Using Design Patterns to Develop Object-Oriented Communication Software Frameworks and Applications

Douglas C. Schmidt

http://www.cs.wustl.edu/~schmidt/

schmidt@cs.wustl.edu

Washington University, St. Louis

Motivation

- Developing *efficient*, *robust*, *extensible*, and *reusable* communication software is hard

- It is essential to understand successful techniques that have proven effective to solve common development challenges

- *Design patterns* and *frameworks* help to capture, articulate, and instantiate these successful techniques

Observations

- Developers of communication software confront recurring challenges that are largely application-independent
  - *e.g.*, service initialization and distribution, error handling, flow control, event demultiplexing, concurrency control

- Successful developers resolve these challenges by applying appropriate *design patterns*

- These patterns have traditionally been either:

  1. *Locked inside the heads of expert software developers*
  2. *Buried within the source code*

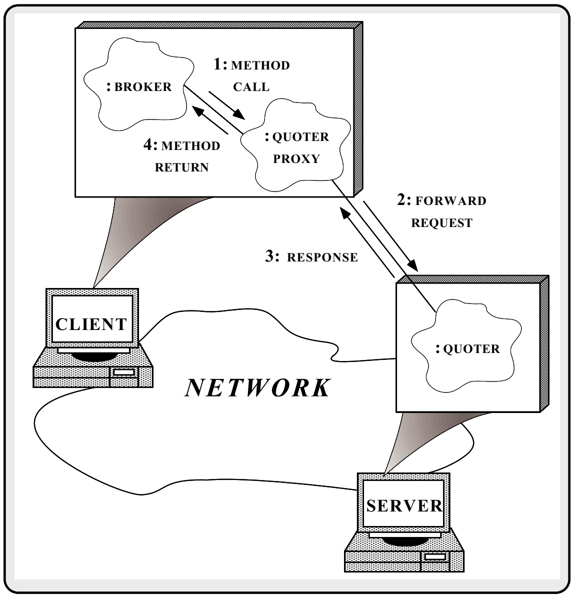
Design Patterns

- Design patterns represent *solutions* to *problems* that arise when developing software within a particular *context*
  - *i.e.*, "Patterns == problem/solution pairs in a context"

- Patterns capture the static and dynamic *structure* and *collaboration* among key *participants* in software designs
  - They are particularly useful for articulating how and why to resolve *non-functional forces*

- Patterns facilitate reuse of successful software architectures and designs

Proxy Pattern



- *Indent*: provide a surrogate for another object that controls access to it

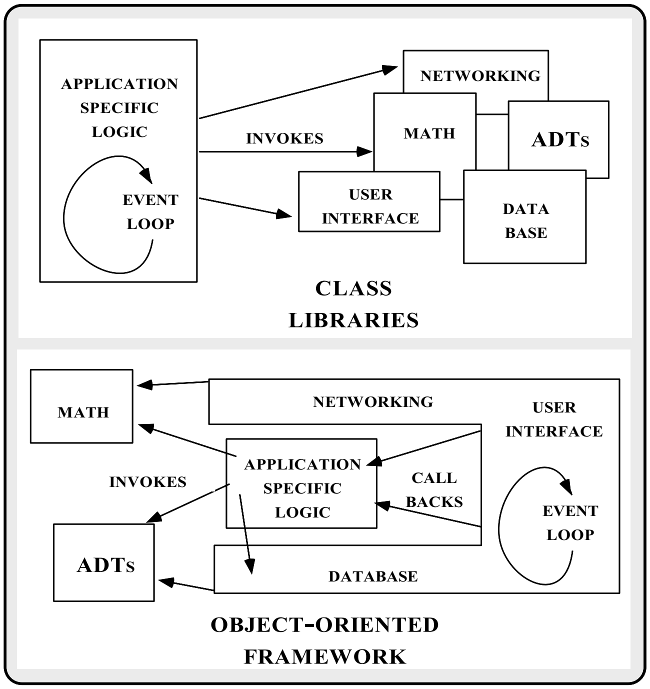
More Observations

- Reuse of patterns alone is not insufficient
  - Patterns enable reuse of architecture and design knowledge, but not code (directly)
- To be productive, developers must also reuse detailed designs, algorithms, interfaces, implementations, etc.
- Application *frameworks* are an effective way to achieve broad reuse of software

Frameworks

- A framework is:
  - "An integrated collection of components that collaborate to produce a reusable architecture for a family of related applications"
- Frameworks differ from conventional class libraries:
  1. Frameworks are "semi-complete" applications
  2. Frameworks address a particular application domain
  3. Frameworks provide "inversion of control"
- Typically, applications are developed by *inheriting* from and *instantiating* framework components

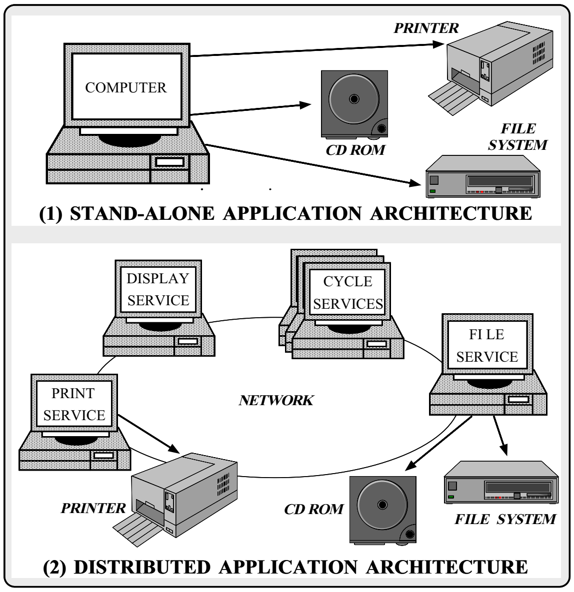
Differences Between Class Libraries and Frameworks



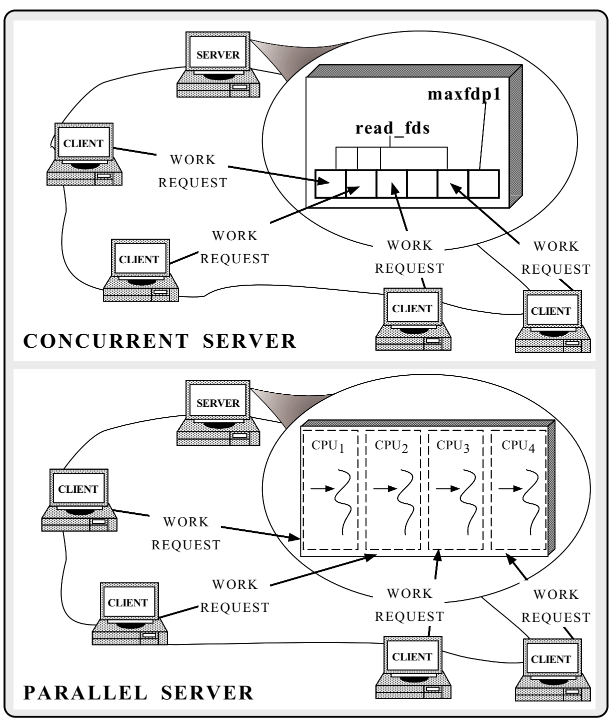
Tutorial Outline

- Outline key challenges for developing communication software
- Present the key reusable design patterns in an application-level Gateway
  - Both single-threaded and multi-threaded solutions are presented
- Discuss lessons learned from using patterns on production software systems

Stand-alone vs. Distributed Application Architectures



Concurrency vs. Parallelism



Sources of Complexity

- Distributed application development exhibits both *inherent* and *accidental* complexity
- *Inherent complexity* results from fundamental challenges, *e.g.*,
  - Distributed systems
    - ▹ *Latency*
    - ▹ *Error handling*
    - ▹ *Service partitioning and load balancing*
  - Concurrent systems
    - ▹ *Race conditions*
    - ▹ *Deadlock avoidance*
    - ▹ *Fair scheduling*
    - ▹ *Performance optimization and tuning*

Sources of Complexity (cont'd)

- *Accidental complexity* results from limitations with tools and techniques, *e.g.*,
  - Lack of type-secure, portable, re-entrant, and extensible system call interfaces and component libraries
  - Inadequate debugging support
  - Widespread use of *algorithmic* decomposition
    - ▷ Fine for *explaining* network programming concepts and algorithms but inadequate for *developing* large-scale distributed applications

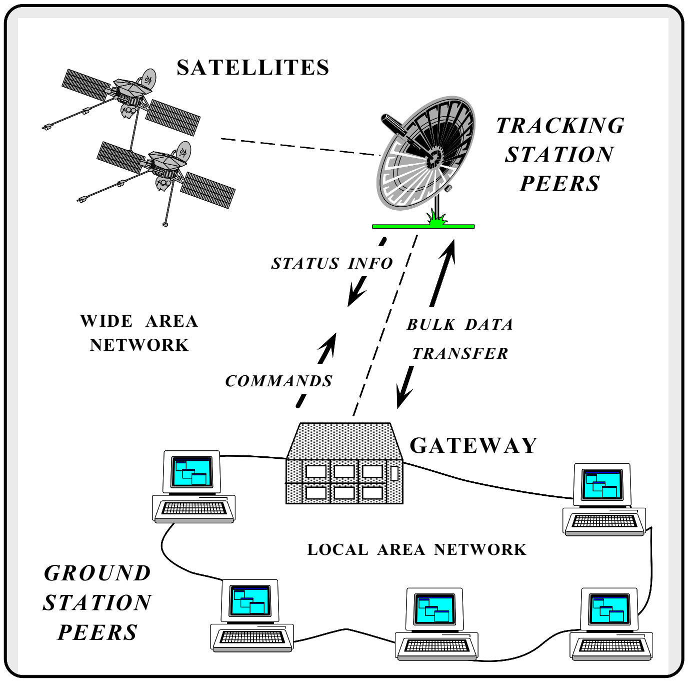
OO Contributions

- Concurrent and distributed programming has traditionally been performed using low-level OS mechanisms, *e.g.*,
  - *fork/exec*
  - *Shared memory*
  - *Signals*
  - *Sockets and select*
  - *POSIX pthreads, Solaris threads, Win32 threads*
- OO *design patterns* and *frameworks* elevate development to focus on application concerns, *e.g.*,
  - *Service functionality and policies*
  - *Service configuration*
  - *Concurrent event demultiplexing and event handler dispatching*
  - *Service concurrency and synchronization*

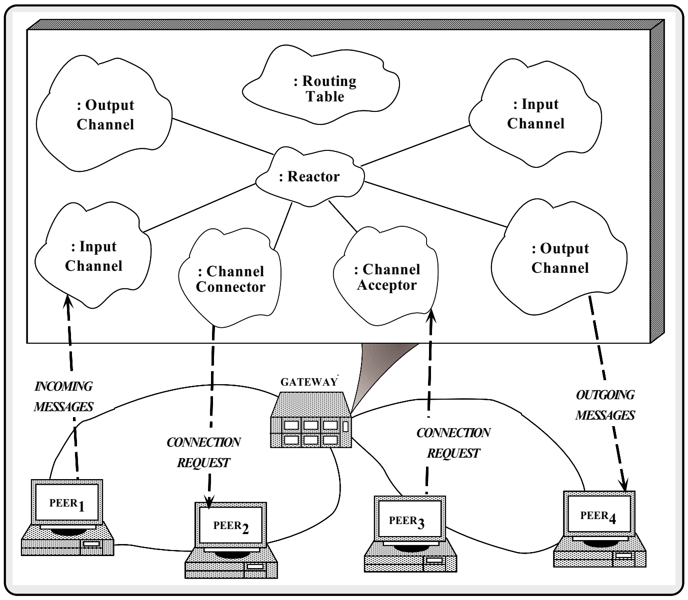
Application-level Gateway Example

- This example illustrates the reusable *design patterns* and *framework* components used in an OO architecture for *application-level Gateways*
- Gateways route messages between Peers in a distributed system
- Peers and Gateways communicate via a connection-oriented transport protocol
  - *e.g.*, TCP/IP, IPX/SPX, TP4

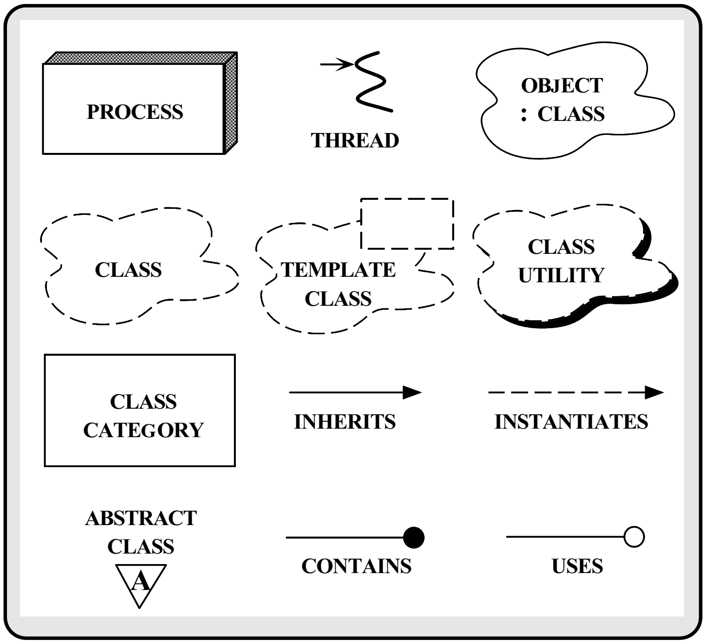
Physical Architecture of the Gateway



OO Software Architecture of the Gateway



Graphical Notation



Gateway Behavior

- Components in the Gateway behave as follows:

1. `Gateway` parses configuration files that specify which Peers to connect with and which routes to use

2. `Channel_Connector` connects to Peers, then creates and activates the appropriate `Channel` subclasses (`Input_Channel` or `Output_Channel`)

3. Once connected, Peers send messages to the Gateway

   – Messages are handled by the appropriate `Input_Channel`

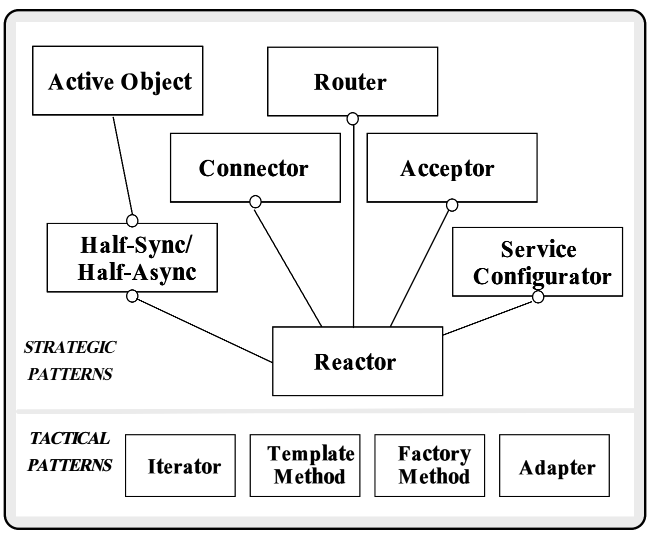
   – `Input_Channels` work as follows:

   (a) Receive and validate messages

   (b) Consult a `Routing_Table`

   (c) Forward messages to the appropriate Peer(s) via `Output_Channels`

Design Patterns in the Gateway



- The Gateway components are based upon a family of design patterns

Tactical Patterns

- Iterator
  - "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation"
- Factory Method
  - "Define an interface for creating an object, but let subclasses decide which class to instantiate"
    - ▷ Factory Method lets a class defer instantiation to subclasses
- Adapter
  - "Convert the interface of a class into another interface client expects"
    - ▷ Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

Concurrency Patterns

- *Reactor*
  - "Decouples event demultiplexing and event handler dispatching from application services performed in response to events"
- *Active Object*
  - "Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"
- *Half-Sync/Half-Async*
  - "Decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency"

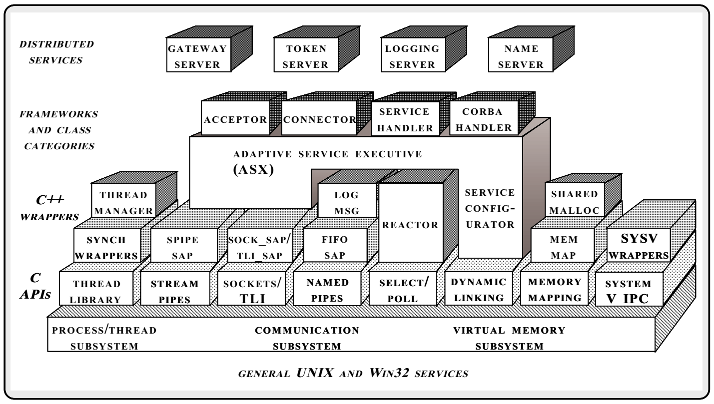
Service Initialization Patterns

- *Connector*
  - "Decouples active connection establishment from the service performed once the connection is established"
- *Acceptor*
  - "Decouples passive connection establishment from the service performed once the connection is established"
- *Service Configurator*
  - "Decouples the behavior of network services from point in time at which services are configured into an application"

Application-Specific Patterns

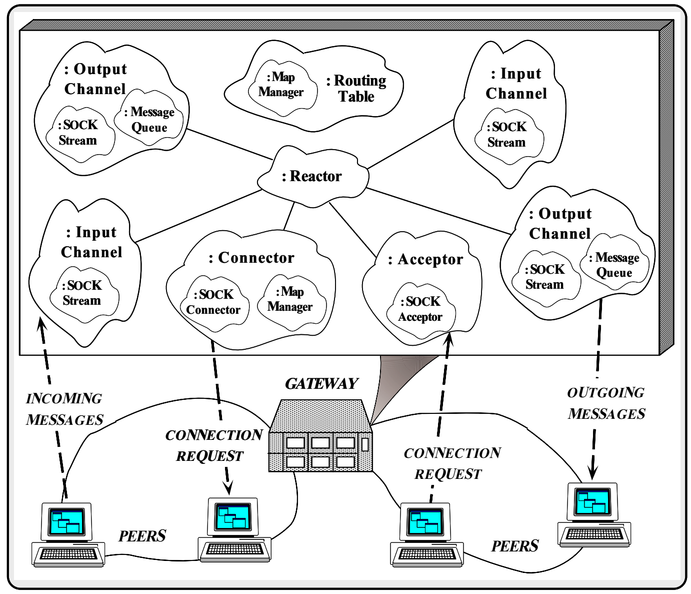
- *Router*
  - "Decouples multiple sources of input from multiple sources of output to route data correctly without blocking"

The ADAPTIVE Communication Environment (ACE)



- A set of C++ wrappers and frameworks based on common design patterns

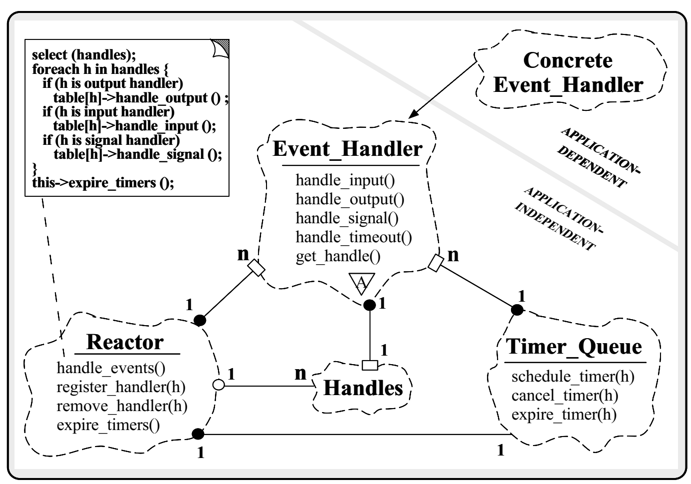
ACE Components in the Gateway



The Reactor Pattern

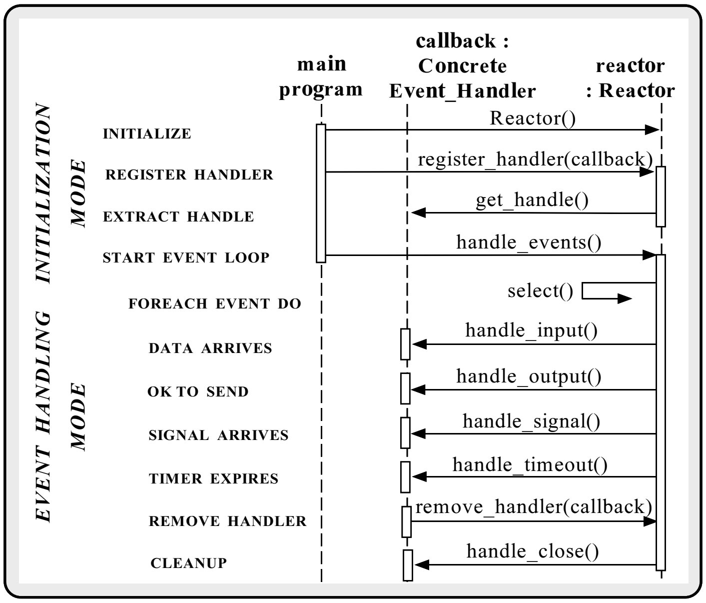
- *Intent*
  - "Decouples event demultiplexing and event handler dispatching from the services performed in response to events"
- This pattern resolves the following forces for event-driven software:
  - *How to demultiplex multiple types of events from multiple sources of events efficiently within a single thread of control*
  - *How to extend application behavior without requiring changes to the event dispatching framework*

Structure of the Reactor Pattern

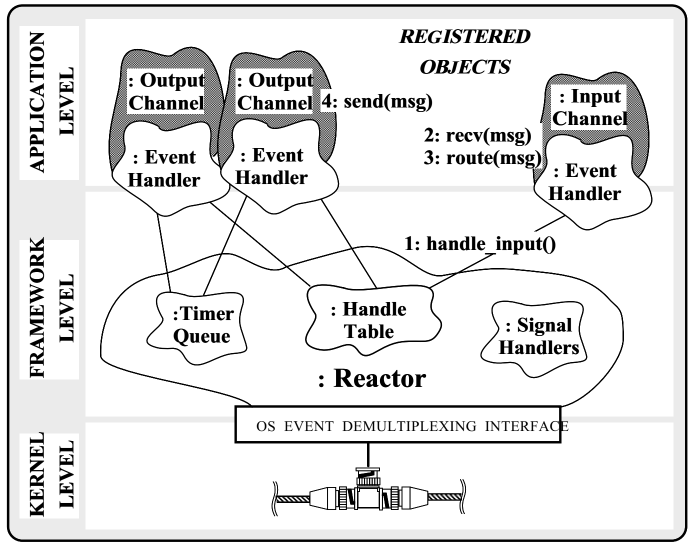


- Participants in the Reactor pattern

Collaboration in the Reactor Pattern



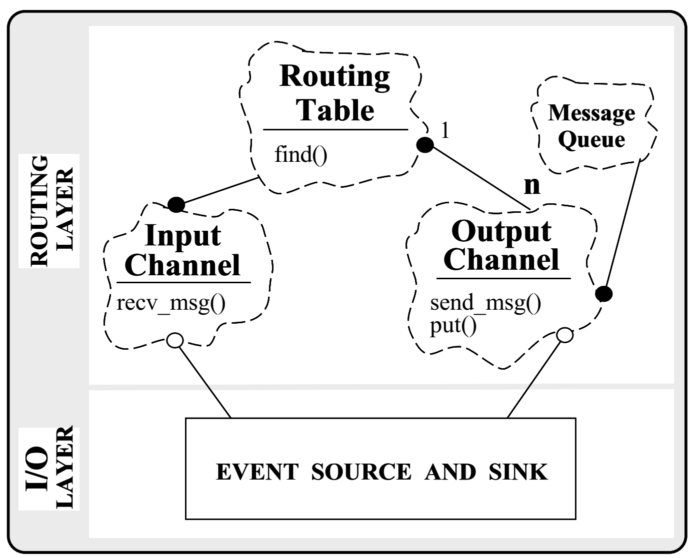
Using the Reactor for the Gateway



The Router Pattern

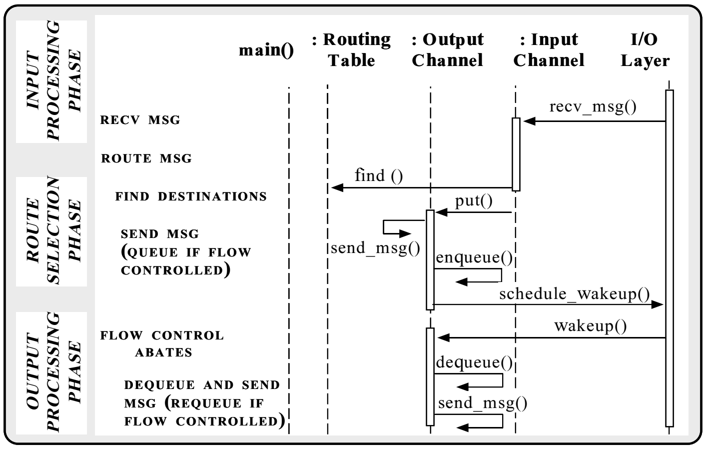
- *Intent*
  - "Decouple multiple sources of input from multiple sources of output to route data correctly without blocking"
- The Router pattern resolves the following forces for connection-oriented routers:
  - *How to prevent misbehaving connections from disrupting the quality of service for well-behaved connections*
  - *How to allow different concurrency strategies for Input and Output Channels*

Structure of the Router Pattern

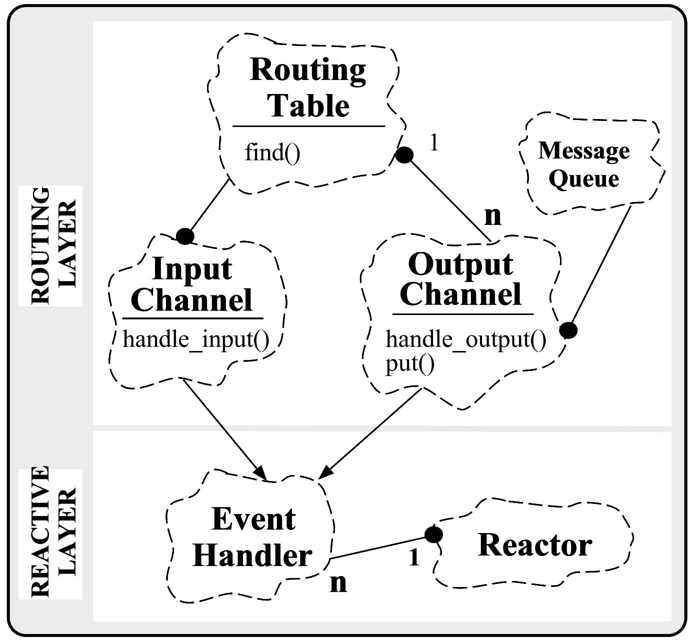


- Participants in the Router pattern

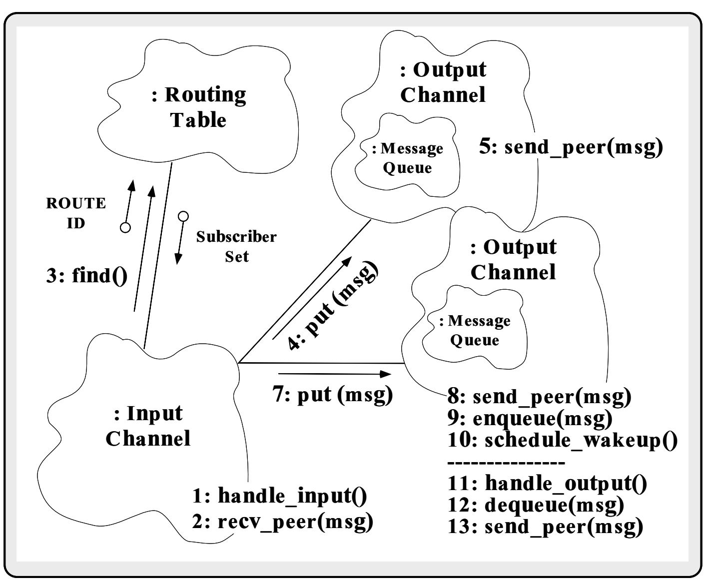
Collaboration in the Router Pattern



Structure of the Single-Threaded Router Pattern



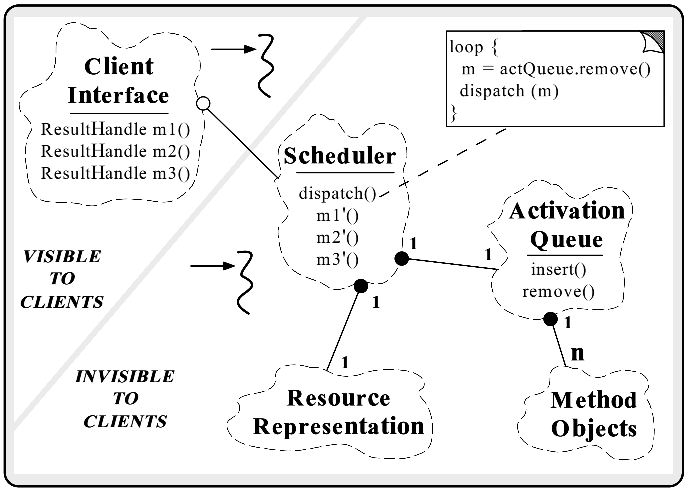
Collaboration in Single-threaded Gateway Routing



The Active Object Pattern

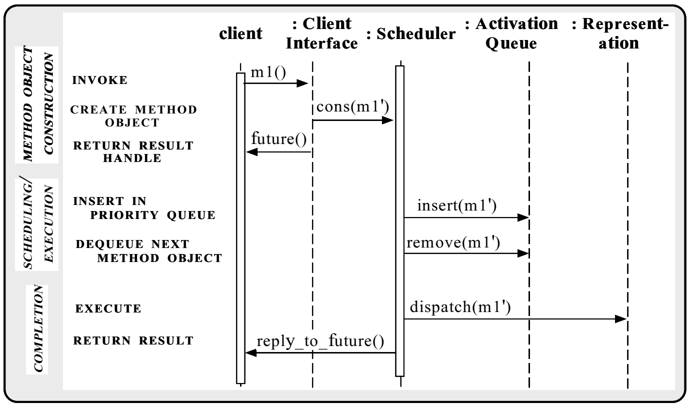
- *Intent*
  - "Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"
- This pattern resolves the following forces for concurrent communication software:
  - *How to allow blocking read and write operations on one endpoint that do not detract from the quality of service of other endpoints*
  - *How to simplify concurrent access to shared state*

Structure of the Active Object Pattern

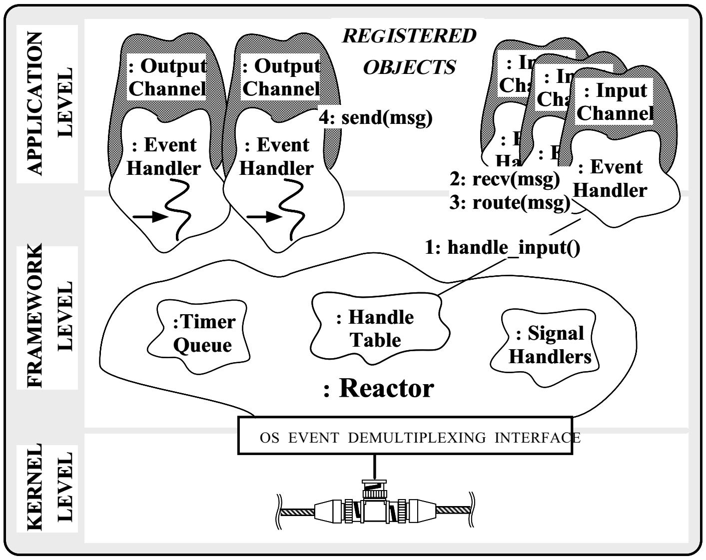


- The `Scheduler` is a "meta-object" that determines the sequence `Method Objects` are executed

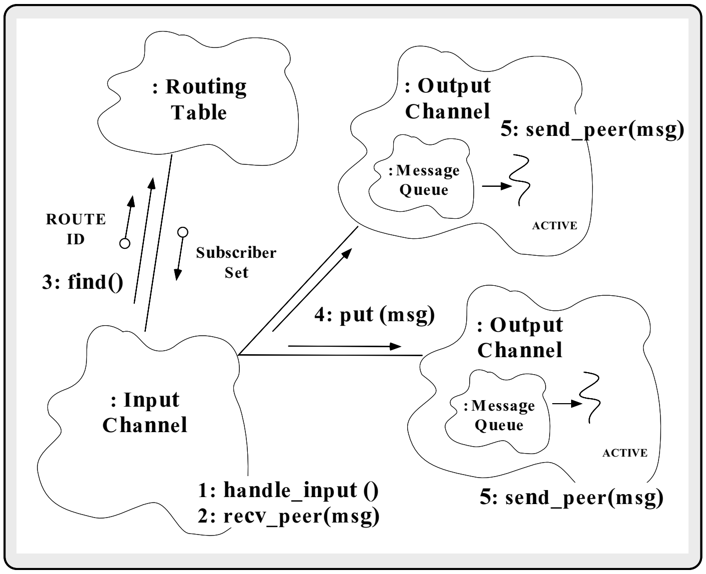
Collaboration in the Active Object Pattern



Using the Active Object Pattern for the Gateway



Collaboration in the Active Object-based Gateway Routing

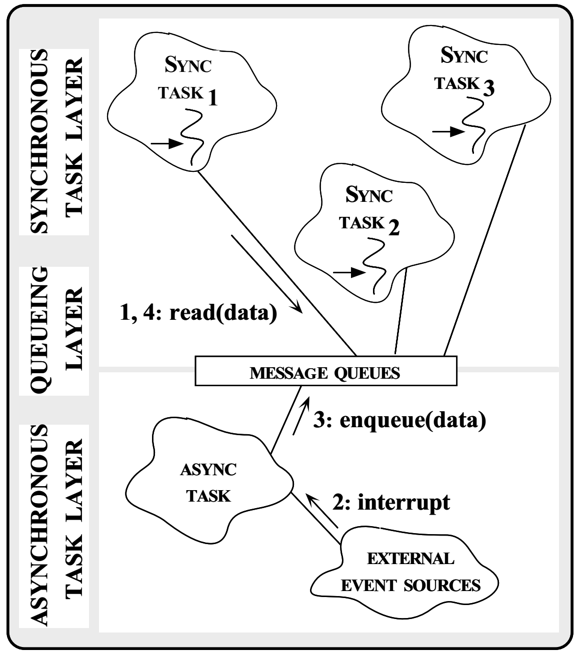


Half-Sync/Half-Async Pattern

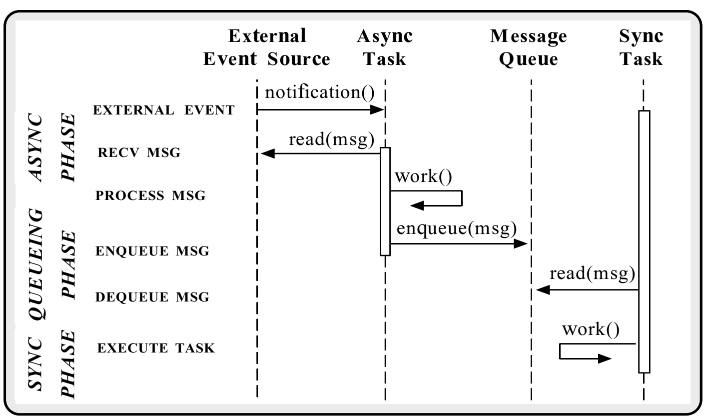
- *Intent*
  - "Decouples synchronous I/O from asynchronous I/O in a system to simplify programming effort without degrading execution efficiency"

- This pattern resolves the following forces for concurrent communication systems:
  - *How to simplify programming for higher-level communication tasks*
    - ▷ These are performed synchronously
  - *How to ensure efficient lower-level I/O communication tasks*
    - ▷ These are performed asynchronously

Structure of the Half-Sync/Half-Async Pattern

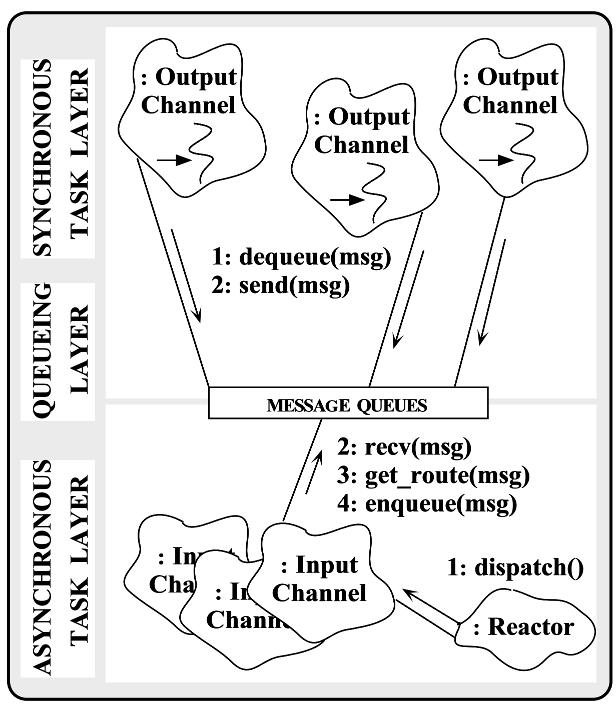


Collaborations in the Half-Sync/Half-Async Pattern



- This illustrates *input* processing (*output* processing is similar)

Using the Half-Sync/Half-Async Pattern in the Gateway

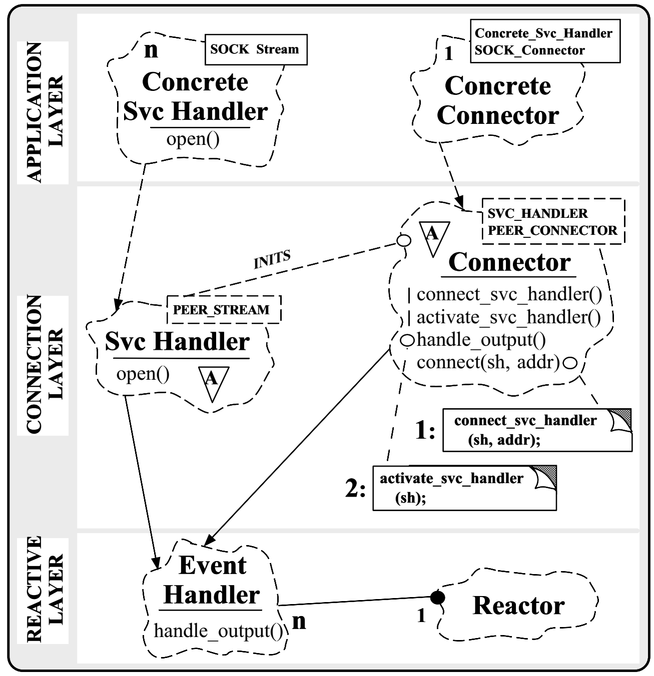


The Connector Pattern

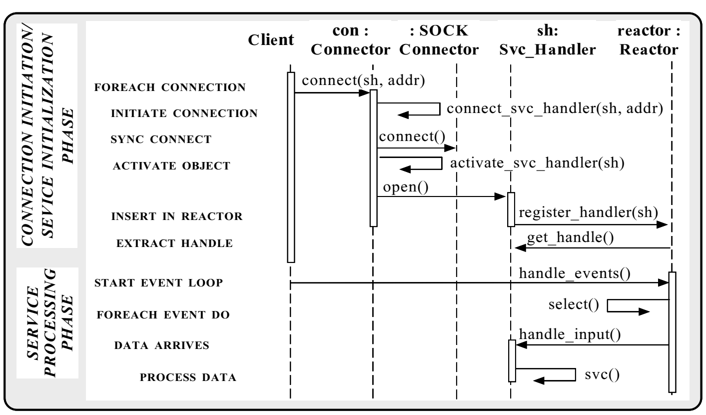
- *Intent*
  - "Decouples active initialization of a service from the task performed once a service is initialized"

- This pattern resolves the following forces for network clients that use interfaces like sockets or TLI:
  1. *How to reuse active connection establishment code for each new service*
  2. *How to make the connection establishment code portable across platforms that may contain sockets but not TLI, or vice versa*
  3. *How to enable flexible service concurrency policies*
  4. *How to actively establish connections with large number of peers efficiently*

Structure of the Connector Pattern

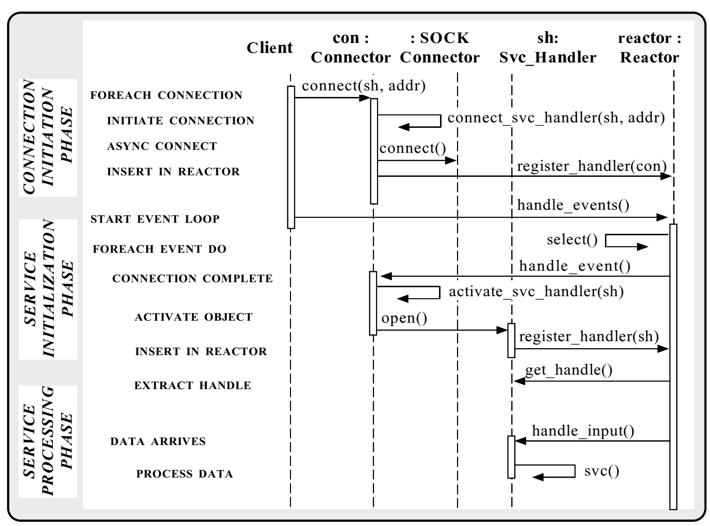


Collaboration in the Connector Pattern



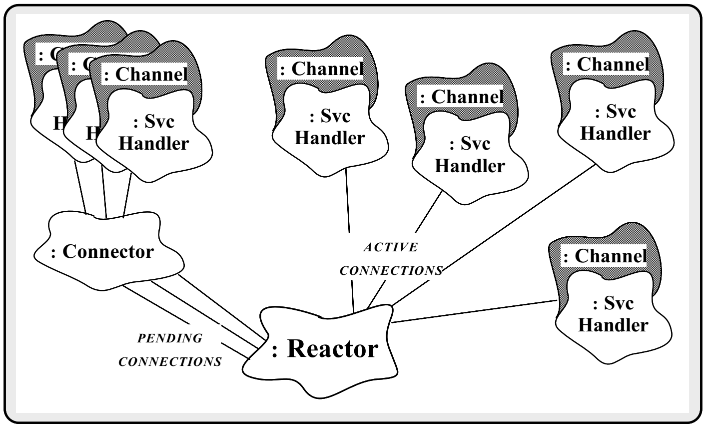
- Synchronous mode

Collaboration in the Connector Pattern



- Asynchronous mode

Using the Connector for the Gateway



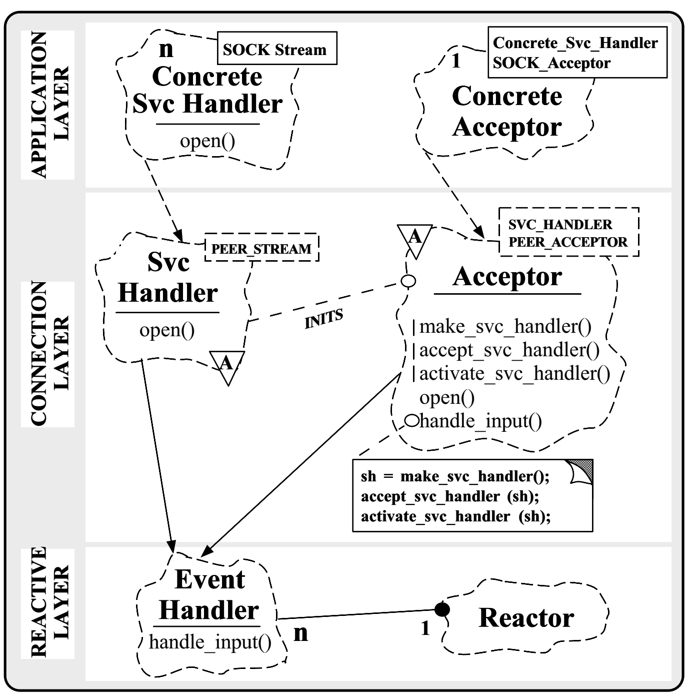
The Acceptor Pattern

- *Intent*
  - "Decouples passive initialization of a service from the tasks performed once the service is initialized"

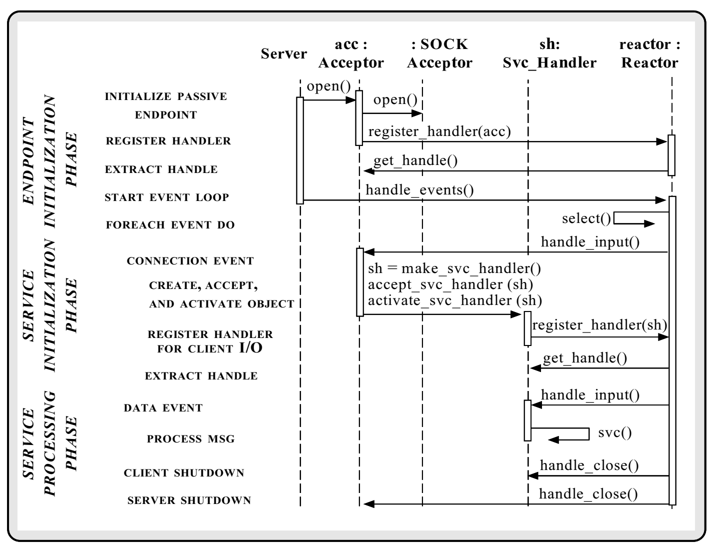
- This pattern resolves the following forces for network servers using interfaces like sockets or TLI:

  1. *How to reuse passive connection establishment code for each new service*
  2. *How to make the connection establishment code portable across platforms that may contain sockets but not TLI, or vice versa*
  3. *How to ensure that a passive-mode descriptor is not accidentally used to read or write data*
  4. *How to enable flexible policies for creation, connection establishent, and concurrency*

Structure of the Acceptor Pattern

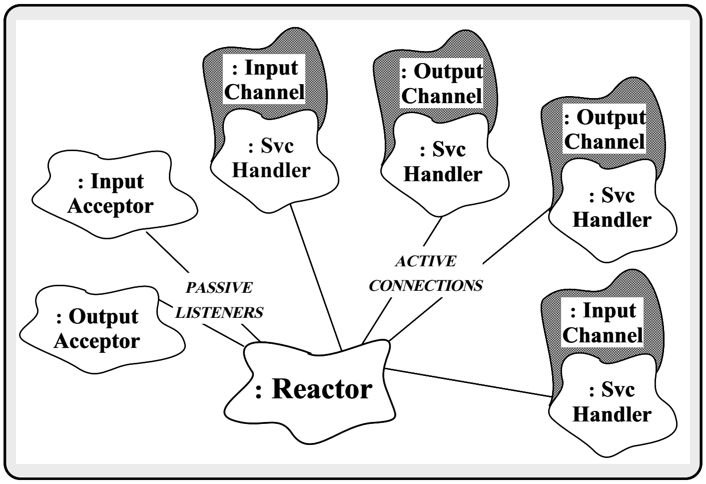


Collaboration in the Acceptor Pattern



- `Acceptor` is a factory that creates, connects, and activates a `Svc_Handler`

Using the Acceptor Pattern in the Gateway

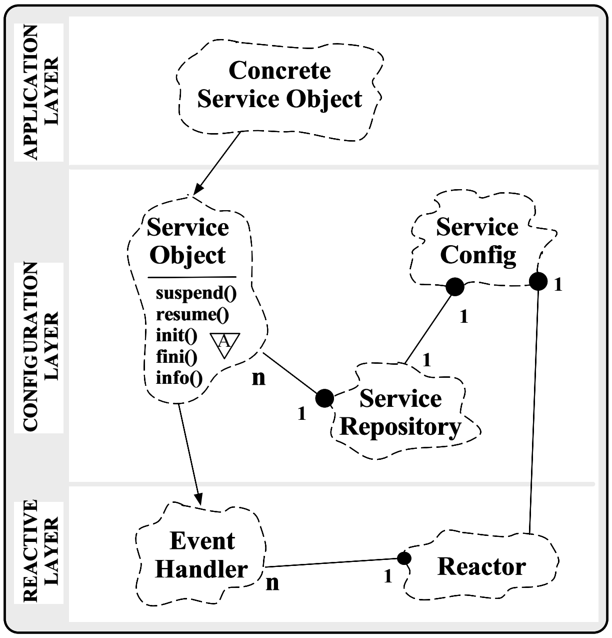


The Service Configurator Pattern

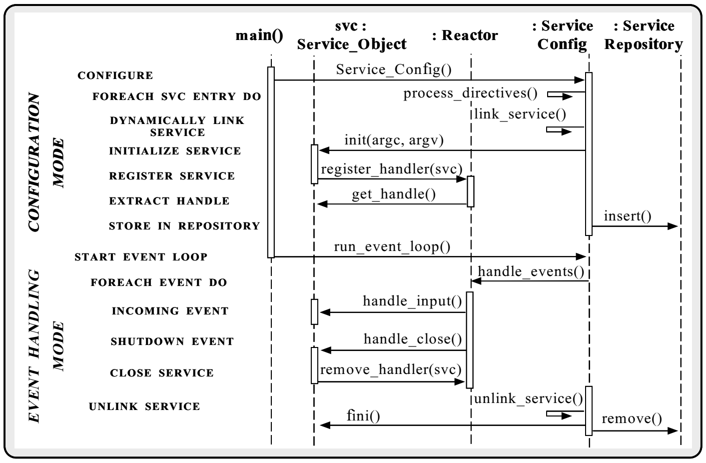
- *Intent*
  - "Decouples the behavior of network services from the point in time at which these services are configured into an application"

- This pattern resolves the following forces for highly flexible communication software:
  - *How to defer the selection of a particular type, or a particular implementation, of a service until very late in the design cycle*
    - ▷ *i.e.*, at installation-time or run-time
  - *How to build complete applications by composing multiple independently developed services*
  - *How to reconfigure and control the behavior of the service at run-time*

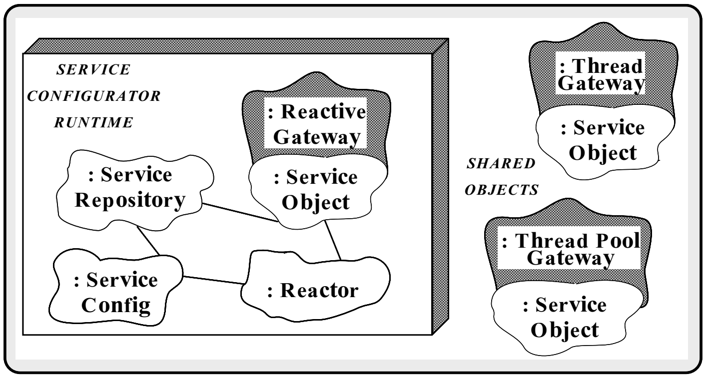
Structure of the Service Configurator Pattern



Collaboration in the Service Configurator Pattern



Using the Service Configurator Pattern for the Gateway



- Replace the single-threaded Gateway with a multi-threaded Gateway

Benefits of Design Patterns

- *Design patterns enable large-scale reuse of software architectures*
- *Patterns explicitly capture expert knowledge and design tradeoffs*
- *Patterns help improve developer communication*
- *Patterns help ease the transition to object-oriented technology*

Drawbacks to Design Patterns

- *Patterns do not lead to direct code reuse*
- *Patterns are deceptively simple*
- *Teams may suffer from pattern overload*
- *Patterns are validated by experience rather than by testing*
- *Integrating patterns into a software development process is a human-intensive activity*

Suggestions for Using Patterns Effectively

- *Do not recast everything as a pattern*
  - Instead, develop strategic domain patterns and reuse existing tactical patterns
- *Institutionalize rewards for developing patterns*
- *Directly involve pattern authors with application developers and domain experts*
- *Clearly document when patterns apply and do not apply*
- *Manage expectations carefully*

Books and Magazines on Patterns

- *Books*
  - Gamma et al., "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, Reading, MA, 1994.
  - "Pattern Languages of Program Design," editors James O. Coplien and Douglas C. Schmidt, Addison-Wesley, Reading, MA, 1995
- *Special Issues in Journals*
  - "Theory and Practice of Object Systems" (guest editor: Stephen P. Berczuk)
  - "Communications of the ACM" (guest editors: Douglas C. Schmidt, Ralph Johnson, and Mohamed Fayad)
- *Magazines*
  - C++ Report and Journal of Object-Oriented Programming, columns by Coplien, Vlissides, and De Souza

Conferences and Workshops on Patterns

- Joint *Pattern Languages of Programs* Conferences
  - 3rd PLoP
    - ▷ September 4–6, 1996, Monticello, Illinois, USA
  - 1st EuroPLoP
    - ▷ July 10–14, 1996, Kloster Irsee, Germany
  - http://www.cs.wustl.edu/~schmidt/jointPLoP–96.html/
- USENIX COOTS
  - June 17–21, 1996, Toronto, Canada
  - http://www.cs.wustl.edu/~schmidt/COOTS–96.html/

Obtaining ACE

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns
- All source code for ACE is freely available
  - Anonymously ftp to `wuarchive.wustl.edu`
  - Transfer the files `/languages/c++/ACE/*.gz` and `gnu/ACE-documentation/*.gz`
- Mailing lists

  * ace-users@cs.wustl.edu
  * ace-users-request@cs.wustl.edu
  * ace-announce@cs.wustl.edu
  * ace-announce-request@cs.wustl.edu
- WWW URL
  - http://www.cs.wustl.edu/~schmidt/