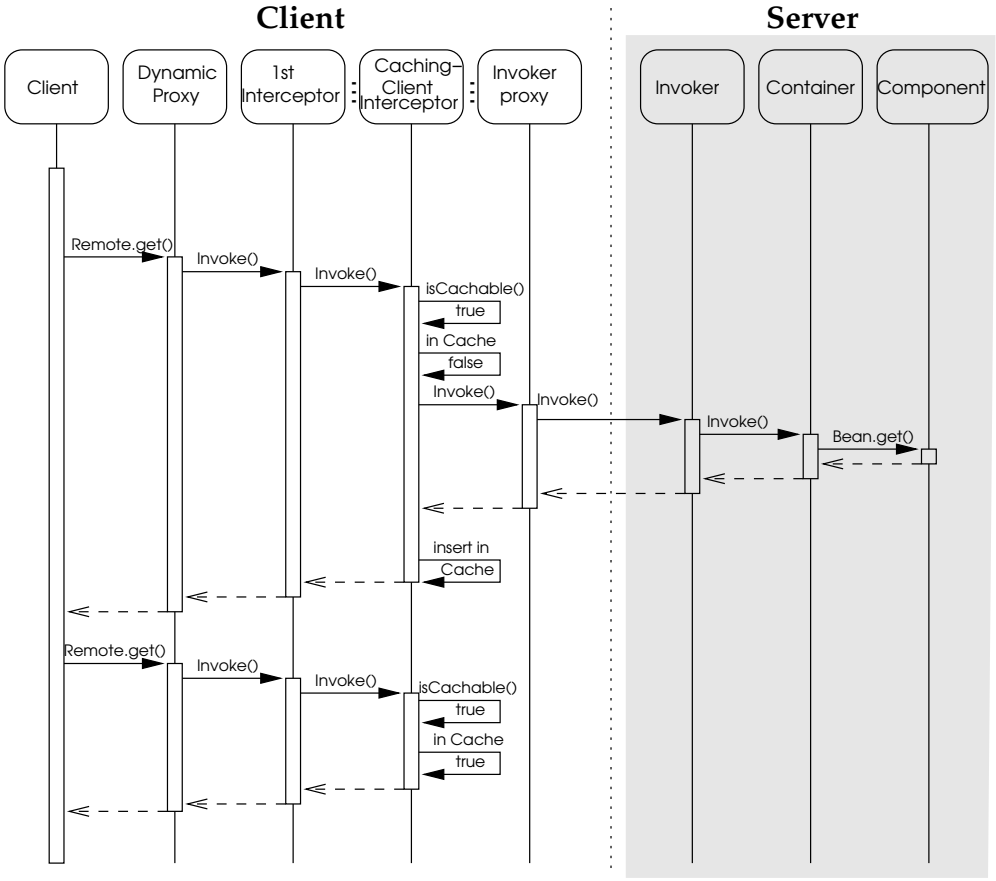

Fig. 6.5: Sequence of static caching

Prototypically, the cache back-end `CachePolicy` was implemented selectively using JBoss' `LRUCachePolicy` or `TimedCachePolicy` with component identity[6] i, method m, and method parameters $\{p\}$ as combined keys and results r as values, i.e., $(i, m, \{p\}) \rightarrow r$. The basic granularity of cached data is per-attribute but as these are members of identifiable components, collective invalidation of attributes is still possible.

Although we initially took measures to let `CachingClientInterceptors` perform *multiple reference handling* as explained in section 6.1.2 on page 149 by checking all returned remote references (i.e., proxies) for duplicates in the

[6] A Bean instance identity in JBoss is composed of `objectName` and `id`; the former uniquely identifies a deployment (or *installed component* as in figure 2.18 on page 63), the latter represents either the *primary key* of an Entity Bean or a session key of a Session Bean.

Listing 6.5: A simple `CachingClientInterceptor`

```
1  public class CachingClientInterceptor extends Interceptor {
2    // (...)
3    public InvocationResponse invoke(Invocation mi) throws Throwable {
4      // get cacheability data for the invocation
5      CacheabilityDBEntry dbe = cacheabilityDB.getEntry(mi);
6      if (dbe!=null && dbe.getCacheability()!=NOT_CACHEABLE) {
7        // create the cache key for the invocation
8        CacheEntry ce = new CacheEntry(mi.getObjectName(), mi.getId(),
           mi.getMethod(), mi.getArguments());
9        // obtain reference to the corresponding cache
10       CachePolicy cp = CachePolicy.getCachePolicy(dbe);
11       // look up result
12       CacheResult result = (CacheResult) cachePolicy.get(ce);
13       if (result==null) { // cache miss
14         // invoke next interceptor in chain
15         InvocationResponse response = getNext().invoke(mi);
16         // new result wrapper object
17         result = new CacheResult(response.getResponse());
18         cp.insert(ce, result); // insert into cache
19         return response;
20       } else { // cache hit
21         // return response from cache
22         return new InvocationResponse(result.getResult());
23       }
24     } else { // not cacheable
25       return getNext().invoke(mi); // normal invocation
26     }
27   }
28   // (...)
29 }
```

local cache, we eventually removed this feature because it is not really needed: The dynamic proxy compositions of figure 6.4 on page 156 do not contain a noteworthy amount of state. The crucial parts, i.e., `CacheabilityDB`, `CachePolicy`, and `InvokerProxy`, are actually static members (*Singletons*) that have to be transferred only once, upon initial access.

Bidirectional Piggyback Communication

Adaptivity and invalidation as designed in section 4.2 on page 111 require the presence of bidirectional piggyback communication for exchanging metadata between client and server.

Version 3.0.6 of JBoss, which was used as the basis for our middleware extensions, unfortunately provided only a unidirectional channel for piggybacking information with invocations from client to server. The opposite direction has not been considered in this version. Hence, we had to provide a back-port from JBoss version 4.0.0 RC2 to enable the return channel for piggyback information. This back-port basically consists of the aforemen-

tioned `org.jboss.invocation.InvocationResponse` as the return type of generic `org.jboss.proxy.Interceptor.invoke(Invocation)` and `org.jboss.ejb.Interceptor.invoke(Invocation)` methods instead of `java.lang.Object`.

6.2.2 Conclusion

Our experiences with the framework showed the general feasibility of the concept. The use of client-side interceptors is mandatory with JBoss, so the overhead for invoking yet another interceptor is quite low. Cache lookups turned out to be faster by magnitudes than direct component attribute queries requiring a full client-server round-trip. A simple test scenario was set up with both client and server JVM running on the same host[7] to eliminate the interfering influence of variable network latency. Results of cache miss times for queries to a component's *value object* were around 20 *ms* per request, reaching peak values of up to 1 *sec*, compared to 1 *ms* and even less for cache hits. Depending on networking infrastructures, several more *ms* can be added for cache misses in non-local scenarios.

Further details about this architecture and how its concepts may also be applied to other non-functional middleware services like monitoring and adaptation have also been published by in [PG03].

[7] AMD Athlon™ XP1600+, 1GB RAM, Linux 2.6.5, Sun J2SE 1.4.2, JBoss 3.0.6

"The more frequently one uses the word 'coincidence' to explain bizarre happenings, the more obvious it becomes that one is not seeking, but evading, the real explanation." Or, shorter: "The belief in coincidents is the prevalent superstition in the Age of Science."

Illuminatus!—The Eye in the Pyramid by Robert Joseph Shea (∗1933–†1994) and Robert Anton Wilson (∗1932), American journalists and authors.

7 Conclusions and Outlook

The most prominent achievement of this work is represented by the continuous integration of the aspect of caching in distributed, component-oriented applications as an orthogonal, descriptively configurable middleware service. Dynamic adaptation to changing access characteristics helps to reach optimal configurations of component attribute cacheability and thus complements this work with respect to data dependency. Consequential capturing and generative reuse of caching-related metadata from design time to runtime underpin the approach from the perspective of software engineering.

7.1 Evaluation

The following claims have been made in section 1.3.3 on page 5 with respect to the contributions of this work:

1. The aspect of replication of component state can be transparently outsourced as a middleware service.
2. Early gathering of replication-related metadata stabilizes the development of distributed applications and reduces the necessary amount of hand-written code.
3. Adaptivity helps to reach appropriate configurations.

The existence of a working prototype can already be seen as a proof for claim *(1)*. A virtual example application served as a touchstone for the effectiveness of our prototype as explained in section 7.1.1 on the next page. Part of claim *(3)* is also covered by this test example. However, the challenges of a

full-fledged quantitative performance evaluation are discussed in section 7.1.2 on the facing page. The argumentation in favor of claim *(2)* is given in section 7.1.3 on page 167.

7.1.1 Functional Evaluation

The functionality of the prototypical middleware service implementation has been evaluated qualitatively using a simple application data model from the e-learning domain. This example has been chosen to reflect the requirements of originally intended scenarios like the *Java-based Teleteaching Kit (JaTeK)* [Neu03].

This test application consists of a simple text-based client application in the form of *JUnit* test cases and four different Entity Beans that are accessed by the client:

- `RootElement` represents a central Entry point into the data server-side structure;
- `CourseElement` represents components containing course data, i.e., coherent collections of learning material;
- `ChapterElement` represents sections of the course structure;
- `LessonElement` represent atomic learning objects, which are grouped in chapters.

This simple structure spans a tree with a depth of four levels. The existence of attributes at each level and the potentially considerable width of the tree structure especially in the first three levels provided a good basis for testing caching as well as prefetching functionality. The test client repeatedly traverses and modifies the tree structure and its attributes in random order, starting with a test dataset of three children per parent at each level.

The *developer of the client application* does not have to take care of integrating the caching service. This integration is performed transparently by the transmission of proxy objects, which have been preconfigured appropriately at server side according to the information given in the components' deployment descriptors.

The component developer (the *bean provider* in terms of EJB) has to consider the aspect of client-side replication only at a much higher level of abstraction by modeling cacheability categorizations descriptively for component attributes and method results during component design. This step can also be performed at a later point in component development by experts we refer to as *cache advisors*. Deployment descriptors are generated accordingly to configure the middleware service at deployment time.

Hence, the general functionality and *effectiveness* can be tested with respect to general caching, adaptation of expiration times, as well as prefetching, but

no real client access patterns are simulated. Thus, no statements can be made about the *efficiency* of the implemented middleware service in terms of timely delivery of responses, accuracy of predicted expiration times, and the resulting potential staleness of cached data. Therefore, a *quantitative evaluation* is necessary.

7.1.2 Quantitative Evaluation

Caching is about performance. Hence, a quantitative analysis is needed to judge the actual benefits of caching-related optimizations. However, improved caching algorithms and strategies have explicitly been marked as out of this work's scope in section 1.3.2 on page 4. We thus mitigated our endeavors somewhat in this respect and settled for a general *cost-benefit estimation* of our proposed extensions, followed by a discussion of the special challenges of a proper evaluation. Actual *performance benefits* can be estimated roughly by drawing parallels to the success of used base algorithms in other domains.

Cost-Benefit Estimation

As a starting point, we compared the different proposed extensions of our middleware service in table 7.1 with respect to

- Gained *flexibility* during software development and design;
- Decreased *latency* of invocation responses (i.e., the *benefit*); and
- General runtime *overhead* for statistics, adaptation, etc. (i.e., the *cost*).

Tab. 7.1: Cost-benefit estimation

Extension	*Flexibility*	*Latency*	*Overhead*
Static Caching	∘	⊖	∘
Dynamic Caching	⊕	⊖	⊕
Static Prefetching	⊕	⊖⊖	⊕⊕
Dynamic Prefetching	⊕⊕	⊖⊖	⊕ ⊕ ⊕

Static caching based on interception as proposed in section 4.1 on page 107 is taken as a basis for comparison since all the other concepts have been implemented as step-by-step extensions of this base mechanism. This solution already provides a much higher degree of flexibility than traditional, hard-coded, *implicit* approaches towards integrating caching services, because introduces the concept of *explicit* (descriptive) middleware to the domain of

caching. However, it offers only little flexibility in terms of altering cacheability metadata at runtime. Although a small penalty is imposed to response latency because of the additional client-side interceptor responsible for processing cached data, the overhead for bookkeeping is non-existent because this basic mechanism does not feature any adaptation for which statistical analysis would be needed.

For the target platform JBoss, the use of client-side interceptors is mandatory, so the latency overhead for invoking yet another interceptor is quite low. Cache lookups turned out to be faster by magnitudes than direct component attribute queries requiring a full client-server round-trip. A simple test scenario was set up with both client and server JVM running on the same host[1] to eliminate the interfering influence of variable network latency. Results of cache miss times for queries to a component's *value object* were abound 20 *ms* per request on average, reaching peak values of up to 1 *sec*, compared to 1 *ms* and even less for cache hits. Depending on networking infrastructures, several more *ms* can be added for cache misses in non-local scenarios. Thus, the overall latency of invocation responses is decreased.

Dynamic caching is the first extension introduced in section 4.2 on page 111 as a concept for dynamic adaptation of cacheability categorizations and cache expiration times at runtime. This solution offers more flexibility since optimal cacheability configurations are not anymore ultimately necessary at deployment time. Misconfigurations are "healed" by the middleware service itself, gradually converging to an appropriate configuration. Experiments with the prototype during functional evaluation showed that the additionally imposed latency can be neglected when comparing full client-server round-trips of the traditional version to the augmented invocation processing. This penalty is expected to be outweighed by the increased cache hit rate due to more appropriately adapted cache expiration times. The static overhead for computing access statistics for every time interval at client side and server side increases however in comparison to static caching.

Static prefetching has been designed as yet another extension of dynamic caching in section 4.3 on page 120. Since prefetching dependencies have to be specified in advance (before deployment), it adds only little flexibility at runtime. The latency of invocation responses is expected to decrease on average due to the higher cache hit rate imposed by prefetched results. The effort for triggering additional, asynchronously processed prefetching invocations is the reason for the higher overhead.

Dynamic prefetching has been conceived in section 4.4 on page 125 to parallel the adaptive capabilities of dynamic caching with respect to prefetching dependencies, as well. Hence, it increases the flexibility of the solution at runtime. The imposed response latency is comparable to static prefetching. The additional latency for transmission of statistical data is expected to be outweighed by the higher cache hit rate due to the more appropriately config-

[1] AMD Athlon™ XP1600+, 1GB RAM, Linux 2.6.5, Sun J2SE 1.4.2, JBoss 3.0.6

ured prefetching dependencies. However, the estimated overhead in terms of memory consumption and processing time for statistical analysis increases much more for the designed solution, which seriously affects the overall performance of the system.

Conclusion We have shown that runtime flexibility increases with our extensions while latency of invocation responses decreases on the *benefit* side. These positive effects are complemented by an increasing overhead on the *cost* side. The actual break-even is highly application-specific and depends on a number of factors, for instance high-level indicators such as cache hit rates, but also low-level factors such as deviation of user access behavior with respect to average time between component attribute modifications / method invalidations or individual prefetching dependencies. Quantitative statements for these performance parameters require a proper performance analysis and benchmarking.

Performance Benchmarking

We have argued above that the evaluation of performance benefits of our solutions requires a *quantitative* analysis. From a networking or systems engineering point of view, this requires the benchmarking of system characteristics like throughput (requests and/or amount of data per time unit) and maximum number of concurrently serveable clients (determined by iteratively searching for a peak until throughput decreases again with additional clients). This sort of analysis is typically conducted for Web-based, multi-tiered client-server applications by means of benchmark suits, such as the *Rice University Bidding System (RUBIS)* [CMZ02, CCE+03] or *TPC-W* [Smi01].

We originally intended benchmarking our solution with RUBiS, to be able to compare our approach to related work like Pfeifer's Method-based Caching [PJ03]. RUBiS was also chosen instead of TPC-W [Smi01], which rather overloads the database tier, according to [CMZ02]. Its *System Under Test (SUT)* models an *eBay*[2]-like Web-based auction application for which a number of alternative implementations are provided, including PHP4 and Java Servlets, the latter with and without various EJB-based business tier variants. All alternative implementations access the same database schema. Hence, the focus of this benchmark is on comparing different implementation variants of presentation tier and business tier. The EJB-based business tier variants include pure Session Bean or Entity Bean solutions, as well as implementations with Session Façades and Value Objects (see section 3.1.4 on page 91) for different versions of the EJB specification.

Our original plan was to augment the variant based purely on Entity Beans with caching metadata and to compare its performance to the "hand-optimized" variant with Session Façades and Value Objects.

[2] `http://www.ebay.com/`

Problems with Traditional Benchmarks. However, a number of considerations led us to abandon these plans. Benchmarks like RUBiS and TPC-W focus on *transactional* Web applications. In contrast, our solution explicitly focuses on applications with weaker consistency requirements (cf. section 1.4 on page 5). For the RUBiS components it was sometimes hard to give meaningful cacheability categorizations that reflect the actual consistency requirements. In general, most attributes would have to be marked as ≪not-cacheable≫, rendering our caching service useless.

Most existing benchmarks capture only a limited scope of possible client access behavior. Hence, the general applicability of expected test results is rather questionable, since the characteristics of one benchmark (i.e., cache hit rates, deviation of client access behavior and prefetching dependencies etc.) may be especially (un)suitable while those of another benchmark are not.

Moreover, changing access behavior is not covered by any of the known benchmarks. These parameters are typically preconfigured statically by giving transition probabilities between navigation steps of the modeled work flows. A could have been conducted by monitoring real user behavior. However, this would have exceeded the personnel and time constraints of this work.

Another problem of existing benchmarks like RUBiS is that the benchmarked applications (SUT) usually feature hard-coded patterns like Value Objects even in the pure Entity Beans variant. These patterns try to solve the same problems as our caching service, although at a different architectural level. Our declared goal is to avoid *any* of these implicit concepts for streamlining data transfer as a rule, because this sort of code is orthogonal to the actual business logic.Hence, a lot of refactoring and reverse-engineering is required to make these existing benchmarks useful for our purposes.

Comparison of Used Base Mechanisms. The issues listed above would require either serious refactoring of existing benchmarks or a completely new benchmark implementation from scratch. Both solutions were not feasible because of time constraints.

Instead, we argue for the plausibility of our expected results by referencing the successful deployment of the same base mechanisms that underly our service in other application scenarios. Web-Caching protocols (cf. section 3.1.1 on page 84) like *adaptive time-to-live (ATTL)* [RS02, Sect. 10.1.2], *Piggyback Client Validation (PCV)* [KW97], and *Piggyback Server Invalidation (PSI)* [KW98] closely resemble our approach for dynamic/adaptive caching (see section 4.2 on page 111) as we have shown before. We assume that our middleware service will thus perform comparatively for similar low-write application scenarios like the ones introduced in section 1.2 on page 2.

7.1.3 Software Development Process

The goal of *qualitative* analysis from the software engineering perspective is to prove claim *(2)*, i.e., early gathering of replication-related metadata stabilizes the development of distributed applications and reduces the necessary amount of hand-written code.

The primary target of our proposed extensions to the software development process is indeed rather *stability* than *acceleration*. We capture caching-related metadata early in the design phase for later generative transformation into descriptors for middleware service configuration. This obviously reduces the amount of hand-written code, which would otherwise be necessary if caching logic was to be integrated implicitly into application code. Less hand-written code makes the whole development process less error-prone, thus increasing the stability of software development.

A similar success story can be observed for other middleware services as well, for instance, security, transactions, and persistence, among others. This has already been discussed as the major advantage of *descriptive* middleware over *implicit* middleware in section 2.2.2 on page 18. Thus, we argue that the proposed consideration of our caching service throughout the software development cycle is a consequential and necessary step towards simple reconfigurability, which follows the general rule of *separation of concerns*.

Acceleration of software development is a side effect of this approach. However, we did not investigate the quantitative extent of this effect. This would have required a field study of at least two representative user groups challenged with a given software engineering task over a longer period of time—one group using the traditional approach of hand-coding caching logic, the other using our new approach.

Another consideration is towards usability of our extension of the development process. We argue that visual modeling and descriptive configuration is generally easier to use than hand-coded access to non-standard frameworks and programming libraries. This argumentation is easy to follow if the general history of Computer-Aided Software Engineering (CASE) tools and their gradual success over traditional, non-visual programming is taken into account. However, a thorough analysis in this respect would also require a longer case study with at least two representative user groups. We have left this as an outlook as well due to time constraints.

7.2 Outlook

Throughout this work, a number of open issues have been isolated and left for future work. In this section we will capture the most important ones of these issues and sketch a short outline of possible next steps.

Performance Evaluation. In section 7.1.2 on page 163 we have discovered that proper benchmarking would be necessary to be able to make more definite statements about the actual performance of our prototype implementation for certain use cases and application scenarios. A number of problems with existing benchmarks have been isolated, which have to be solved first, either by refactoring and extending an existing benchmark or by implementing a new one. These problems include:

- Strong consistency requirements of most benchmarks;
- Limited, statically modeled access behavior is insufficient for evaluating the adaptivity of our dynamic caching solution; and
- Hard-coded patterns for optimized data transfer in most benchmarks.

Improved Software Development Support. Especially with respect to modeling of prefetching dependencies (cf. section 5.2.2 on page 138), we have discovered a need for further optimizations. The tagged values of signatures, depths, and parameters all refer to *lists* of values because multi-valued tagged values are not properly supported by the current UML specification [OMG03c]. As soon as the OMG introduces a more comfortable modeling mechanism for this issue, our profile should be adapted accordingly.

Reverse Engineering. Our modeling facilities are quite useful for applications design from scratch. This was to be expected since our solution was designed especially for this use case. However, hardly any tool support is currently available for extending existing applications to incorporate our caching service. We discovered this, for instance, during our attempt to extend the RUBiS benchmark SUT. In analogy to other model-driven tools that gradually begin to address reverse engineering of existing applications and bidirectional transformation of metadata between different model abstraction levels (cf. section 2.5.2 on page 65), a solution would be desirable that allows to obtain cacheability metadata for existing components by means of visual modeling tools instead of hand-coding the corresponding `caching.xml` descriptor file.

Feature Interaction A major challenge of aspects and non-functional properties—or of separation of concerns in general—is the often unanticipated interaction of these features, which is also sometimes referred to as *aspect interference*. Examples of such interferences include the relation between the response time of an invocation served from cache and its potential staleness, or the confidentiality of passwords or cryptographic keys, which in turn affects the cacheability of this data from a security point of view. It is plain to see, that there are more driving forces behind the cacheability categorization of component attributes and method results than plain application-specific consistency requirements and tolerances.

Feature interaction has been a research area for more than ten years, especially in the telecommunications domains. A further investigation of its implications in our application domain poses be an interesting research topic.

Alternative Consistency Protocols. Up to now, our prototype implements only one consistency protocol: our heuristic for adaptive invalidation of cached method results. This protocol is inappropriate for applications with more stringent consistency requirements. Competing related approaches like Pfeifer's Method-based Caching [PJ03] in turn provide only one protocol for strong consistency. Such protocols impose an unnecessary performance penalty for applications or application parts with weaker consistency requirements. Hence, it would be sensible to have a flexible possibility for integrating of alternative consistency protocols in a descriptive manner. In analogy to our dynamic caching solution, one could even imagine an adaptive selection of consistency protocols depending on the current network topology[3] other factors.

[3] E.g., Piggyback (in)validation for environments with NAT and firewalls; broadcast or callback invalidation for LAN environments.

Bibliography

[AAG+02] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. ACME: Adaptive caching using multiple experts. In *Proceedings of the 4th Workshop on Distributed Data and Structures (WDAS 2002)*. Carleton Scientific, 2002. 85, 86

[ABCdO96] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the 4th Conference on Parallel and Distributed Information Systems (PDIS'96)*, Miami Beach, FL, USA, 1996. IEEE. 11

[ABGMA88] Rafael Alonso, Daniel Barbará, Hector Garcia-Molina, and Soraya Abad. Quasi-copies: Efficient data sharing for information retrieval systems. In Joachim W. Schmidt, Stefano Ceri, and Michele Missikoff, editors, *Proceedings of the 1st International Conference on Extending Database Technology (EDBT'88)—Advances in Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 443–468, Venice, Italy, 14–18 March 1988. Springer. 87

[ACM01] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall / Sun Microsystems Press, 1st edition, 26 June 2001. 70, 92, 94

[AEB03] Omar Aldawud, Tzilla Elrad, and Atef Bader. UML profile for aspect-oriented software development. In Aldawud [Ald03]. In conjunction with the International Conference on Aspect-Oriented Software Development (AOSD 2003). 76

[AGLM95] Atul Adya, Robert E. Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the International Conference on Management of Data*, pages 23–34, San Jose, CA, USA, 22–25 May 1995. ACM SIGMOD, ACM Press. 89, 119

[AH03] Hamud Al Hammoud. Entwicklung eines adaptiven Caching-Dienstes für verteilte Komponentensysteme. Diplomarbeit, Technische Universität Dresden, Dresden, Germany, 1 September 2003. 110, 119, 120, 126

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. 70

[Ald03] Omar Aldawud, editor. *Proceedings of the 3rd International Workshop on Aspect-Oriented Modeling with UML*, Boston, MA, USA, 18 March 2003. ACM SIGSOFT / SIGPLAN. In conjunction with the International Conference on Aspect-Oriented Software Development (AOSD 2003). 76, 171

[Asp01] Aspect-oriented programming with AspectJ. Project homepage of aspectj.org (Xerox PARC) `http://www.aspectj.org`, 2001. 75, 76, 77

[Aßm03] Uwe Aßmann. *Invasive Software Composition*. Springer, Berlin, Heidelberg, 2003. 16

[BAB+00] Arno Bakker, E. Amade, Gerco Ballintijn, Ihor Kuz, P. Verkaik, I. van der Wijk, Marteen van Steen, and Andrew S. Tanenbaum. The globe distribution network. In *Proceedings of the USENIX Annual Technical Conference 2000*, pages 141–152, San Diego, CA, USA, June 2000. 97

[Bak02] David E. Bakken. *Encyclopedia of Distributed Computing*, chapter Middleware. Kluwer Academic Press, 2002. In press. 18, 19

[BB03] Bill Burke and Adrian Brock. Aspect-Oriented Programming and JBoss. *ONJava.com: The O'Reilly Network*, 28 May 2003. 29, 77, 78, 96

[BC87] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. In *OOPSLA Workshop on Specification and Design for Object-Oriented Programming*, 17 September 1987. 70

[BCRP98] Gordon S. Blair, Geoff Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In Davies et al. [DRS98], pages 191–206. 71, 72, 75

[BEA00] BEA Systems. *Using Custom WebLogic JSP Tags (cache, process, repeat)*, WebLogic server 6.0 edition, 2000. `http://e-docs.bea.com/wls/docs60/jsp/customtags.html`. 83

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1st edition, 5 October 1999. 61

[Bel66] Lazlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):79–101, 1966. 12, 100

[Bes95] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of the 4th ACM International Conference on Information and Knowledge Management (CIKM'95)*, pages 403–410, Baltimore, MD, USA, November 1995. ACM Press. 103

[Bes96] Azer Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time in distributed information systems. In Stanley Y. W. Su, editor, *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*, pages 180–187, New Orleans, LA, USA, March 1996. IEEE Computer Society Press. 103

[BH04] Andreas Bümann and Theo Härder. Cache groups—an in-depth analysis of design issues. In *Proceedings of the 30th VLDB Conference*, Toronto, Canada, 31 August–3 September 2004. Very Large Data Base Endowment. 87, 89, 101

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Boston, MA, USA, 1987. 11, 46, 47, 48, 49, 50, 51, 52, 90, 91, 99

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. *Web Service Architecture*. W3C, 11 February 2004. W3C Working Group Note. 41, 42

[Bir04] Kenneth P. Birman. Like it or not, Web Services are Distributed Objects. *Communications of the ACM*, 47(12):60–62, December 2004. 44

[BKTJ92] Henri E. Bal, M. Frans Kaashoek, Andrew S. Tanenbaum, and Jack Jansen. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency: Practice and Experience*, 4(5):337–355, August 1992. 91, 92, 97

[BLFM98] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. Internet Engineering Task Force, Network Working Group, August 1998. IETF RFC 2396. 83, 84

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996. 17, 70, 72

[BMR+00] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1 of *Software Design Patterns*. John Wiley & Sons, 2000. 70, 72

[Boh04] Matthias Bohlen. AndroMDA. Project homepage: `http://andromda.org/`, 2004. 64, 69, 132, 135

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin / Cummings, Redwood City, CA, USA, 2nd edition, 1994. 59

[Bor01] Jerry Bortvedt. JCache - Java Temporary Caching API. Java Specification Request #107, 19 March 2001. 96, 149

[Bra04] Gilad Bracha. A metadata facility for the Java programming language. Java Specification Request #175, 13 September 2004. 29, 68

[Bro01] Kyle Brown. Session bean wraps entity beans. Portland Pattern Repository `http://c2.com/ppr`, February 2001. 94

[BV03] Bernd Brügge and Christoph Vilsmeier. Reducing CORBA call latency by caching and prefetching. *IEEE Distributed Systems Online*, June 2003. 103, 108, 127

[BW98] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, September/October 1998. 15, 16

[CCE+03] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In Endler and Schmidt [ES03], pages 242–261. 98, 165

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL)*. W3C, 1.1 edition, 15 March 2001. 43

[CD00] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Component Software Series. Addison-Wesley, October 2000. 16, 17, 42, 63, 64

[CDFV00] Gregory Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. Implementing a caching service for distributed CORBA objects. In Sventek and Coulson [SC00], pages 1–23. 95

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 1st edition, 6 June 2000. 55, 75

[Cer02] Ethan Cerami. *Web Service Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly, February 2002. 44

[CMZ02] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, pages 246–261, Seattle, WA, USA, 4–8 November 2002. ACM SIGPLAN, ACM Press. 98, 165

[Cod70] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970. 19, 48

[Cou01] Geoff Coulson. What is reflective middleware? *IEEE Distributed Systems Online*, 2(8), December 2001. 72

[CPS93] Steve J. Caughey, Graham D. Parrington, and Santosh K. Shrivastava. SHADOWS - a flexible support system for objects in distributed systems. In *Proceedings of the 3rd International Workshop on Object Orientation and Operating Systems (IWOODS'93)*, pages 73–82, Asheville, NC, USA, 1993. 91

[DeM03] Linda G. DeMichiel. *Enterprise JavaBeans Specification Version 2.1*. Sun Microsystems, final release edition, 12 November 2003. 22, 23, 29, 42

[DeM04] Linda G. DeMichiel. *Enterprise JavaBeans Specification Version 3.0*. Sun Microsystems, early draft edition, 24 June 2004. 22, 29, 69, 77

[DFJ+96] Shaul Dar, Michael J. Franklin, Björn Thór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB'96)*, pages 330–341, Mumbai (Bombay), India, 3–6 September 1996. Very Large Data Base Endowment, Morgan Kaufmann. 10, 88

[DL03] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In Stefani et al. [SDH03], pages 1–14. 57, 75, 94

[DMP+02] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 7(5):50–58, September 2002. 84, 85, 103

[DRS98] Nigel Davies, Kerry Raymond, and Jochen Seitz, editors. *Middleware 1998—International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, England, 15–18 September 1998. IFIP, Springer. 172, 176

[DSO03] Introduction to message-oriented middleware (MOM). *IEEE Distributed Systems Online*, 31 August 2003. 19, 25

[DYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification Version 2.0*. Sun Microsystems, final release edition, 14 August 2001. 22, 25, 26, 28, 29, 93, 151

[EAB02] Tzilla Elrad, Omar Aldawud, and Atef Bader. Aspect oriented modeling—bridging the gap between design and implementation. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 189–201, Pittsburgh, PA, USA, 6–8 October 2002. ACM SIGPLAN/SIGSOFT, Springer. 76

[ES03] Markus Endler and Douglas C. Schmidt, editors. *Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, Rio de Janeiro, Brazil, 16–20 June 2003. ACM / IFIP / USENIX, Springer. 173, 175, 185

[ET01] John Eberhard and Anad Tripathi. Efficient object caching for distributed Java RMI applications. In R. Guerraoui, editor, *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 15–35, Heidelberg, Germany, 2001. ACM / IFIP / USENIX, Springer. 92, 96, 98, 108, 110

[FBLL02] Robert E. Filman, Stuart Barrett, Diana D. Lee, and Ted Linden. Inserting ilities by controlling communications. *Communications of the ACM*, 45(1):116–122, 2002. 18, 75, 94

[FC94] Michael J. Franklin and Michael J. Carey. Client-server caching revisited. In M. Tamer Özsu, Umeshwar Dayal, and Patrick Valduriez, editors, *Distributed Object Management*, pages 57–78. Morgan Kaufmann, 1994. Papers from the International Workshop on Distributed Object Management (IWDOM) 1992. 87, 89

[FCL97] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Transactions on Database Systems (TODS)*, 22(3):315–363, September 1997. 11, 89

[FF03] Martin Fowler and Matthew Foemmel. Continuous integration. ThoughtWorks Whitepaper, `http://martinfowler.com/articles/continuousIntegration.html`, 2003. 61, 67

[FGM+99] Roy T. Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk Nielsen, Paul Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, Network Working Group, June 1999. IETF RFC 2616. 13, 23, 43, 84

[Fow03] Martin Fowler. *A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3rd edition, 19 September 2003. 59

[FR03] Marc Fleury and Francisco Reverbel. The JBoss extensible server. In Endler and Schmidt [ES03], pages 344–373. 29, 74, 98, 153, 156

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages (TOPLAS)*, 7(1):80–112, January 1985. 19

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, August 1994. 26, 27, 70, 72, 74, 91, 92, 148, 149

[GJL97] Steven D. Gray, Roger Jennings, and Rick A. Lievano. *Microsoft Transaction Server 2.0*. Roger Jennings' Database Workshop. Sams Publishing, 1 December 1997. 40

[GKW02] David Garlan, Jeff Kramer, and Alexander L. Wolf, editors. *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*, Charleston, SC, USA, 18–19 November 2002. ACM SIGSOFT, ACM Press. Collocated with the FSE-10. 57, 58, 176

[GLP75] Jim Gray, Raymond A. Lorie, and Gianfranco R. Putzulo. Granularity of locks and degrees of consistency in a shared database. In *Proceedings of the 1st International Conference on Very Large Data Bases (VLDB'75)*, Framingham, MA, USA, 22–24September 1975. ACM. IBM Research Report RJ1654. 47

[Gre01] Jack Greenfield. *UML Profile for EJB*. Rational Software Corporation, public draft edition, 28 August 2001. Java Specification Request 26: UML/EJB Mapping Specification. Withdrawn 29 Mar, 2004. 63, 64, 133, 135

[Gru97] Robert E. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, February 1997. Technical Report MIT/LCS/TR-708. 11, 89, 119

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In Garlan et al. [GKW02], pages 27–32. Collocated with the FSE-10. 57, 58

[Här84] Theo Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984. Special issue on databases: Their creation, management and utilization. 49

[HB01] Daniel Hagimont and Fabienne Boyer. A configurable RMI mechanism for sharing distributed java objects. *IEEE Internet Computing*, 5(1):36–43, January / February 2001. 97, 149

[HB04] Theo Härder and Andreas Bümann. Datenbank-Caching – Eine systematische Analyse möglicher Verfahren. *Informatik – Forschung und Entwicklung*, 19(1), 2004. 87, 89, 101, 145

[HBG+98] Franz J. Hauck, Ulrich Becker, Martin Geier, Erich Meier, Uwe Rastofer, and Martin Steckermeier. AspectIX: An aspect-oriented and CORBA-compliant ORB architecture. Technical Report TR-I4-98-08, Univ. of Erlangen-Nuernberg, IMMD IV, 1998. 97

[HL98] Daniel Hagimont and D. Louvegnies. Javanaise: Distributed shared objects for Internet cooperative applications. In Davies et al. [DRS98], pages 339–354. 97

[HO93] Willian Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In Andreas Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, number 10 in SIGPLAN Notices 28, pages 411–428, Washington, DC, USA, October 1993. ACM Press. 75

[HOV04] Richard Hightower, Warner Onstine, and Paul Visan. *Professional Java Tools for Extreme Programming: Ant, XDoclet, JUnit, Cactus and Maven*. Wrox, April 2004. 67, 133

[HR83] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, December 1983. 46

[HSW94] Yixiu Huang, Robert H. Sloan, and Ouri Wolfson. Divergence caching in client-server architectures. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS '94)*, pages 131–139, Austin, TX, USA, September 1994. IEEE. 86

[IC98] Arun Iyengar and Jim Challenger. Data update propagation: A method for determining how changes to underlying data affect cached objects on the web. Technical Report RC 21093(94368), IBM Research, T.J.Watson Research Center, Yorktown Heights, NY, USA, February 1998. 96

[ISO95] ISO/IEC, ITU-T, Geneve, Switzerland. *Open Distributed Processing Reference Model—Part 3: Architecture*, 1995. International Standard 10746-3 / ITU-T Recommendation X.903. 4, 18, 22, 72

[Iye99] Arun Iyengar. Design and performance of a general-purpose software cache. In *Proceedings of the 18th IEEE International Performance Conference (IPCCC'99)*, 1999. 96

[Jac91] Ivar Jacobson. *Object-oriented software engineering*. ACM Press, New York, NY, USA, 1st edition, 1991. 59

[JBo04] JBoss Group. *JBoss*, 2004. Project homepage `http://www.jboss.org/`. 74

[JdT+95] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'95)*, number 5 in Operating System Review 29, pages 156–171, Copper Mountain Resort, CO, USA, 3–6 December 1995. ACM Press. 99

[JM02] Karsten Januszewski and Ed Mooney. *UDDI Features List*. OASIS, 3rd edition, 2002. 43

[KAD96] Rammohan Kordale, Mustaque Ahamad, and Murthy V. Devarakonda. Object caching in a CORBA compliant system. *Computing Systems*, 9(4):377–404, 1996. 95

[KB94] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. In *Proceedings of the 3rd International Conference on on Parallel and Distributed Information Systems (PDIS'94)*, pages 229–238, Austin, TX, USA, 28–30 September 1994. IEEE Computer Society Press. 10, 88

[KC03] Jeffrey O. Keffart and David M. Chess. The vision of autonomic computing. *Computer Magazine*, 36(1):41–50, January 2003. 57

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002. Special issue on adaptive middleware. 71, 72

[KGDA00] Vijaykumar Krishnaswamy, Ivan B. Ganev, Jaideep M. Dharap, and Mustaque Ahamad. Distributed object implementations for interactive applications. In Sventek and Coulson [SC00]. 96

[KK00] Rainer Koster and Thorsten Kramp. Loadable smart proxies and native code-shipping for CORBA. In *Proceeding of the 3rd International IFIP/GI Working Conference on Universal Service Market USM 2000*, volume 1890 of *Lecture Notes in Computer Science*, pages 202–213, Munich, Germany, 2000. Springer. 73

[Kla04] Sebastian Klamar. Adaptive Architekturen für verteilte Systeme. Diplomarbeit, Technische Universität Dresden, Dresden, Germany, 30 September 2004. 57

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages

220–242, Jyväskylä, Finland, 1997. AITO / ACM SIGPLAN, Springer. 75

[Kob04] Cris Kobryn. UML 3.0 and the future of modeling. *Software and Systems Modeling*, 3(1):4–8, March 2004. 59

[KRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. 72

[KRB01] Vijaykumar Krishnaswamy, Michel Raynal, and David E. Bakken. Shared state consistency for time-sensitive distributed applications. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*, pages 606–614, Phoenix, AZ, USA, 16–19 April 2001. IEEE Computer Society Press. 96

[Kru00] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 3nd edition, 19 December 2000. 61

[KS91] James. J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, number 5 in Operating Systems Review 25, pages 213–225, Pacific Grove, CA, USA, 3–6 December 1991. ACM Press. 102

[Kue97] Geoffrey H. Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation*. PhD thesis, University of California, Los Angeles, CA, USA, May 1997. Technical Report UCLA-CSD-970015. 102

[KW97] Balachander Krishnamurthy and Craig E. Wills. Study of piggyback cache validation for proxy caches in the world wide web. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, 8–11 December 1997. 85, 119, 166

[KW98] Balachander Krishnamurthy and Craig E. Wills. Piggyback server invalidation for proxy cache coherency. *Computer Networks*, 30(1–7):185–193, April 1998. Proceedings of the 7th International World Wide Web Conference . 85, 119, 166

[KWB+98] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java remote method invocation (RMI). In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS'98)*, Santa Fe, NM, USA, 27–30 April 1998. USENIX. 95, 98, 108

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture—Practice and Promise*. Addison-Wesley, 1st edition, 25 April 2003. 66

[KWZ90] Klaus Kratzer, Hartmut Wedekind, and Georg Zörntlein. Prefetching—a performance analysis. *Information Systems*, 15(4):445–452, 1990. Pergamon Press. 101

[Len97] Richard Lenz. *Adaptive Datenreplikation in Verteilten Systemen*, volume 23 of *Teubner-Texte zur Informatik (TTzI)*. Teubner Verlag, Leipzig, Germany, 1st edition, 1997. 10, 46, 50, 51, 52, 86, 87

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996. 54, 55

[Lös02] Frank Löschau. Realisierung kontextsensitiver Anwendungen auf Basis eines Frameworks zur Kontextverarbeitung und -verwaltung. Diplomarbeit, Technische Universität Dresden, Dresden, Germany, 9 October 2002. 3

[Lot00] Tony Loton. The smart approach to distributed performance monitoring with Java. *JavaWorld*, September 2000. 95, 149, 152

[LS88] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'88)*, number 7 in SIGPLAN Notices 23, pages 260–267, Atlanta, GA, USA, 22–24 July 1988. ACM Press. 99, 122

[LZ77] Abraham Lempel and Jacob Ziv. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977. 101

[Mae87] Pattie Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels, Belgium, January 1987. 71

[Mar06] Andrei Andreevich Markov. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete*, 15(2):135–156, 1906. (Extension of the limit theorems of probability theory to a sum of variables connected in a chain). 101

[McA96] Jeff McAffer. Meta-level architecture support for distributed objects. In Gregor Kiczales, editor, *Proceedings of Reflection'96 Conference*, pages 39–62, San Francisco, CA, USA, 1996. Published before in IWOODS'95. 71, 72

[MCC99] Paul Martin, Victor Callaghan, and Adrian Clark. High performance distributed objects using caching proxies for large scale applications. In *Proceedings of the 1st International Symposium on Distributed Objects and Applications (DOA'99)*, pages 110–119, Edinburgh, UK, 1999. 91

[McI64] M. Douglas McIlroy. Pipes and filters. Internal Bell Labs memo, original title lost, 11 October 1964. see `http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.html`. 17, 70

[McI69] M. Douglas McIlroy. "Mass produced" software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, Belgium, 1969. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. 15

[MH98] Vlada Matena and Mark Hapner. *Enterprise JavaBeans Specification Version 1.0*. Sun Microsystems, 21 March 1998. 22, 28, 93

[MH99] Vlada Matena and Mark Hapner. *Enterprise JavaBeans Specification Version 1.1*. Sun Microsystems, final release edition, 24 November 1999. 22, 28, 64, 93, 151

[Mit03] Nilo Mitra. *SOAP Part 0: Primer*. W3C, 1.2 edition, 24 June 2003. 43

[MM03] Joaquin Miller and Jishnu Mukerji. *MDA Guide*. Object Management Group, 1.0.1 edition, 12 June 2003. omg/03-06-01. 65

[Moc87] Paul Mockapetris. *Domain Names—Concepts and Facilities*. Internet Engineering Task Force, Network Working Group, November 1987. IETF RFC 1034. 84

[Moh01] C. Mohan. Caching technologies for Web applications. In *Tutorial at VLDB Conference 2001*, Rome, Italy, 2001. `http://www.almaden.ibm.com/u/mohan/Caching_VLDB2001.pdf`. 81, 86, 87

[MSa] Microsoft. *ActiveX Controls*. `http://www.microsoft.com/com/tech/ActiveX.asp`. 39

[MSb] Microsoft. *COM+*. `http://www.microsoft.com/com/tech/COMPlus.asp`. 40

[MSc] Microsoft. *Component Object Model (COM)*. `http://www.microsoft.com/com/tech/com.asp`. 38

[MSd] Microsoft. *Distributed Common Object Model (DCOM)*. `http://www.microsoft.com/com/tech/dcom.asp`. 19, 39, 99

[MSe] Microsoft. *Microsoft Transaction Server (MTS)*. `http://www.microsoft.com/com/tech/MTS.asp`. 39

[MSf] Microsoft. *.NET Framework*. `http://msdn.microsoft.com/netframework/`. 40, 42

[MSg] Microsoft. *Visual C# Language Specification*. `http://msdn.microsoft.com/library/en-us/csspec/html/CSharpSpecStart.asp`. 40, 69

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. 17, 37, 57

[MW88] Klaus Meyer-Wegener. *Transaktionssysteme: Funktionsumfang, Realisierungsmöglichkeiten, Leistungsverhalten*. Leitfäden der angewandten Informatik. B. G. Teubner, Stuttgart, Germany, 1988. 18, 19, 46

[Neu03] Olaf Neumann. *Wiederverwendbare Komponenten für eLearning*. Dissertation, Technische Universität Dresden, Dresden, Germany, 2003. 3, 148, 162

[NPF99] Olaf Neumann, Christoph Pohl, and Katrin Franze. Caching in Stubs und Events mit Enterprise Java Beans bei Einsatz einer objektorientierten Datenbank. In Clemens H. Cap, editor, *Java-Informations-Tage JIT'99*, Informatik Aktuell, pages 17–25, Düsseldorf, Germany, 20–21 September 1999. Springer. 107, 117, 147

[Oes02] Bernd Oestereich. *Developing Software with UML: Object-Oriented Analysis and Design in Practice*. Addison-Wesley, 2nd edition, June 2002. 16, 61

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999. 57, 58

[OMG99] Object Management Group. *White Paper on the Profile Mechanism*, April 1999. ad/99-04-07. 63, 64

[OMG01] Object Management Group. *CORBA Portable Interceptor Specification*, March 2001. ptc/01-03-04, formal/02-05-18. 73, 74, 156

[OMG02a] Object Management Group. *CORBA Components*, version 3.0 edition, June 2002. formal/02-06-65. 29, 31, 37

[OMG02b] Object Management Group. *UML Profile for Enterprise Distributed Object Computing (EDOC)*, February 2002. ptc/2002-02-05. 63, 64, 133

[OMG03a] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification*, October 2003. ptc/03-10-04. 41, 59, 60, 62

[OMG03b] Object Management Group. *UML 2.0 Infrastructure Specification*, September 2003. ptc/03-09-15. 16, 59, 61, 62, 63

[OMG03c] Object Management Group. *UML 2.0 Superstructure Specification*, August 2003. ptc/03-08-02. 59, 61, 63, 64, 133, 135, 140, 168

[OMG03d] Object Management Group. *Unified Modeling Language Specification, v1.5*, March 2003. formal/03-03-01. 59, 61

[OMG03e] Object Management Group. *XML Metadata Interchange (XMI Specification*, 2.0 edition, May 2003. formal/03-05-02. 60, 132

[OMG04a] Object Management Group. *Common Object Request Broker Architecture: Core Specification*, v3.0.3 edition, March 2004. formal/04-03-12. 19, 20, 30, 33, 39, 41, 73

[OMG04b] Object Management Group. *Metamodel and UML Profile for Java and EJB Specification*, 1.0 edition, February 2004. formal/04-02-02. 63, 64

[OMG04c] Object Management Group. *UML Profile for CORBA Components Specification*, March 2004. ptc/04-03-04. 35, 63

[ÖSA+03] Rickard Öberg, Andreas Schaefer, Ara Abrahamian, Aslak Hellesøy, Dmitri Colebatch, and Vincent Harcq. XDoclet. Project homepage: `http://xdoclet.net/`, 2003. 67, 68

[Out04] Outsource Cafe. *JavaGen: Automated Software Development*, 2004. Project homepage `http://www.javagen.com/`. 69

[ÖV99] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 1999. 46, 51

[ÖVU98] M. Tamer Özsu, Kaladhar Voruganti, and Ronald C. Unrau. An asynchronous avoidance-based cache consistency algorithm for client caching DBMSs. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB'98)*, pages 440–451, New York, NY, USA, 24–27 August 1998. Very Large Data Base Endowment, Morgan Kaufmann. 89

[PB03] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, December 2003. 12, 84, 85, 86

[PDF+02] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. A UML notation for aspect-oriented software design. `http://jac.aopsys.com/papers/uml_short/uml.html`, 3 April 2002. 76, 77

[PDN86] Ruben Prieto-Diaz and James Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, November 1986. 16, 17

[Pen04] Srini Penchikala. J2EE object-caching frameworks. *JavaWorld*, May 2004. 96

[Pfe04a] Daniel Pfeifer. Eine Algebra für Cache-Modelle zum methodenbasierten Caching im Applikationsserver-Bereich. Technical Report 2004-3, IPD, Universität Karlsruhe, 10 February 2004. 98, 110

[Pfe04b] Daniel Pfeifer. Transaktionales Methoden-Caching im Applikationsserver-Berreich. Technical Report 2004-13, IPD, Universität Karlsruhe, 6 August 2004. 89, 90, 98, 120

[PG03] Christoph Pohl and Steffen Göbel. Integrating orthogonal middleware functionality in components using interceptors. In Klaus Irmscher and Klaus-Peter Fähnrich, editors, *Kommunikation in Verteilten Systemen (KiVS 2003)*, Informatik Aktuell, pages 345–358, Leipzig, Germany, February 2003. VDE/ITG & GI, Springer. 55, 155, 156, 160

[PJ03] Daniel Pfeifer and Hannes Jakschitsch. Method-based caching in multi-tiered server applications. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003—Proceedings of the OTM Confederated International Conferences CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1312–1332, Catania, Sicily, Italy, 3–7 November 2003. Springer. 98, 165, 169

[Poh99] Christoph Pohl. Entwurfsrichtlinien zum komponentenbasierten Aufbau eines Teleteaching-Systems. Großer Beleg (term paper), Technische Universität Dresden, 15 December 1999. In German. 30, 45

[Poh03] Christoph Pohl. Adaptively caching distributed components. In *Middleware2003 Companion*, page 325, Rio de Janeiro, Brazil, June 2003. PUC-Rio. 108, 112, 118, 156

[Pro02] Jeff Prosise. *Programming Microsoft .NET*. Microsoft Press, 2002. 41

[PS02] Christoph Pohl and Alexander Schill. Middleware support for transparent client-side caching. *Electronic Notes in Theoretical Computer Science*, 65(4), 7 April 2002. Software Composition Workshop (SC 2002) at European conference on Theory And Practice of Software (ETAPS'02), Grenoble, France. 107, 110, 120, 131, 148, 154, 155

[PS03] Christoph Pohl and Alexander Schill. Client-side component caching. In Stefani et al. [SDH03], pages 141–152. 108, 110, 112, 118, 120, 126, 156

[PTM97] Jelica Protic, Milo Tomaevic, and Veljko Milutinovic. *Distributed Shared Memory: Concepts and Systems*. John Wiley & Sons, 1997. 20

[PTS05] Christoph Pohl, Boon Chong Tan, and Alexander Schill. Semantic caching of code archives. In *Kommunikation in Verteilten Systemen (KiVS 2005)*, Informatik Aktuell, Kaiserslautern, Germany, 28 February–02 March 2005. VDE/ITG & GI, Springer. To appear. 88

[Rah94] Erhard Rahm. *Mehrrechner-Datenbanksystem: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison-Wesley, 1994. 46, 50, 51

[RAJM01] Ed Roman, Scott Ambler, Tyler Jewell, and Floyd Marinescu. *Mastering Enterprise JavaBeans*. John Wiley & Sons, 2nd edition, 14 December 2001. 24, 25, 27, 28

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991. 59

[Res01] Martijn Res. Reduce EJB network traffic with astral clones. *JavaWorld*, January 2001. 93

[Rog97] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997. 39

[RR76] Juan Rodriguez-Rosell. Empirical data reference behavior in data base systems. *IEEE Computer*, 9(11):9–13, November 1976. 101

[RS02] Michael Rabinovich and Oliver Spatscheck. *Web Caching and Replication*. Addison-Wesley, January 2002. 12, 83, 84, 85, 86, 100, 101, 102, 103, 119, 166

[RW02] Berthold Reinwald and Sanjiva Weerawarana. Web services framework. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA, USA, 26 February–1 March 2002. IEEE, IEEE Computer Society Press. 44

[SC00] Joseph Sventek and Geoff Coulson, editors. *Middleware 2000—International Conference on Distributed Systems Platforms and Open Distributed Processing*, volume 1795 of *Lecture Notes in Computer Science*, New York, NY, USA, 4–8 April 2000. IFIP WG6.1 / ACM, Springer. 173, 177

[SCA] Scientific Computing Associates. *Linda Tuple Space*. `http://lindaspaces.com/`, originally developed by the Yale Linda Group. 19

[Sch02] Andreas Schaefer. JBoss: an in-depth look at the interceptor stack. *ONJava.com: The O'Reilly Network*, 24 July 2002. 74

[Sch03] Daniel Schaller. Nebenläufige Aktualisierung teil-replizierter Datenbestände bei Transaktionsmigration. Diplomarbeit, Technische Universität Dresden, 15 June 2003. 50, 90

[SDH03] Jean-Bernard Stefani, Isabelle Demeure, and Daniel Hagimont, editors. *Proceedings of the 4th International Conference on Distributed Applications and Interoperable Systems (DAIS 2003)*, volume 2893 of *Lecture Notes in Computer Science*, Paris, France, 19–21 November 2003. IFIP WG 6.1, Springer. 174, 182

[SDMML03] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Web cache prefetching as an aspect: Towards an dynamic-weaving based solution. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 110–119, Boston, MA, USA, 17–21 March 2003. ACM. 77, 102

[Sha86] Mark Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems (ICDCS'86)*, pages 198–204, Cambridge, MA, USA, 19–23 May 1986. IEEE Computer Society Press. 72

[Sha03] Bill Shannon. *Java 2 Platform Enterprise Edition Specification, v1.4*. Sun Microsystems, final release edition, 24 November 2003. 22, 23, 42

[Smi78] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3):223–247, September 1978. 101

[Smi82] Brain Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, USA, February 1982. 71

[Smi01] Wayne D. Smith. *TPC-W: Benchmarking An Ecommerce Solution*. The Transaction Processing Performance Council, v1.2 edition, 2001. 98, 165

[Spr04] Thomas Springer. *Ein komponentenbasiertes Meta-Modell kontextabhängiger Adaptionsgraphen für mobile und ubiquitäre Anwendungen*. Dissertation, Technische Universität Dresden, Dresden, Germany, March 2004. 53, 55, 56

[SS01] Richard E. Schantz and Douglas C. Schmidt. *Encyclopedia of Software Engineering*, chapter Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. John Wiley & Sons, 2001. 19

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley & Sons, 2000. 20, 27, 70, 73, 74

[ST00] Rob Stevenson and Leonard Theivendra. Developing EJB Access Beans in VisualAge for Java. Whitepaper, IBM Toronto Lab, October 2000. 93

[Sun] Sun Microsystems. *Jini Network Technology*. `http://wwws.sun.com/software/jini/`. 19

[Sun02a] Sun Microsystems. *Java Remote Method Invocation Specification*, revision 1.8 edition, 2002. General information at `http://java.sun.com/products/jdk/rmi/`. Available as part of Java 2 SDK Standard Edition at `ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf`. 19, 95, 97, 100

[Sun02b] Sun Microsystems. *JavaBeans Technology*, 2002. `http://java.sun.com/products/javabeans/`. 17, 22, 30, 39, 69, 92, 93, 98

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley, 2nd edition, 2002. 15

[Tan] tangible architect. Product homepage `http://www.tangible.de/`. Tangible engineering: Generative development for .NET. 69

[TAO] Real-time CORBA with TAO. Project homepage `http://www.cs.wustl.edu/~schmidt/TAO.html`. Washington University, St. Louis and University of California, Irvine. 73

[Tea02] TimesTen Team. Mid-tier caching: The TimesTen approach. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the International Conference on Management of Data*, pages 588–593, Madison, WI, USA, 3–6 June 2002. ACM SIGMOD, ACM Press. 88, 101

[TVJ+01] Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Nørregaard Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 233–242, Toronto, Ontario, Canada, 2001. IEEE. 75

[Vit01] Roman Vitenberg. Caching support for CORBA objects. In Maarten van Steen, editor, *Distributed Systems Online*, Research from the Trenches. IEEE Computer Society Press, March 2001. 95

[Vog03] Werner Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, November–December 2003. See also `http://weblogs.cs.cornell.edu/AllThingsDistributed/archives/000343.html`. 44

[vSHT97] Marteen van Steen, Philip Homburg, and Andrew S. Tanenbaum. The architectural design of Globe: A wide-area distributed system. Technical Report IR-422, Vrije Universiteit Amsterdam, Netherlands, 1997. 97

[WFN90] Edward F. Walker, Richard Floyd, and Paul Neves. Asynchronous remote operation execution in distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 253–259, Paris, France, 28 May – 1 June 1990. 99, 122

[WPSO01] Nanbor Wang, Kirthika Parameswaran, Douglas C. Schmidt, and Ossama Othman. The design and performance of meta-programming mechanisms for object request broker middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, January 2001. 73

[WRÖ03] Craig Walls, Norman Richards, and Rickard Öberg. *XDoclet in Action*. In Action. Manning Publications, December 2003. 68

[WZ86] Hartmut Wedekind and Georg Zörntlein. Prefetching in realtime database applications. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 215–226. ACM Press, 1986. 101

[Yeu04] Kwok Cheung Yeung. *Dynamic performance optimisation of distributed Java applications*. PhD thesis, Imperial College, University of London, March 2004. 100

[YK03] Kwok Cheung Yeung and Paul H. J. Kelly. Optimising Java RMI programs by communication restructuring. In Endler and Schmidt [ES03], pages 324–343. 100

[ZBS97] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997. 96

[ZC96] Matthew J. Zelesko and David R. Cheriton. Specializing object-oriented RPC for functionality and performance. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, Hong Kong, May 1996. IEEE Computer Society Press. 91