Chapter 1: Overview

Design Methods: Some Principles
Abstraction
Visualisation
Modeling
Divide & Conquer
Branch & Bound
OO: Cohesion & Coupling, Separation of Concerns, IoC (Dependency Injection)

Design Method vs. Procedure Model
Design Method=The "what" gets translated into the "How"
Examples: Structured Design (SD), Object-oriented design (OOD)
Procedural Model=Rule set for creating software from inception to delivery
Examples: Waterfall Model, Spiral Model, RUP, V-Modell, Agile Models

Verteiltes System
Zusammenschluss unabhängiger Computer bzw. Prozesse, die über Nachrichten miteinander kommunizieren

Why Distributed Systems?
1. Problem-related reasons:
Because the task requires it (e.g. a user in europe wants to see a web page from a server in USA)
2. Property-related reasons:
-Performance and Scalability
-Fault Tolerance
-Service and Client Location Independence
-Maintainability and Deployment
-Security
-Business Integration

Challenges of Distributed Systems:
1. Network Latency
2. Predictability (how long an invocation is running because of network latency etc.)
3. Concurrency: "Real concurrency"
4. Scalability:
5. Partial Failure: If a part of the system fails, the rest should stay working
ONLY DISTRIBUTE YOUR SYSTEM IF YOU HAVE TO!! -> many problems arise

CAP
We would like to have all three of CAP
1. Consistency
2. Availability
3. Partition Tolerance
BUT: Brewer's (CAP) Theorem formally proves, that you only can have two!!

ACID vs. BASE
ACID:
-Atomicity
-Consistency
-Isolation
-Durability
Availability less important, Strong consistency, pessimistic

BASE:
- Basically Available
-Soft-state (=scalability)
-Eventually Consistent
high availability, but weak consistency, optimistic, High scaling possible

Generally: Consistency is a hard requirement; when sacrificed, life's easy.

Application areas of Distributed Systems (DS):
-Internet
-Telecommunication Networks
-International financial transactions
-Scientific applications

Distributed Systems (DS) Examples:
-google search engine
-eBay
-Amazon
-Facebook

Middleware

-abstraction
-user shall not have to deal with low level networking problems because error prone, hard to maintain, NOT TRANSPARENT

Middleware Examples:
-DCOM
-CORBA
-RPC Models (RPC, RMI, NET WCF)
-JavaSpaces / Jini
-Enterprise Service Busses
-MoM (Message oriented Middleware)

Architectural styles:
-Client-Server (=2-Tier)
-N-Tier
-Cluster (tightly coupled)
-Peer-to-Peer (P2P)
-Distributed Storage
-Space based
-Service Oriented Architecture (SOA)
-Command/Query Responsibility Segregation (CQRS) by Martin Fowler

CQRS - Command Query Responsibility Segregation
Idea: Most applications read data more frequently than they write data.
So separate reading and writing (2 different services with different datastores)

Modeling Parallelism:
-Class Diagrams: Active classes
-Activity Diagrams: forking
-Sequence Diagrams
-State Diagrams

Message Oriented Middleware (MOM)
-Topics/Queues
-Persistent/Non-Persistend
-Reliable/Non-Reliable
-Point-To-Point/One-To-Many
-Transactions
-example: JMS

Service Oriented Architecture (SOA)
-SOAP
-WSDL
-Enterprise Service Bus (ESB)

Web Service Stack

Distributed Transactions
-involves more than one transactional resource, involves usually multiple machines
-Distributed Transactions implemented through Transaction processing monitors (TP Monitors)
--mark beginning and end of transaction in code
--at beginning client contacts the TP Monitor to get a transaction id and context
--at the end client again contacts TP Monitor
--TP Monitor runs a commit protocol (usually 2 phase commit = 2PC) to determine outcome of transaction
-through invoker and requestor pattern (at client and server side) or interceptors (transparently, both sides)

Server-side component infrastructures
-Container, in which the components of the distributed system run in. Container provides services as transactions, transparent communication, error handling etc.
-IoC
-often built on top of Middleware
-default implementations/ configuration for commonly used things

Other distributed infrastructures
-Peer-to-Peer (P2P): set of equal peers
-Grid Computing: is about sharing distributed resources, e.g. cpu time, storage, information

Chapter 2: Remoting Patterns

Design: Tasks
Identification of
-interfaces
-subsystems
-nonfunctional requirements

Analysis vs. Design
Analysis:
-focus on understanding the problem
-idealized design
-focus on functional requirements
Design:
-focus on understanding the solution
-many code near details
-also focus on non functional requirements

Remoting Style
Style, how the middleware system communicates:
-RPC
-Messages
-Shared Repository
-Streams (continuous)

Remoting Pattern
-Wiederbenutzbares Pattern, um EIN Problem zu lösen
Consists of
-Adressed Problem
-Adressed Forces
-Context
-Advantages and Disadvantages
-Relationships

Pattern Language

System of patterns with relations between. A distributed system is more complex than a single pattern. Main goal of a pattern language is to
step by step to improve the design of the overall system.

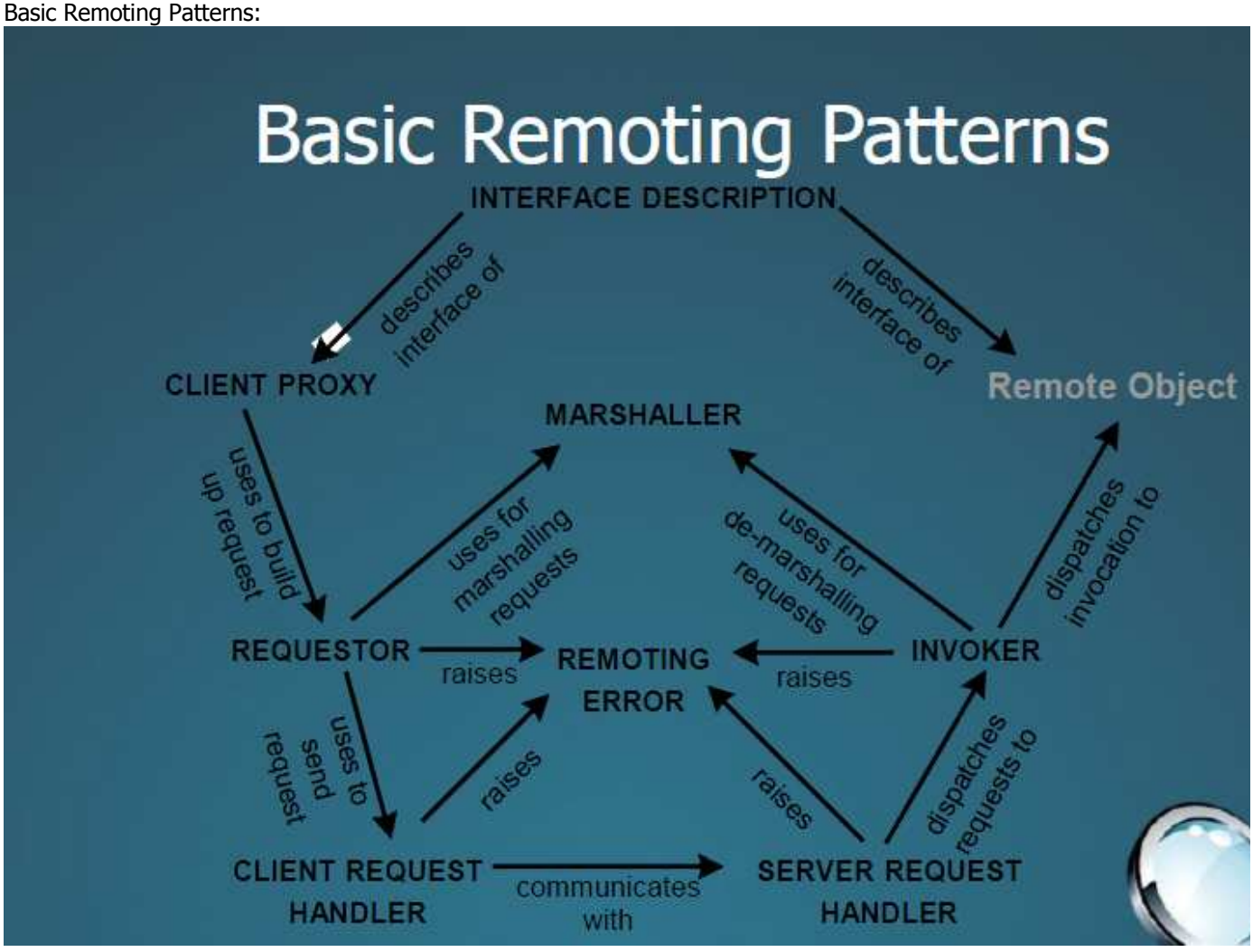
Remoting Patterns
Generally a pattern is:
A well-known and proved to be working Solution to a specific Problem in a certain Context.
A Three-Part Rule, which expresses a relation between a certain context, a certain system of forces and a certain software configuration which allows these forces to be resolved.

Broker
-Problem: Many challenges in distributed systems added let programmer loose their primary focus: to develop an application that solves a given task.
-Solution: Separate communication-related concerns in a broker, which hides and mediates the communication between all components of the distributed. Details are realized by other patterns.

Remote Object
-Problem: accessing an object over a network is different than a local access. Machine boundaries, network latency and unreliability.
-Solution: the application logic is provided in form of remote objects

Server Application
-Problem: Remote objects need an environment for creation and configuration of the distributed middleware
-Solution: A server application that provides the necessary environment, configures the middleware, instanciates the remote object and anvertises it.

Basic Remoting Patterns:



Requestor:

-Context: client needs to access one or more remote objects
-Problem: marshalling must be triggered, connection needs to be established, request information must be sent to the target Remote Object
-Solution: use a Requestor; the client tells the Requestor which remote object and which method to call and the parameters, the Requestor handles or delegates marshalling and connection handling

Client Proxy:
-Context: A Requestor is provided by the middleware
-Problem: We want programming with Remote Objects to be as transparent as possible - ideally no difference between Local Object and Remote
--location, method, arguments have to be passed manually
--Requestor does not support static type checking
-Solution: A Client Proxy supports the same Interface as the Remote Object. Clients only interact with the local Client Proxy, which translates local calls to its methods into remote calls to the specified Remote Object.

In .NET:
Transparent Proxy: The Client Proxy
Real Proxy: Plays role of the Requestor

Invoker
-Context: A requestor sends a remote invocation for a remote object to a server
-Problem: The target Remote Object on the Server has to be reached. The Remote Object should not have to deal with network connections and demarshalling
Solution: An Invoker accepts requests from Requestors, demarshals it, finds the correct local object and invokes the call on the Object.

Server Request Handler
-Context: Remote Objects and Invokers are used
-Problem: Network communication needs to be performed and coordinated for receiving requests.
-Solution: A Server Request Handler deals with the network communication issues. It receives the message from the network and dispatches the message to the correct invoker

Client Request Handler
-Context: Requestor is used.
-Problem: Requestor has to send requests and receive responses. Network connection management (especially time handling) as well as result handling and error detection are necessary.
-Solution: A Client Request Handler handles network connections for all Requestors within a Client.

Marshaller
-Context: Request and Response Messages have to be transported over the network
-Problem: Data to be sent to Remote Object contain different data. Over the network, only byte streams can be sent.
-Solution: A Marshaller marshalls an demarshalls the given data.

Interface Description
-Context: A client wants to invoke a Remote Object operation using a Client Proxy.
-Problem: Interfaces of Client Proxy and Remote Object have to be the same.
-Solution: An Interface Description describes the interfaces of Remote Objects. The Client Proxy and the Remote Object use either code generation (create code from Interface Definition for each Client Proxy/ Remote Object) or runtime configuration techniques (a Generic Client Proxy/Remote Object that is configured dynamically).

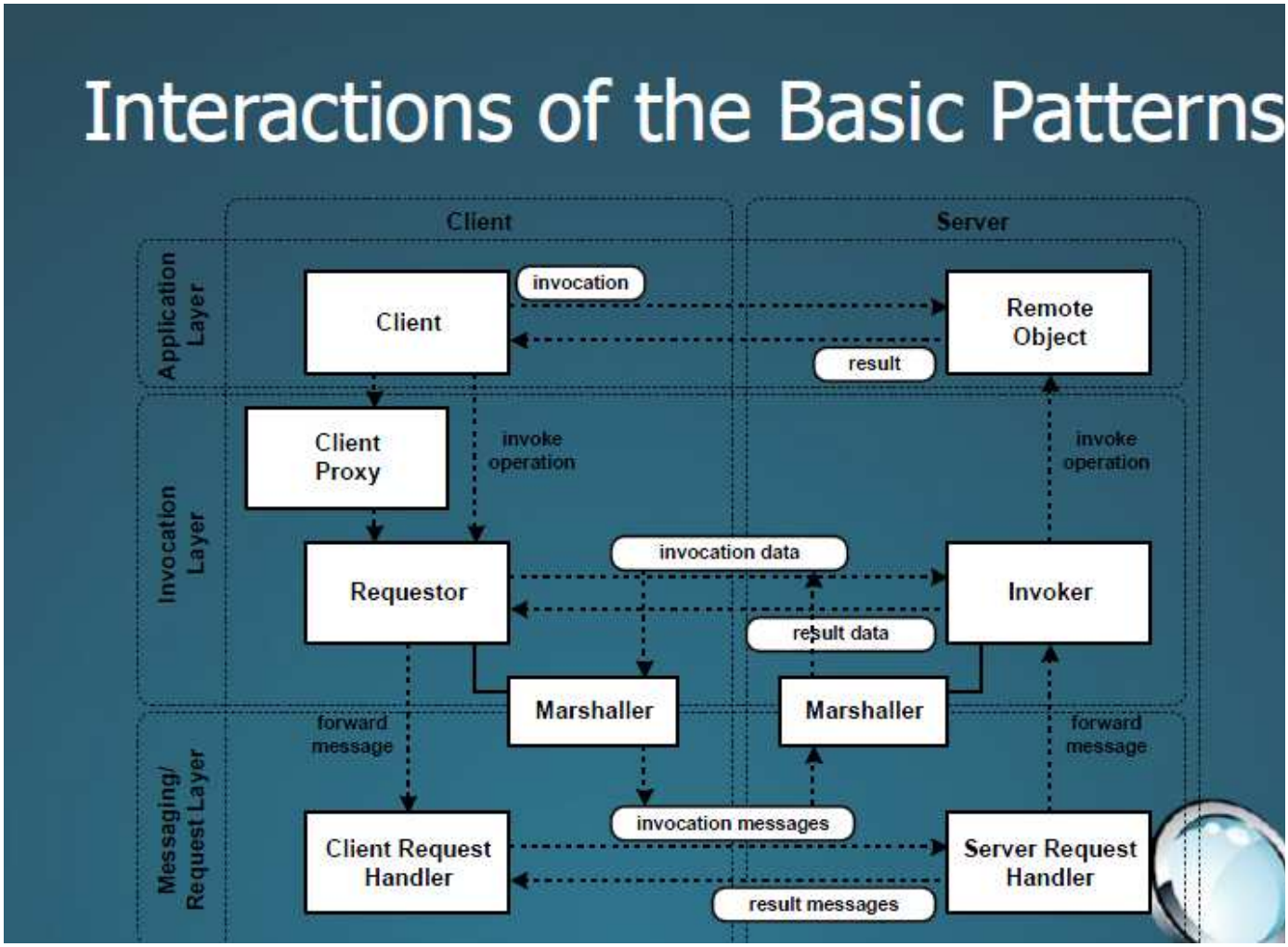
WSDL
WSDL is an Interface Description Language, which describes a service and its methods in an abstract form. Client Proxy and Remote Object can be created through code generation or a Generic Object can be configured accordingly.

IDL (in Corba)
Interface Description Language in Corba.

Remoting Error
-Context: Remote communication is unreliable.
-Problem: Network failures, server crashes and unavailable Remote Objects can occur anytime.
-Solution: Distinguish between Distribution-Related Errors and Application-Logic-Related Errors. Invoker, Requestor and

Request Handlers detect and propagate remoting errors.

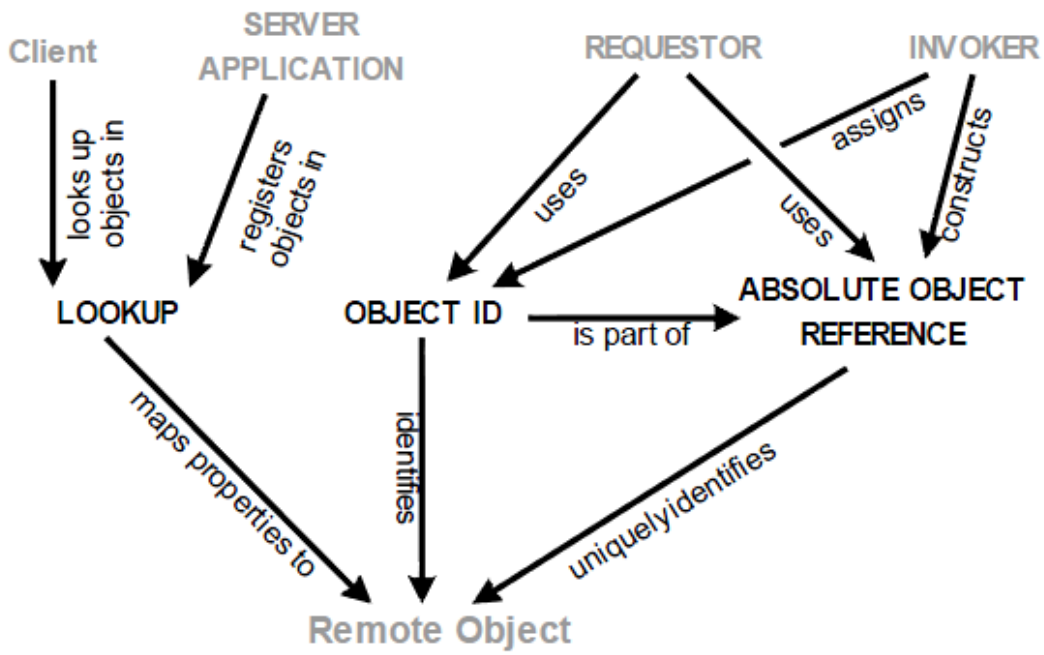
REST - Representational State Transfer
The basic idea is, that the client transfers from one state to the next by using URLs.
Representational State Transfer is an alternative to SOAP (Service Oriented Architecture) for realizing Web Services.
REST is not a standard, it's an Architectural Style.
REST is usually implemented through the use of:
-HTTP
-URLs
-XML (JSON)



Chapter 3

Identification Patterns

Identification Patterns



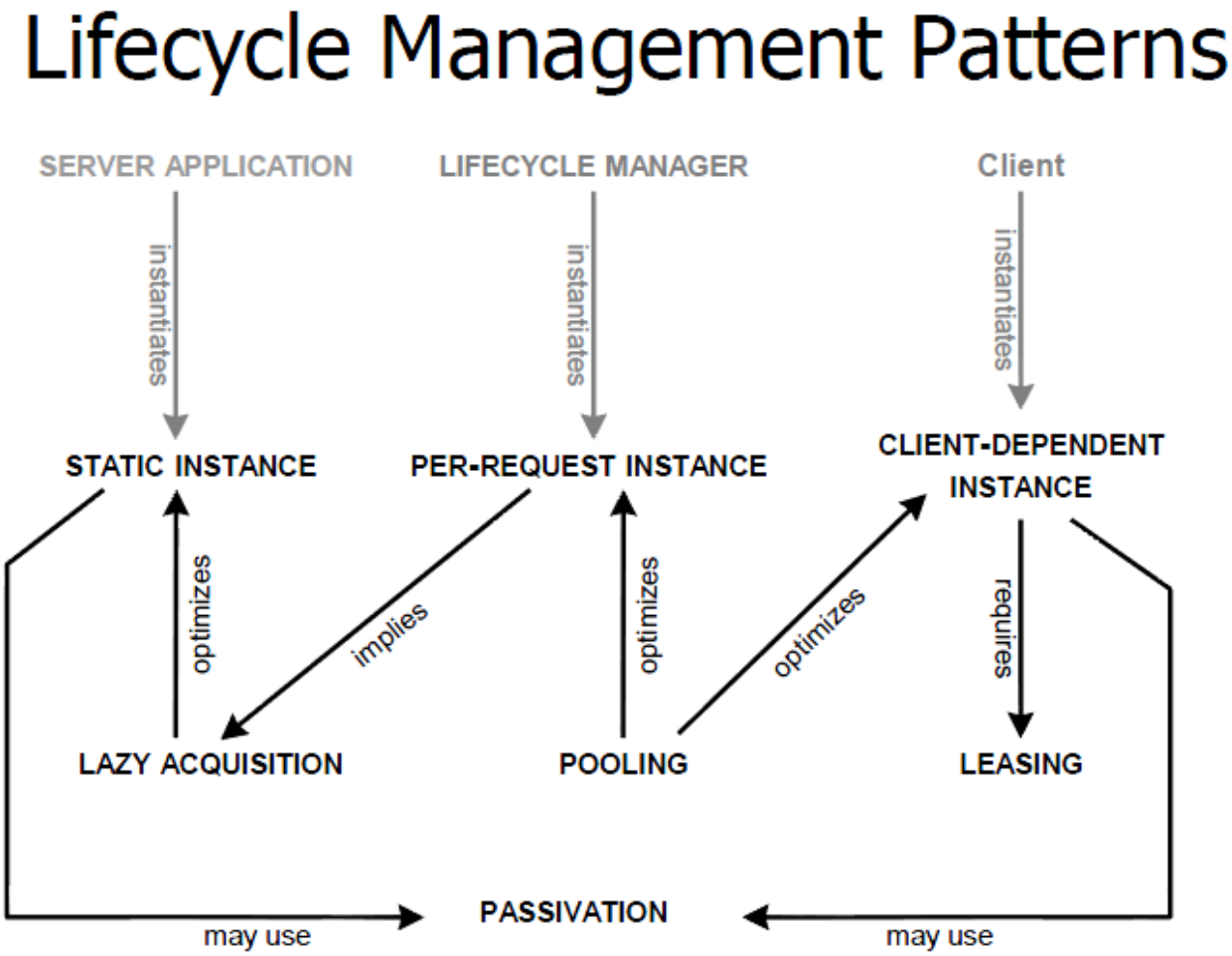
Object ID
-Context: The Invoker has to select the correct Remote Object to invoke.
-Problem: A Invoker handles invocations for many Remote Objects.
-Solution: Associate an unique Object ID to every Remote Object. This Object ID is unique for this invoker.

Absolute Object Reference
-Context: The Invoker uses Object IDs to find the correct Remote Object for invocations.
-Problem: The Invocation has to be delivered to the correct Server Request Handler.
-Solution: An Absolute Object Reference, which uniquely identifies Invoker and Remote Object, including:
--Endpoint information
--ID of the Invoker
--Object ID

Lookup
-Context: Client application wants to use a Remote Object
-Problem: Client has somehow to obtain the Absolute Object Reference. The Remote Object's location and availability may change.
-Solution: Servers Applications register their Remote Objects at Lookup Service, Client Applications use the Lookup Service to get the required Absolute Object References.

Examples of Lookup:
UDDI for looking up Web Services, CORBA Naming Service.

Lifecycle Management Patterns



Static Instance
-Context: Remote Objects are independent of specific Clients.
-Problem: Remote Objects have to be available a long period of time, their state must not be lost between invocations.
-Solution: Static Instances are independent of any Clients, they are activated and registered at the Lookup Service typically at Application Startup.

Per-Request Instance
-Context: Remote Objects are stateless.
-Problem: Remote Objects are accessed by a large number of Clients; always creating and destroying Remote Objects is a big overhead -> Bad Performance.
-Solution: The Framework instanciates a Servant for each invocation. The Servant (Remote Object) handles the request and is then deactivated again. Remote Objects are reused.

Client-Dependent Instance
-Context: Client uses Remote Object
-Problem: Remote Object extends Application Logic of the Client.
-Solution: Provide Remote Objects whose Lifetime is controlled by the Client. For each Client there is a private Remote Object, state can only be changed by this specific Client.

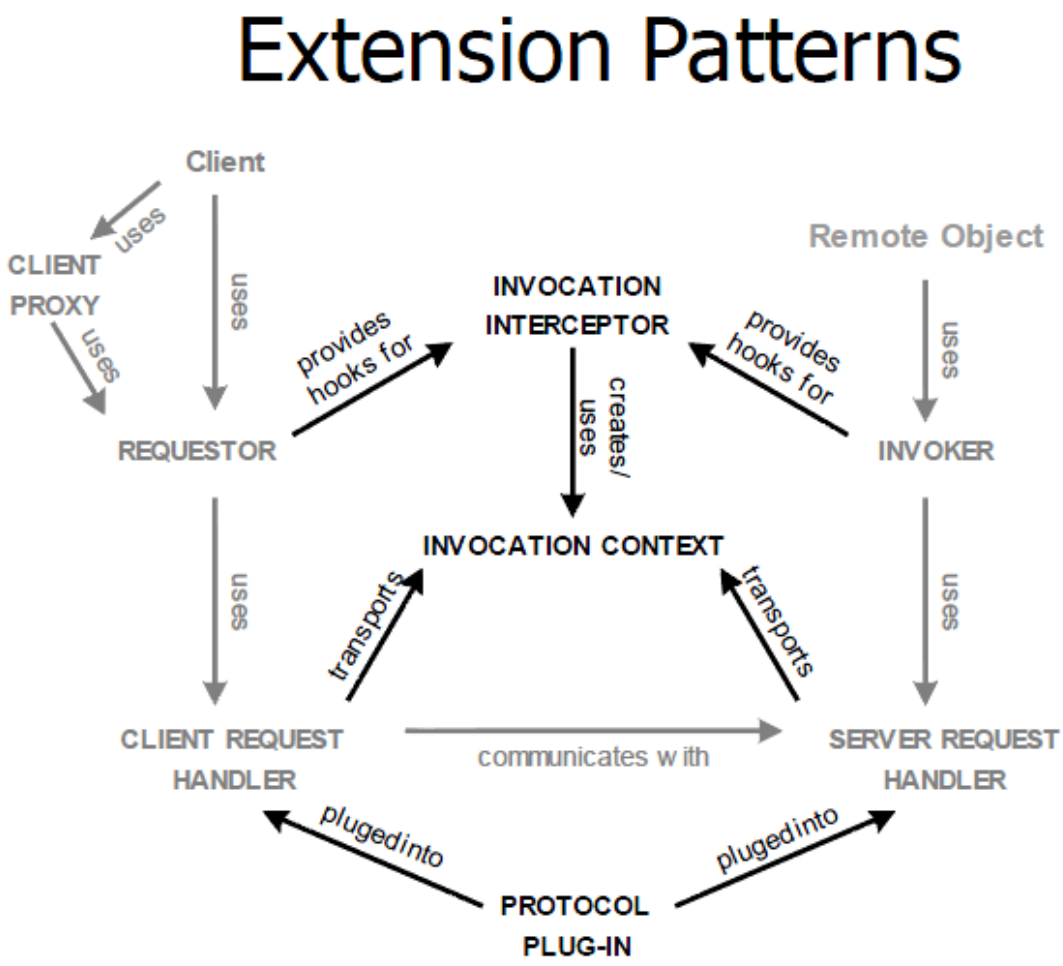
Lazy Acquisition
-Context: Static Instances have to be efficiently managed.
-Problem: Creation of Remote Objects and instanciating at Startup can be a waste of resources and it is slow.
-Solution: Instantiate Servants only when they are invoked by a Client.

Pooling
-Context: Every Remote Object consumes Server Application Resources.
-Problem: Instanciating and Destroying causes a lot of overhead.
-Solution: Introduce a Pool of Servants for Each Remote Object type. Reuse the Servants in the pool.

Leasing
-Context: Clients use Server Application Resources.
-Problem: Resources not longer needed should be released.
-Solution: After a Lease expires, the servant is destroyed by the Lifecycle Manager and is not available to the Client any more. The Client can renew the lease either through invoking an operation again or explicitly through a special command.

Passivation
-Context: Stateful Remote Objects are used.
-Problem: Remote Objects may not be accessed by any Client for a very long time, but they still occupy Server Resources.
-Solution: Passivate not used Remote Objects after a while and only activate them again, if a Client invokes an operation on it.

Extension Patterns



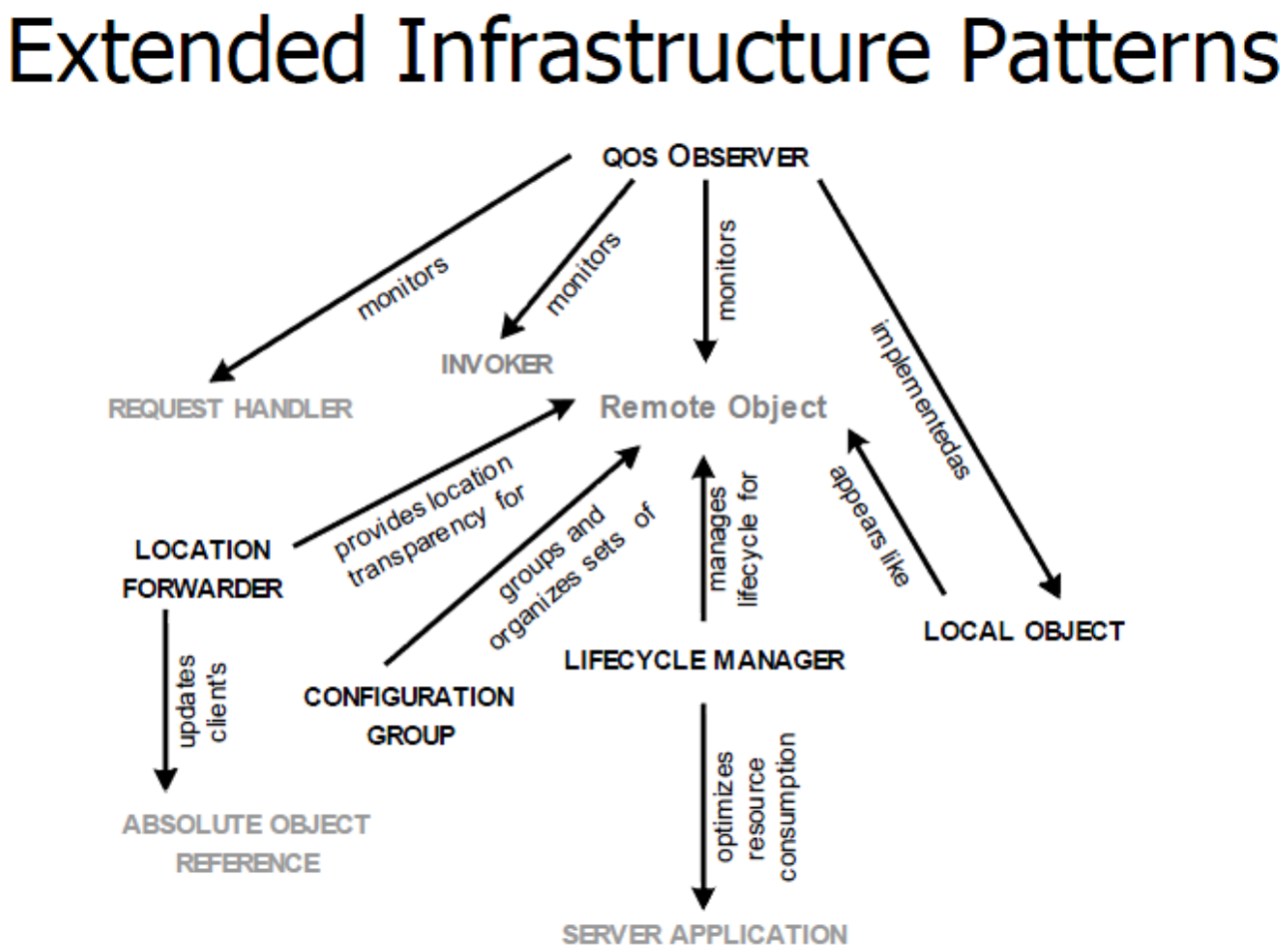
Invocation Interceptor

-Context: Add-Ons need to be integrated transparently.
-Problem: Client and Server have to provide Add-On-Services, e.g. Transaction, Logging, Security, other horizontal Services.
-Solution: Hooks in the invocation path. Before and after each request and response message, all registered Interceptors are invoked and can implement Add-On-Services. This is transparent and is easily and dynamically extendable.

Invocation Context
-Context: Add-On Services are plugged into the framework.
-Problem: Invocation Interceptors often need additional information in order to provide services.
-Solution: Bundle Context-Information in an extensible Invocation Context, that is transferred between Client and Remote Object with every invocation. Invocation Interceptors can add and consume information from the context.

Protocol Plug-In
-Context: Client and Server use the same protocol.
-Problem: The used Communication Protocols should be configurable by Developers.
-Solution: Protocol Plug-Ins extend Client and Server Request Handlers transparently.

Extended Infrastructure Patterns



Lifecycle Manager
-Context: Lifecycles of Remote Objects have to be handled.
-Problem: Based on configuration, usage and available resources servants have to be instantiated and destroyed.
-Solution: A Lifecycle Manager manages and coordinates the Lifecycle of Remote Objects.

Configuration Group
-Context: Remote Objects need similar configurations.
-Problem: Remote Objects have to be configured with various properties, e.g. QoS, Interceptor Tasks, Lifecycle Management and Protocol Support.
-Solution: Provide Configuration Groups which group Remote Objects with common properties.

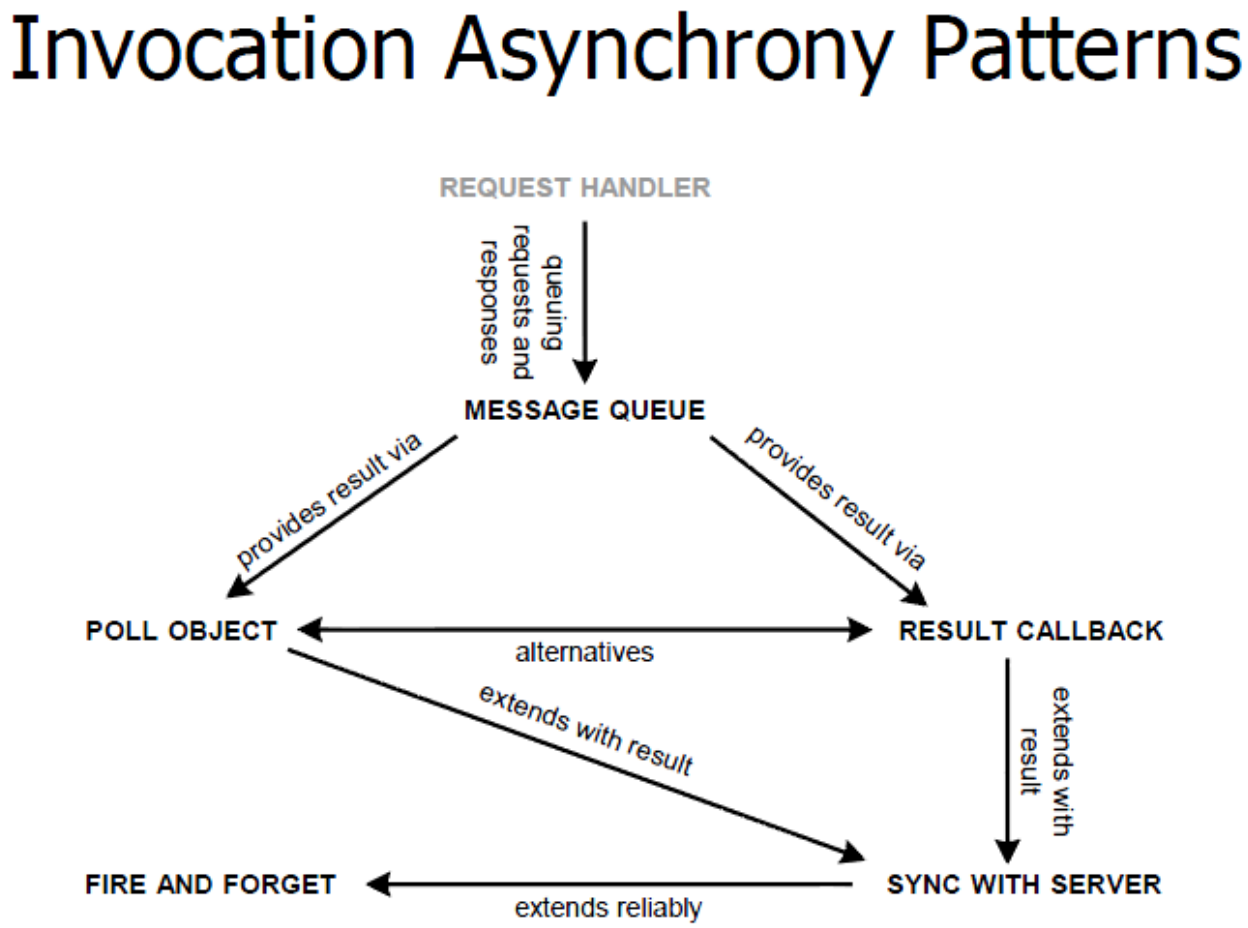
Local Object
-Context: Middleware needs to be configured.
-Problem: To configure Plug-Ins, Configuration Groups and Lifecycle Managers, interfaces are needed. Those Interfaces shouldn't be available remotely, but for consistency reasons should behave similarly to Remote Objects (Parameter Passing, Memory Management)
-Solution: Local Objects. (that behave similarly to Remote Objects)

QoS Observer
-Context: QoS properties have to be controlled.
-Problem: Applications may want to react to QoS changes, but this code should be decoupled from the Middleware Framework.

-Solution: Provide Hooks, for which observers can register. On QoS events those observers can react to the changes.

Location Forwarder
-Context: Invocations for Remote Objects arrive at the Invoker.
-Problem: The Remote Object might be located somewhere else so the Invoker cannot directly invoke it. Invokers should transparently reach Remote Objects.
-Solution: A Location Forwarder forwards invocations to other Servers if necessary.

Invocation Asynchrony Patterns



Fire and Forget
-Context: Operations have neither a return value nor report any exceptions.
-Problem: A client needs to notify a Remote Object of an event and does not expect a return value. The Reliability of the invocation is not critical.
-Solution: Requestor sends invocation, doesn't wait for a result and returns control to the Client immediately.

Sync with Server
-Context: Operations have neither a return value nor report any exceptions.
-Problem: Fire and Forget is too unreliable. A synchronous call should not be used because it blocks the client.
-Solution: The Client sends the invocation, waits for a reply from the server, that it got the invocation command and immediately continues execution. The server then processes the invocation.

Poll Object
-Context: Invocations should execute asynchronously and Clients depend on the result.
-Problem: Client does not need the result of the invocation to continue its execution.
-Solution: Provide a Poll Object, which the Client can use to query if the result is available yet.

Result Callback

-Context: Invocations should execute asynchronously and Clients depend on the results.
-Problem: If the result becomes available, client wants to be informed immediately. Client is able to execute concurrently without the result in the meantime.
-Solution: Provide a callback interface. The Client implements it and passes it to the Middleware. When the result becomes available, the Middleware calls the interfaces Method.

Chapter 4 - Agil-Effektive Software Entwicklung

Essentielle Komplexität vs. Akzidentielle Komplexität
-Essentielle Komplexität: Liegt im Wesen der Sache/Aufgabe - Aufgabe hat diese Komplexität inhärent.
-Akzidentielle Komplexität: Teil der Lösung, nicht notwendige Komplexität, die z.B. durch schlechten Entwurf entsteht.

Wenn Qualität das Ziel ist und folgende Probleme vorhanden sind:
-Komplexität
-Nachhaltigkeit
-Begrenzte Resourcen

Open(X)space:
Scrum
Toolunterstütztes Anforderungsmanagement
Domain Driven Design
Kontinuierliches Integrieren
Testgetriebene Entwicklung

Effektivität durch Automatisierung

Chapter 5 - Software Evolution in Distributed Systems

Software Evolution
is the process of progressive change over time. Continual fixing, adaption and enhancing to maintain stakeholder's satisfaction.

Types of Programs:
-S-Type-Programs (Specifiable): can be stated formally and completely. Those programs do not evolve.
P-Type (Problem-Solving): Imprecise, Real-World problem. This software is likely to evolve.
E-Type (Embedded): A system that becomes part of the world it models. Is inherently evolutionary, because changes in the world and the software affect each other.

Laws of Software Evolution:
1. Law of continuing change
2. Law of increasing entropy
3. Fundamental Law of Software Evolution: SW-Evolution is self-regulating with statistically determinable trends and invariants.
4. Conservation of organisational stability: During active SW-Life the average effective global activity rate is roughly constant.
5. Conservation of familiarity: Incremental growth rate tends to decline.
6. Continuing Growth: Functional content must be continually increasing to maintain user satisfaction.
7. Declining Quality: Quality of E-Type systems will decline unless rigorously maintained and adapted.
8. Feedback System: E-Type evolution must be treated as such to achieve improvement.

Lehman: Formal Model for SW-Evolution

Change Patterns and Evolutionary Narratives:
-Cathedral style: slow evolution, planned, code tested before integrated.
-Bazaar style: lots of low-level changes, frequent fixes, feature-itis.
-Band-aid evolution: Just add a layer; quick and dirty fix.
-Vestigial features: design artifacts persist after rationale for them died long ago.
-Adaptive radiation: when possible, wild variations. later evaluate ideas and let live ony the best ones.
-Convergent evolution: Compare similar systems.

-Radical redesigns: aka refactoring, little new functionality, but structural changes significantly.
-Migration patterns: look out for known translation idioms
-Reuse patterns: components are reused in different systems

Chapter 6 - Case Studies

The Austrian ZMR (Zentrales Melderegister)
Multi-Registerkonzept