Real-Time Design Patterns

Bruce Powel Douglass, PhD

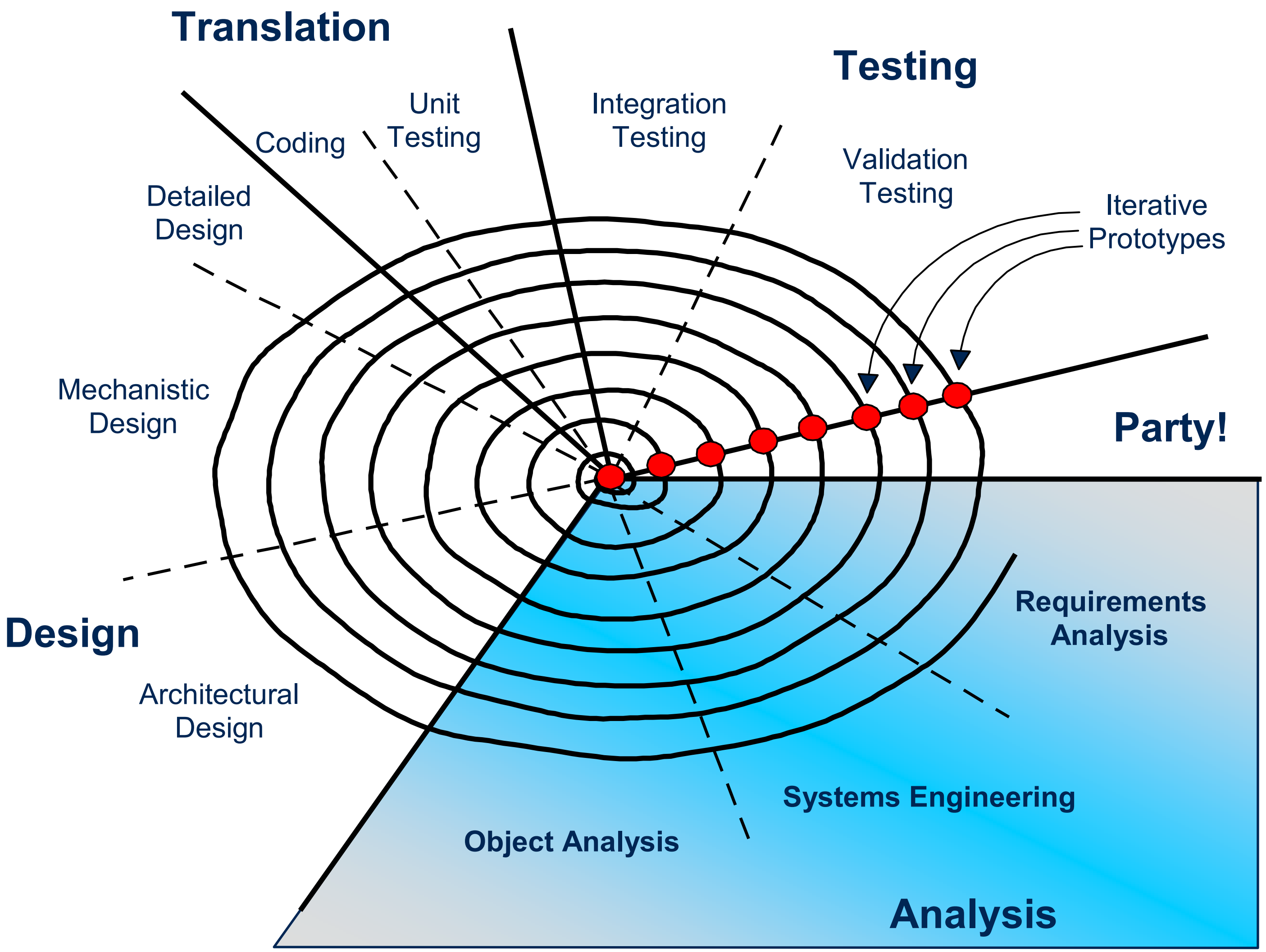
Telelogic, Inc.

www.ilogix.com

Agenda

- Design Process
  - Where does architecture fit in?
- What is a design pattern?
- Architectural design patterns
  - Execution Control
  - Structure
  - Resource Management
  - Safety and Reliability

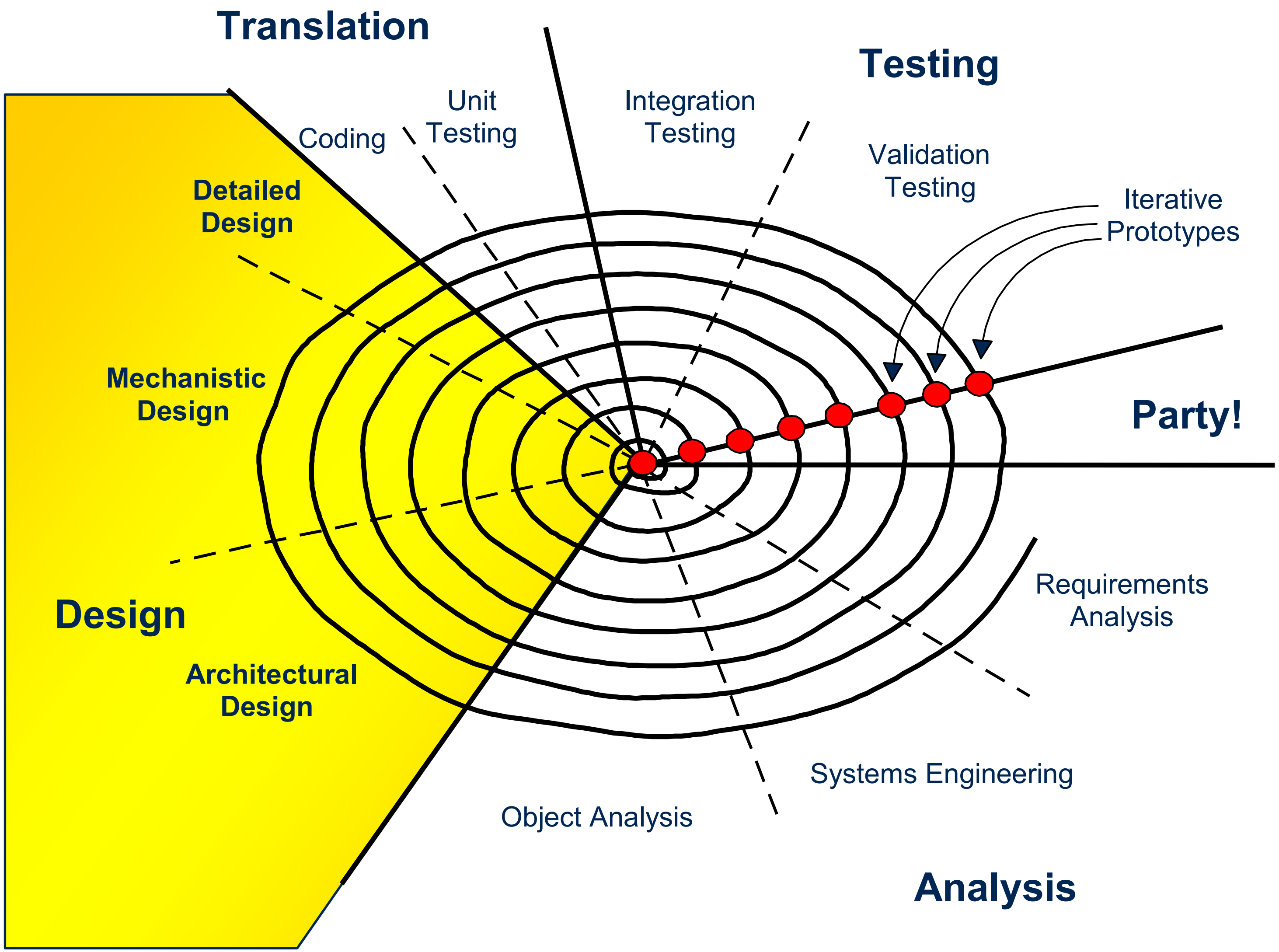
Analysis in ROPES



Analysis

Analysis is the *identification of the required properties of all possible acceptable solutions*

Design in ROPES



Design

Design is the selection of one particular solution which optimizes the set of design criteria with respect to the relative importance of each

Common Design Criteria

- Performance
  - Average
  - Worst-case
  - Predictability
- Resource usage
  - Robustness
  - Thread safety
  - Minimization of resources (space)
  - Minimization of resources (time)
- Safety
- Reliability
- Reusability
- Extensibility & evolvability
- Maintainability
- Time-to-market
- Standard conformance

Quality of Service (QoS) always drives your design

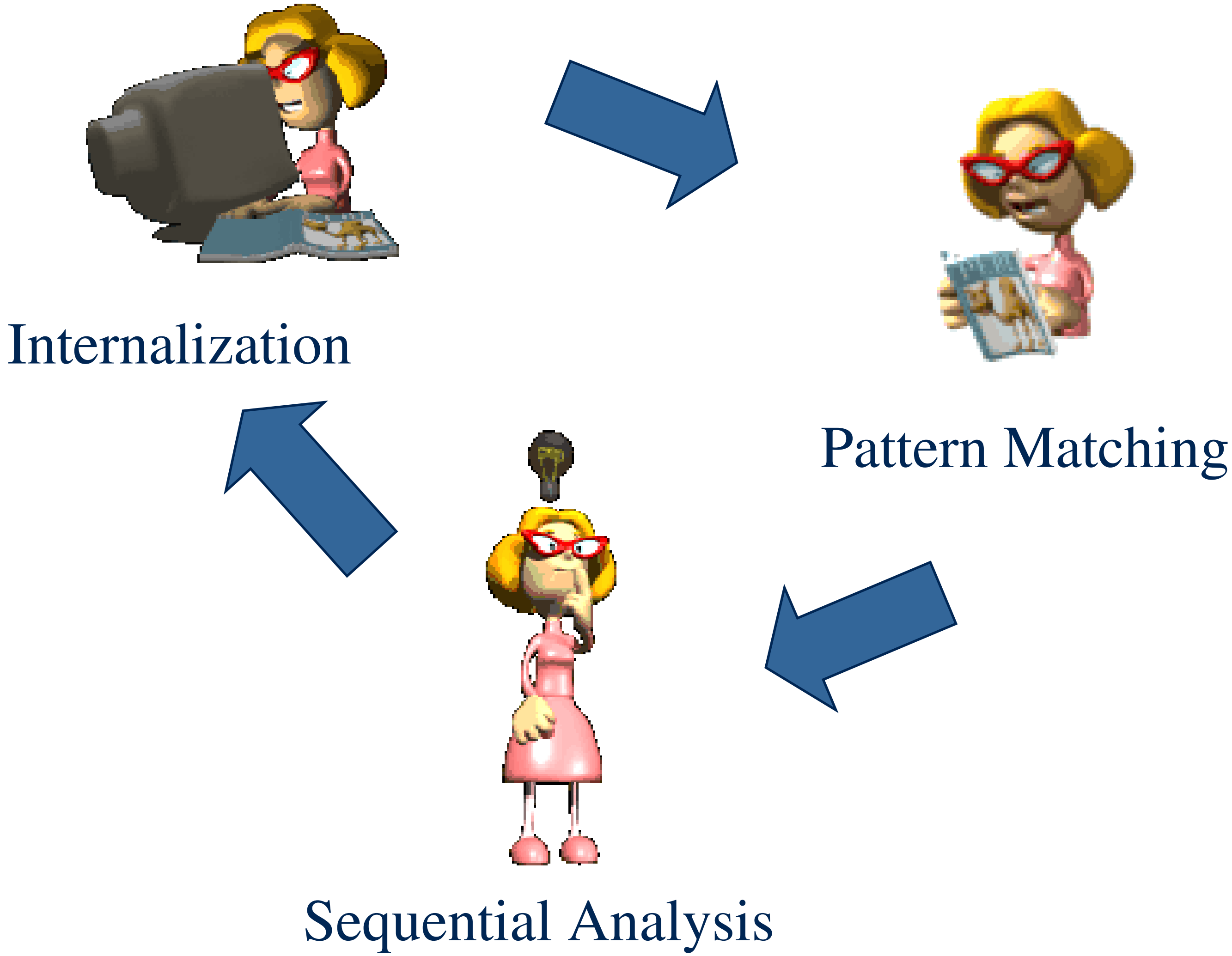
Analysis

- ***What***, not ***How***
- *Identify all properties that must exist in all acceptable solutions*
- Requirements Analysis
  - Identify black box functionality
  - Use case and context views
- Object Analysis
  - Identify essential object model
  - Class, state and scenario views

Design

- *How* analysis model will be implemented
- *Identify and refine the properties of one particular solution*
- Two approaches
  - *Elaborative Design*
    Refine analysis model by adding more detail
  - *Translative Design*
    Build a translator that embodies design decisions

Creative Design



Telelogic

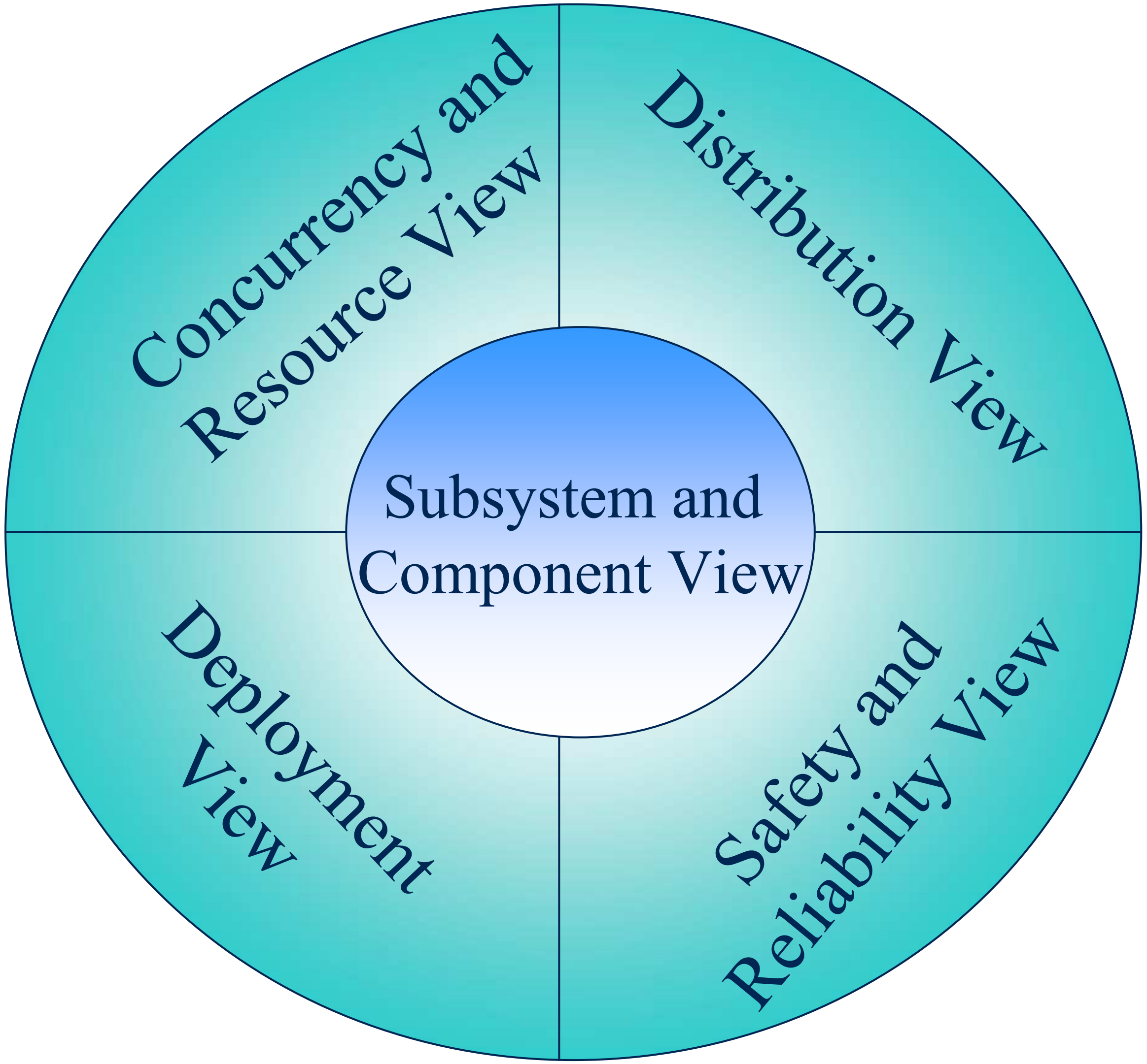
(Physical) Architectural Design

- (Harmony) Architectural Design consists of 5 interrelated model views:
  - Concurrency and Resource View
  - Deployment View
  - Distribution View
  - Safety and Reliability View
  - Subsystem and Component View

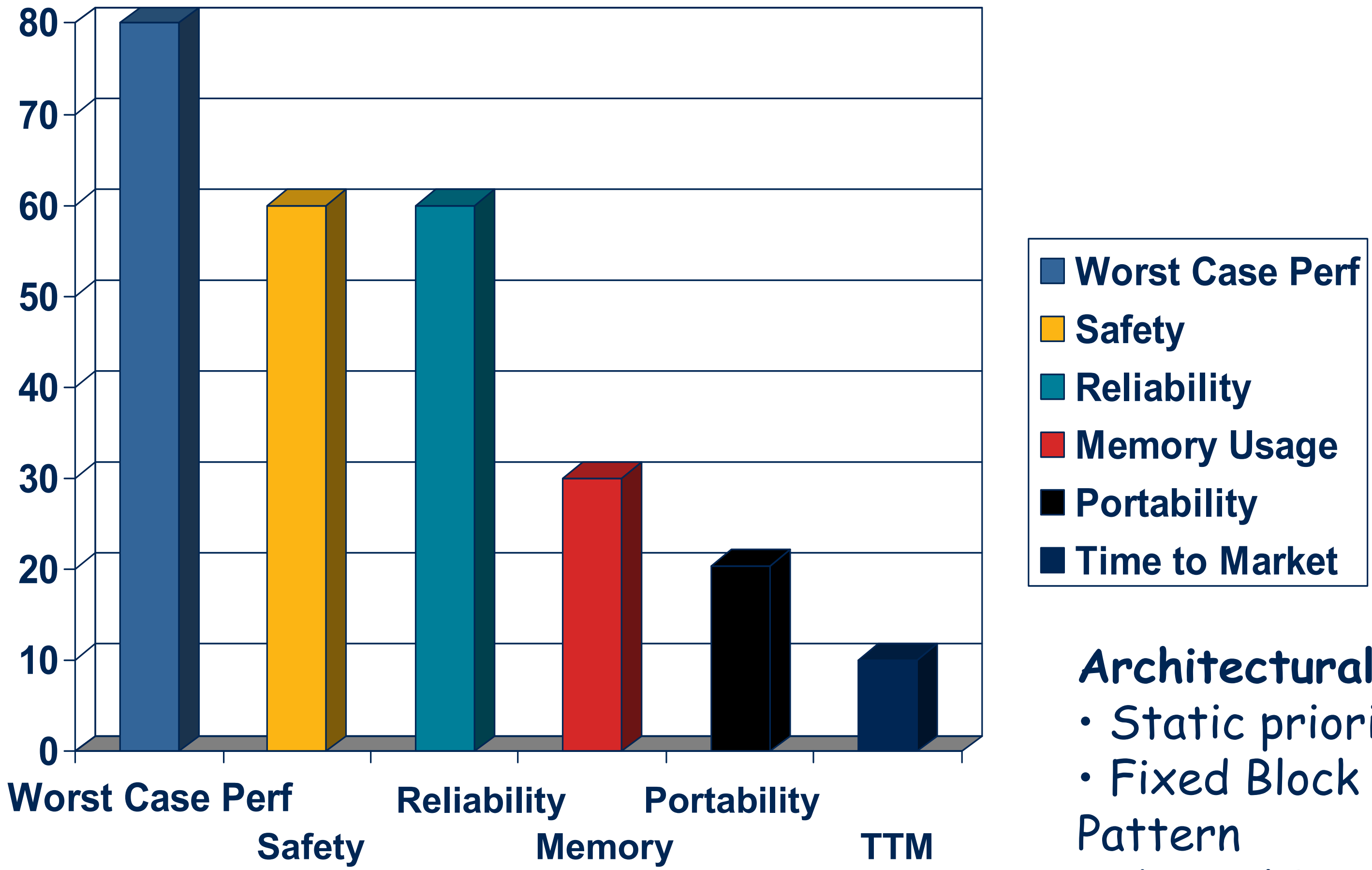
Each Architectural View will have its own design patterns. The complete system architecture is the set of design patterns used in of the aspects of physical architecture.

Telelogic

Aspects of Architecture



Example of Pattern Selection

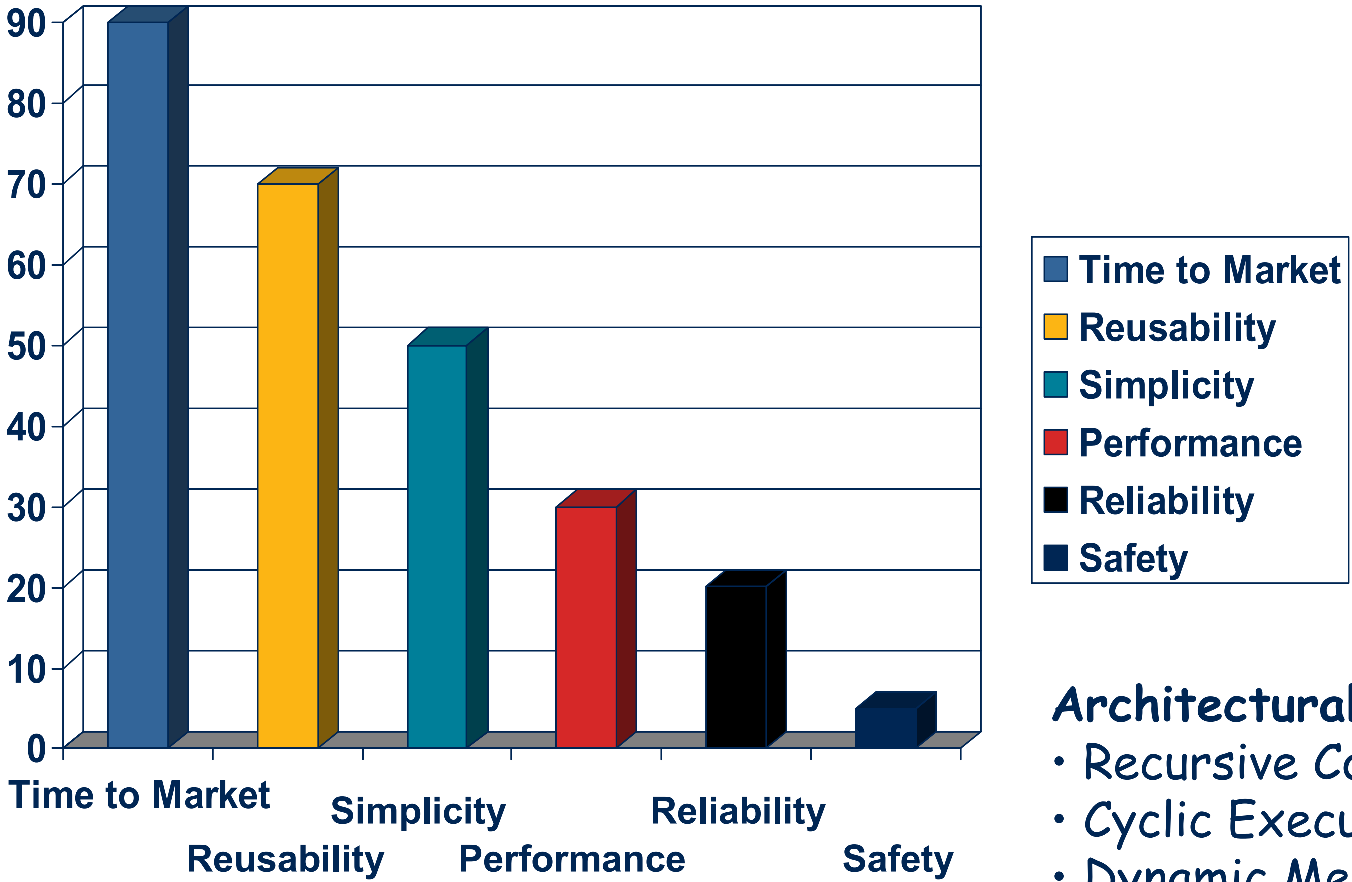


Design Criteria ranked by criticality

Architectural Patterns:

- Static priority Pattern
- Fixed Block Memory Allocation Pattern
- Channel Pattern
- Triple Modular Redundancy Pattern

Example of Pattern Selection (2)



Design Criteria ranked by criticality

Architectural Patterns:

- Recursive Containment Pattern
- Cyclic Executive Pattern
- Dynamic Memory Pattern
- Container-Iterator Pattern

Mechanistic Design

- Addition and refinement of objects to implement analysis models
- Addition of “glue” objects
  - Containers and Collections
  - Interfaces
  - Medium-level policies
  - Optimized rendezvous among threads
  - Aid reuse by enforcing
    - Good abstraction
    - Good encapsulation

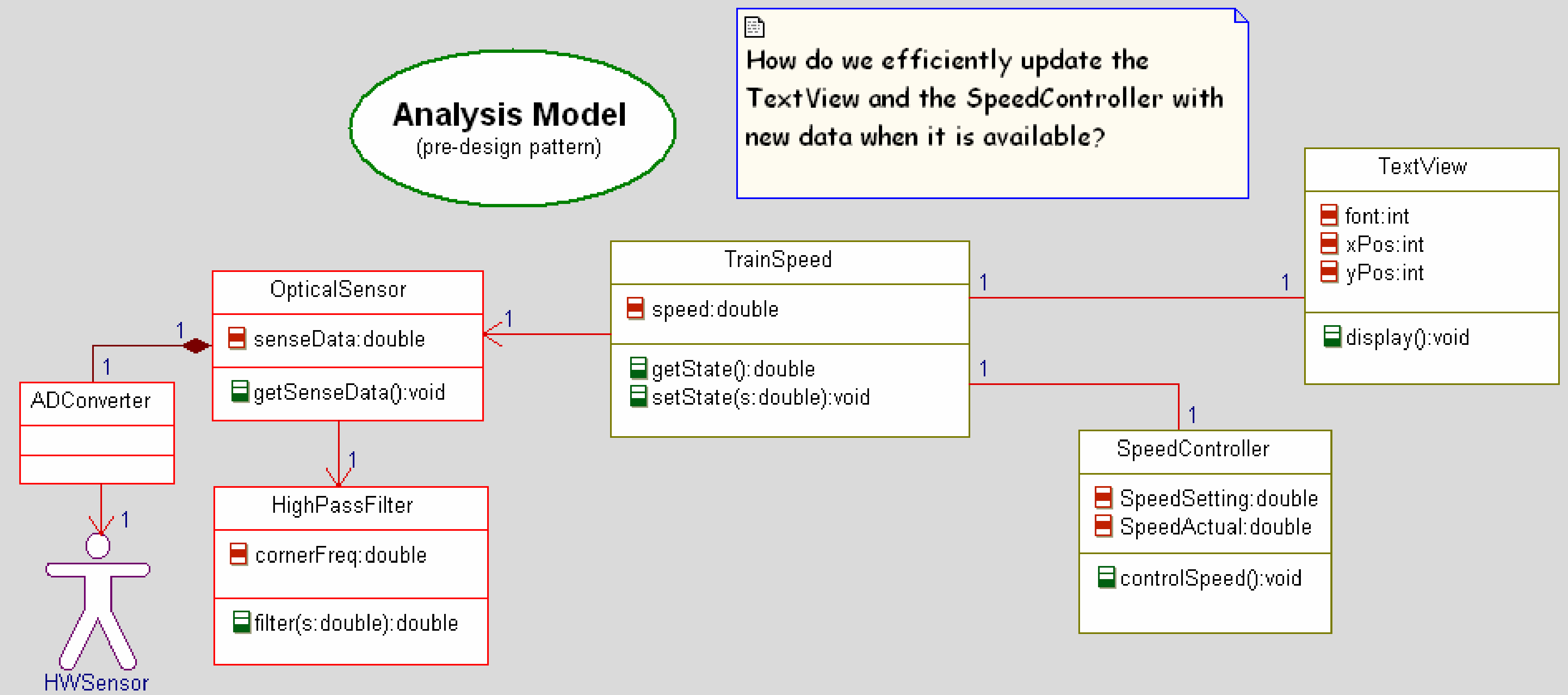
Detailed Design

- Refinement of details within objects themselves
  – Visibility
  – Internal decomposition of services
  – Internal data structuring and typing
  – Internal safety and reliability means
    • Data redundancy
  – Internal implementation policy of associations and object messaging

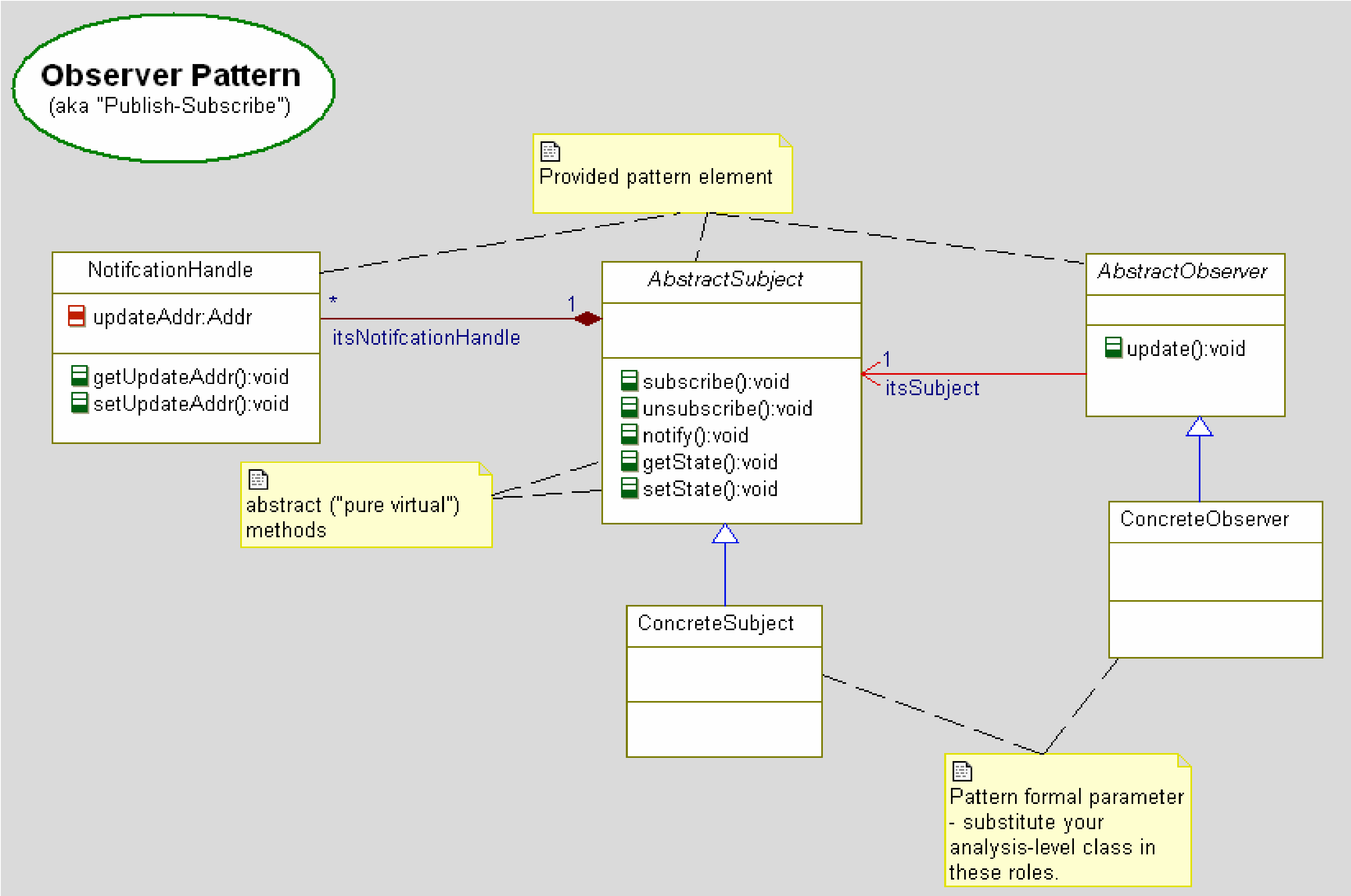
Design Patterns

- Design patterns are
  - generalized solutions to recurring optimization problems
  - reified structures of object collaboration that reappear in a variety of contexts
  - Parameterized collaborations of objects, where the object roles are the *formal parameters* and the objects that play those roles are the *actual parameters* when the pattern is instantiated
- UML has introduced a notation to explicitly capture design patterns.
- Shown as a dotted ellipse with dotted lines to the collaborating objects or classes

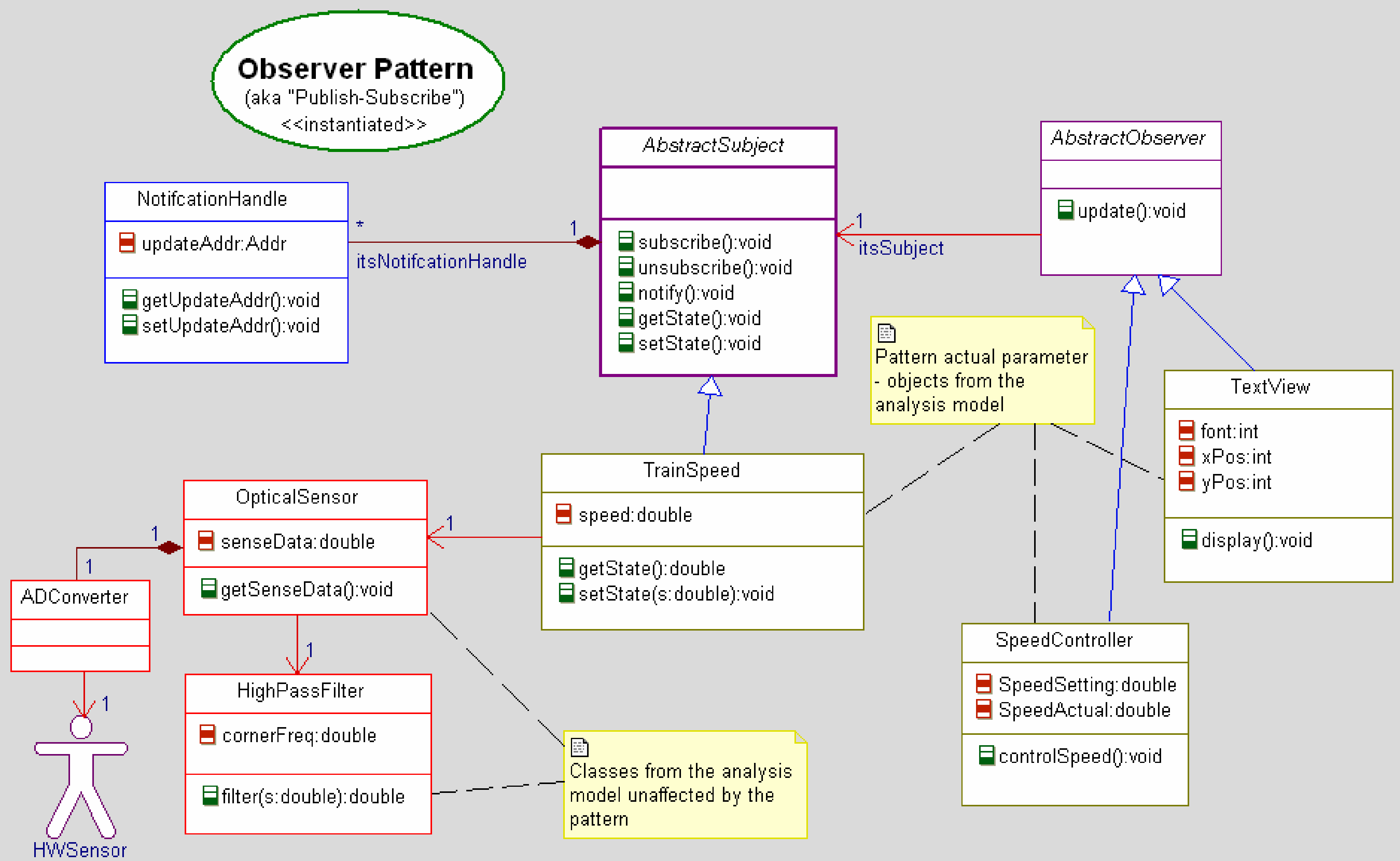
Example Analysis Model Collaboration



Design Pattern: Observer Pattern Specification



Design Pattern: Observer Pattern Instantiation



Telelogic

Why Use Design Patterns?

- Reuse effective design solutions
- Provide a more powerful vocabulary of design concepts to developers
- Develop “optimal” designs for specific design criteria
- Develop more understandable designs

How do I Apply Design Patterns???

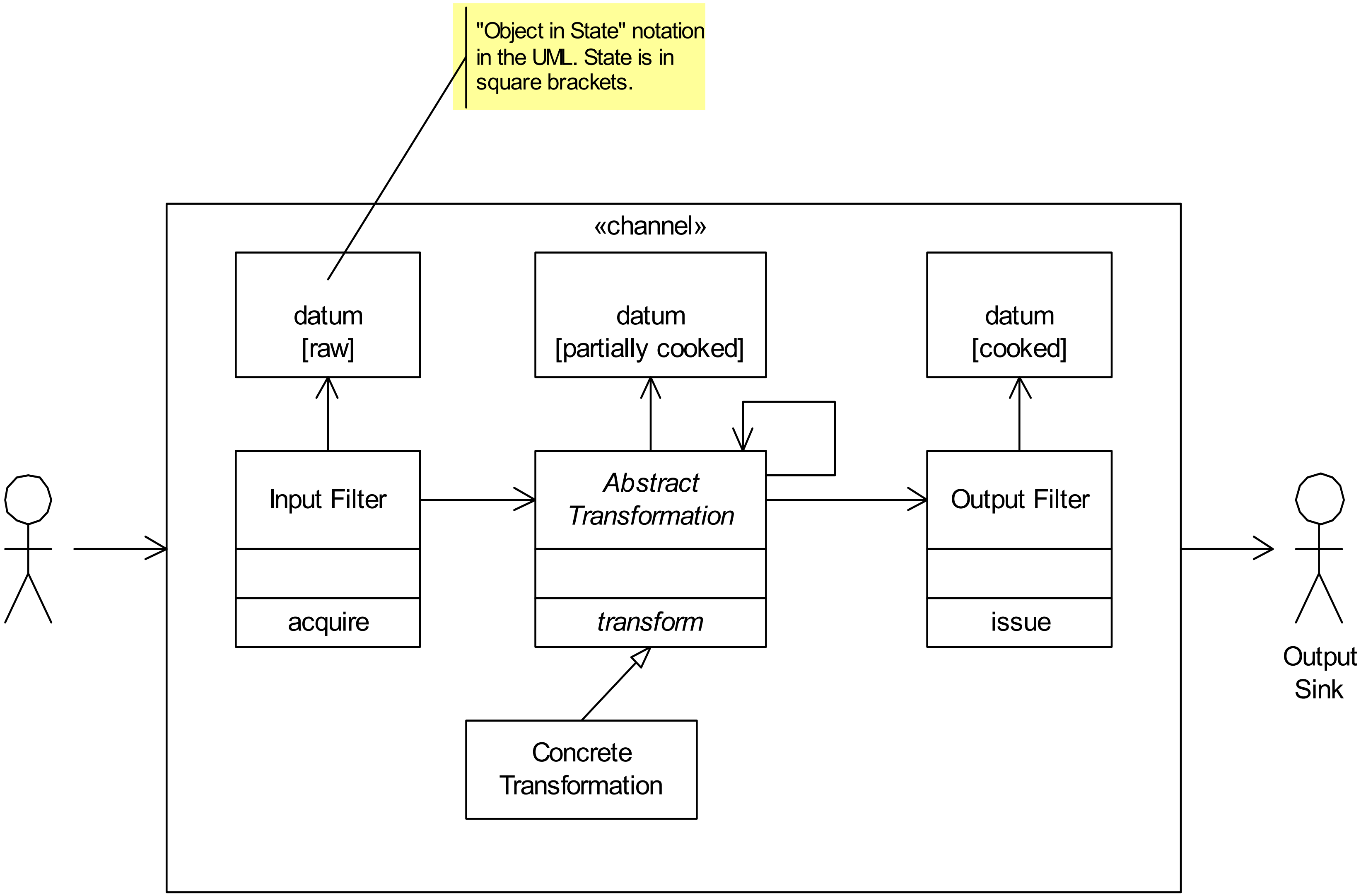
1. Construct the initial model
2. Identify the design criteria
3. Rank the design criteria in order of importance
4. Identify design patterns that optimize the system (architectural) or collaboration (mechanistic) for the critical design criteria at the expense of the lesser important ones
   1. Architectural patterns apply *system-wide*
   2. Mechanistic patterns apply *collaboration-wide*
   3. State behavioral patterns apply *object state machine-wide*

Architectural Design Patterns

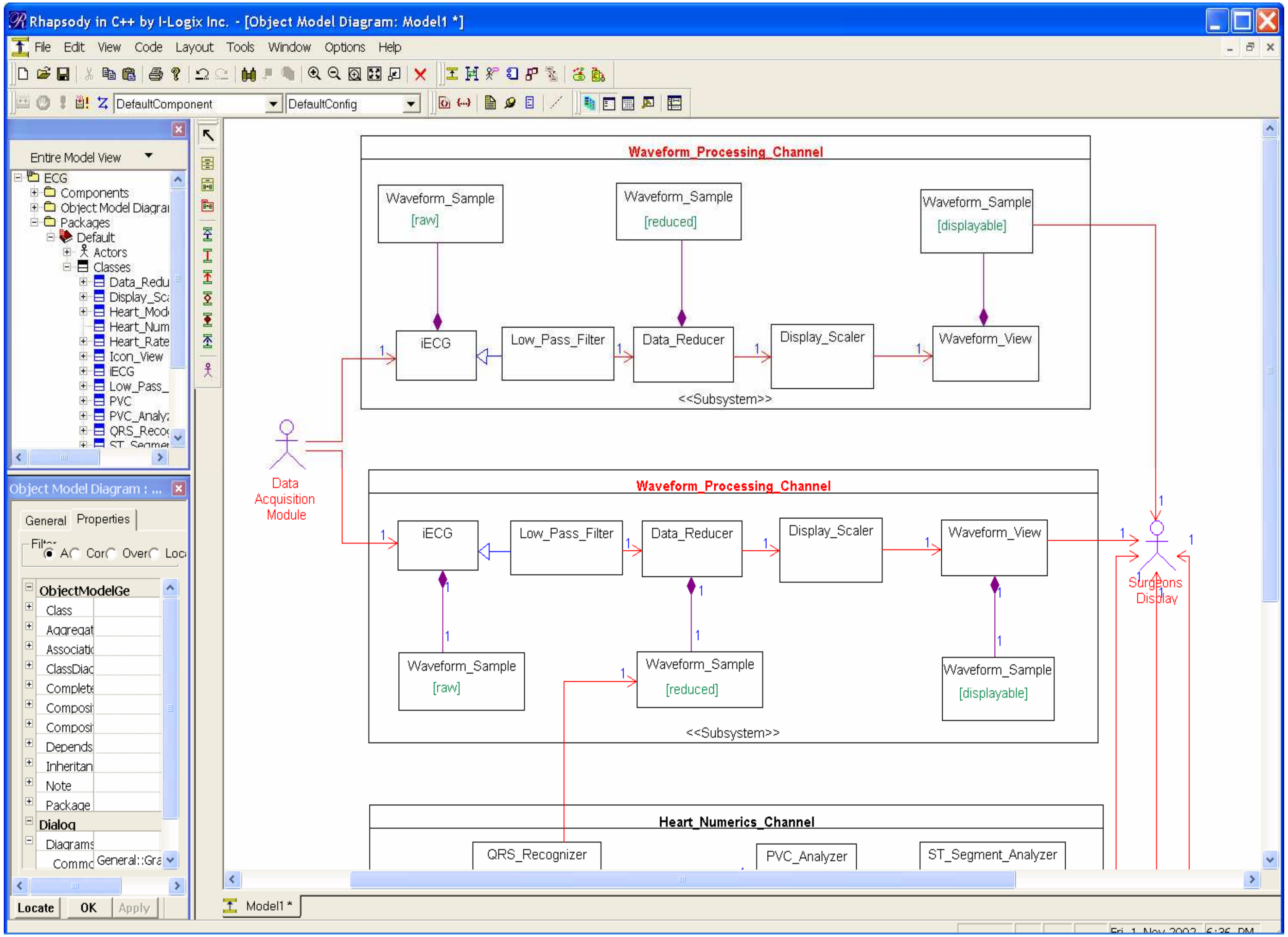
Channel Pattern

- Problem
  - Efficient execution of a system in which data is successively transformed in a series of steps
  - Want to organize and manage a hi-reliability, hi-availability, or safety-critical system that must provide redundancy of end-to-end behaviors
- Solution
  - Construct the system as a channel, a large scale subsystem which handles data from acquisition all the way through dependent actuation. Provide as many independent channels as necessary.
- Consequences
  - A simple organizational pattern that permits redundancy to be easily added.
  - May use additional memory since channels are designed to be independent, requiring replication (redundancy)

Channel Pattern



Channel Pattern Example



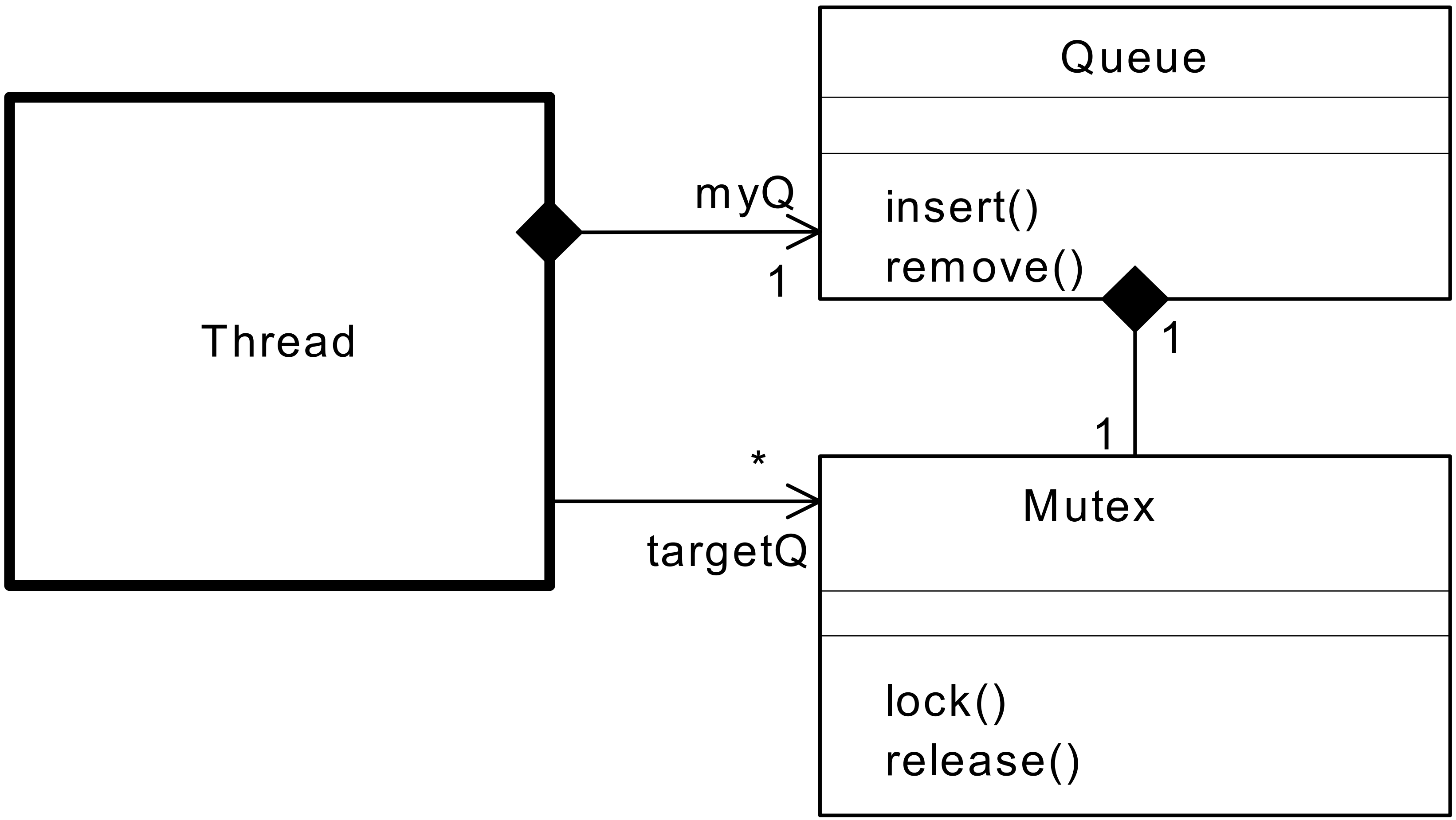
Telelogic

Concurrency Architecture Patterns

Message Queuing Pattern

- Problem
  - A simple way to request services be performed across thread boundaries in a thread-robust way
- Solution
  - Queue incoming services in the receiving thread until that thread runs – then dequeue and service the requests
- Consequences
  - This approach is simple and supported by most operating systems
  - Service responses are delayed until the target thread actually runs, so response may not be timely
  - No mutual exclusion problem because requests are serialized

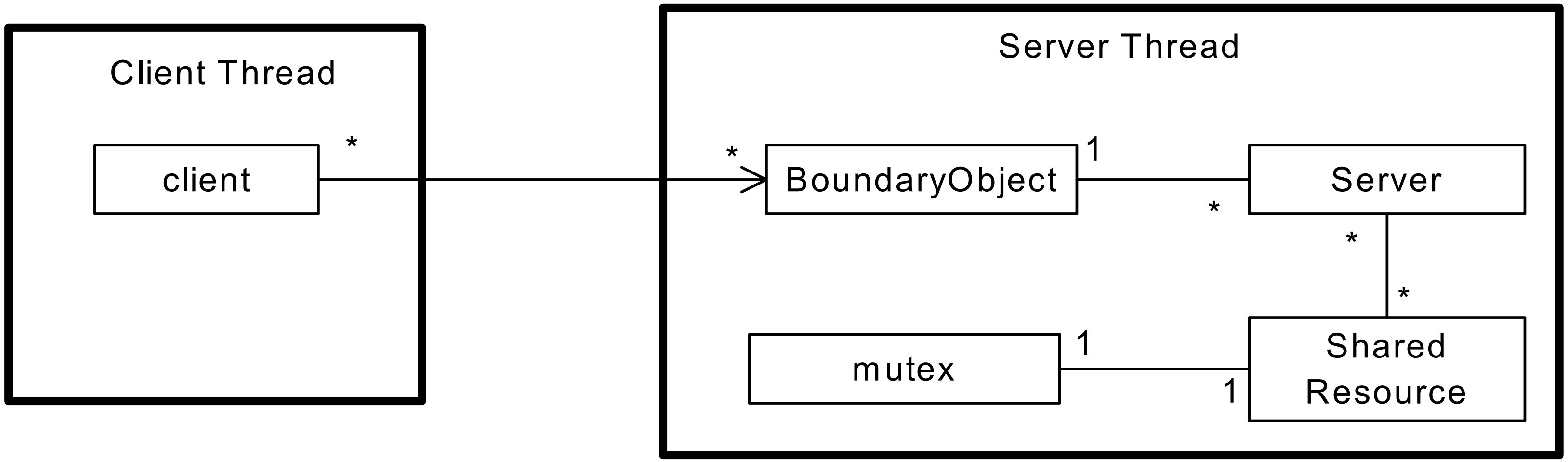
Message Queuing Pattern



Guarded Call Pattern

- Problem
  - Need timely response to service requests across multiple threads, and the synchronization across thread boundaries must handle mutual exclusion issues for thread-safe rendezvous
- Solution
  - Permit calls to methods of object across thread boundaries, but protect those calls with a mutual exclusion semaphore, 1 semaphore per shared object
- Consequences
  - A simple solution supported by most operating systems
  - Can lead to *blocking* when the target object is currently locked
  - Can lead to *unbounded priority inversion* unless a priority inversion control pattern is also applied (e.g. Highest Locker or Priority Inheritance)

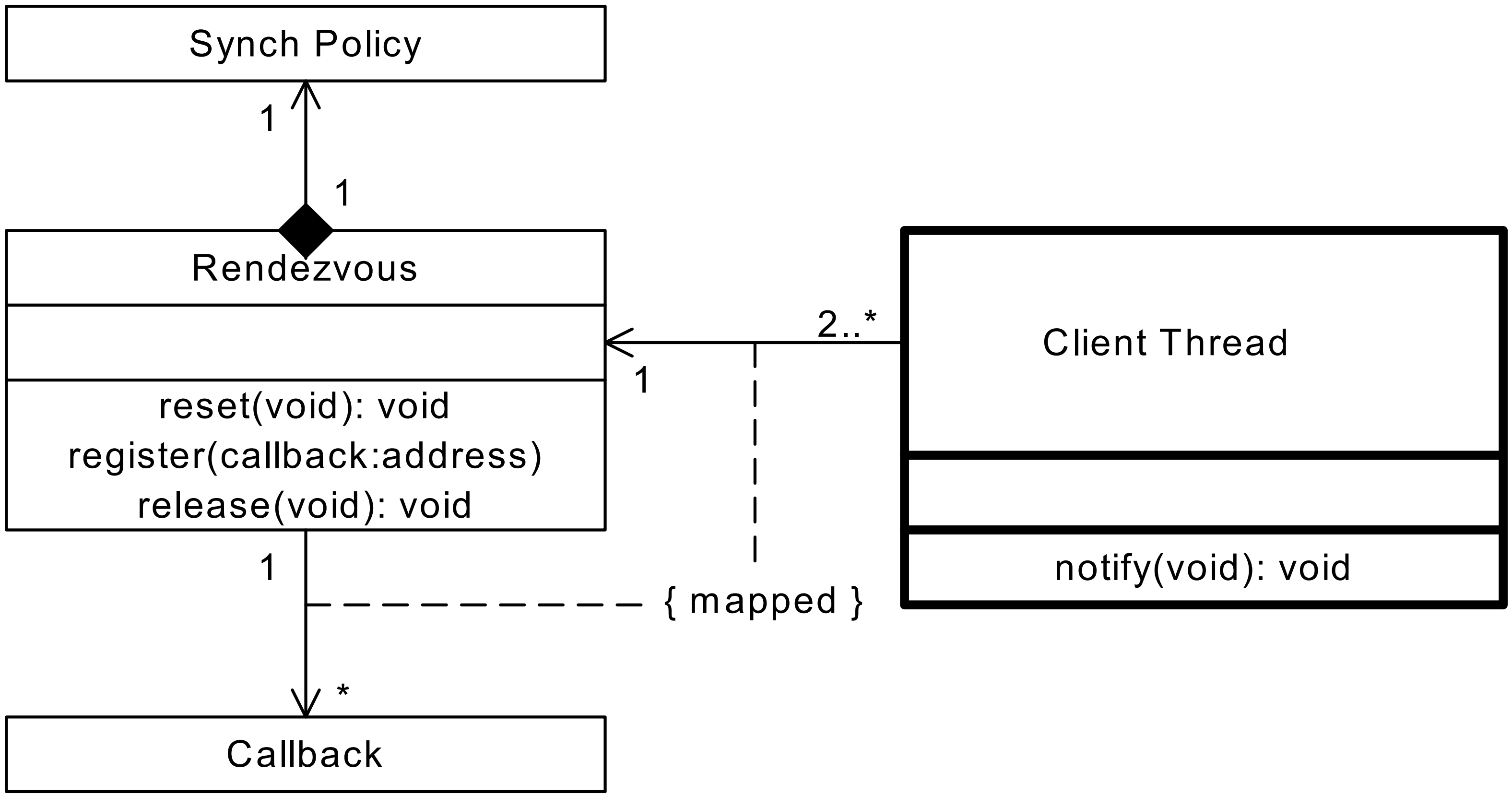
Guarded Call Pattern



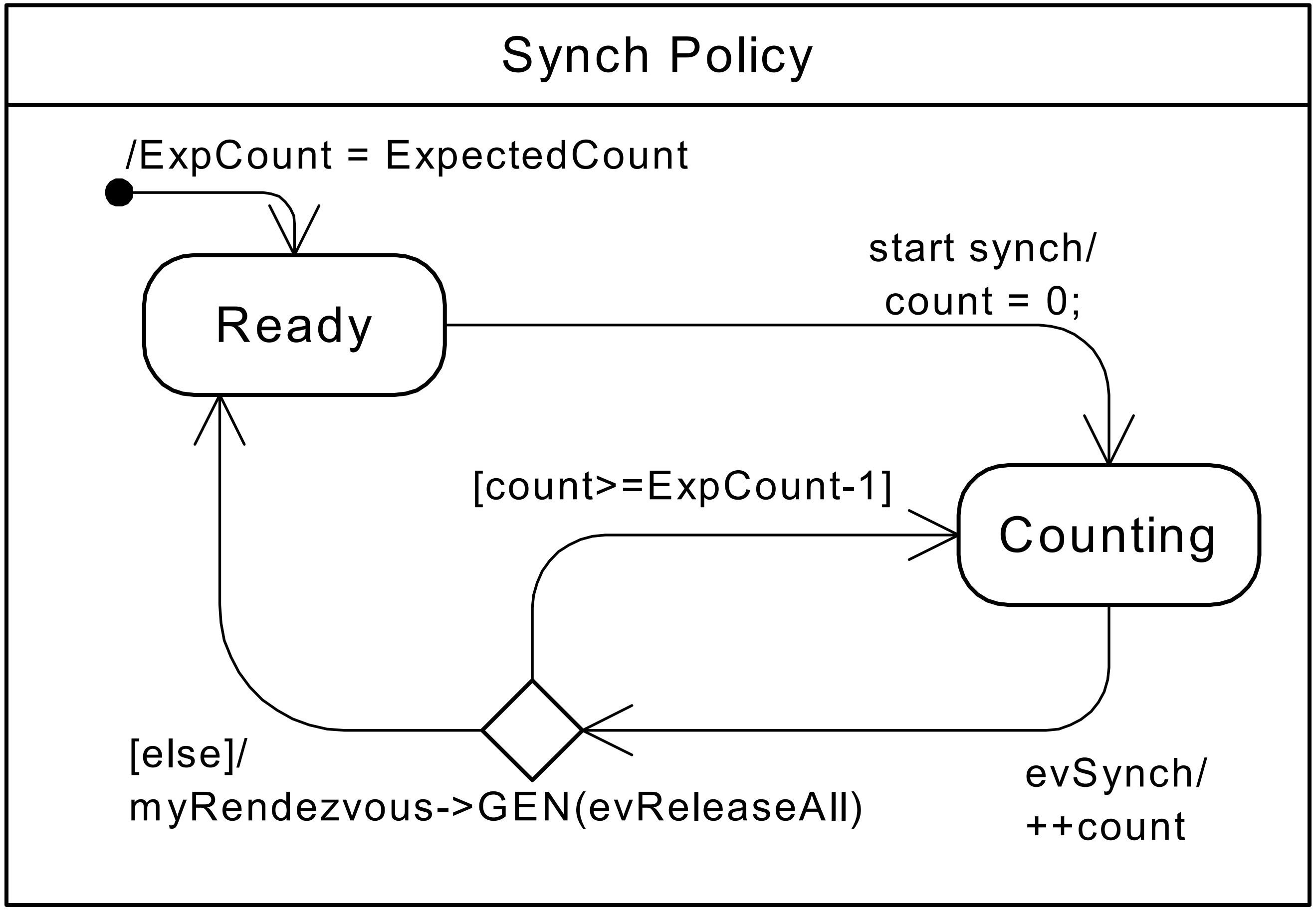
Rendezvous Pattern

- Problem
  - Need a collaboration structure that allows any arbitrary set of preconditional invariants to be met for thread synchronization, independent of task phasings, scheduling policies, and priorities.
- Solution
  - Reify the rendezvous policy as a class that mediates how the threads collaborate
- Consequences
  - An easy-to-implement pattern that can implement an arbitrarily complex set of thread rendezvous preconditions
  - *Thread Barrier Pattern* is a common instantiation of this pattern

Rendezvous Pattern



Thread Barrier Pattern Statechart



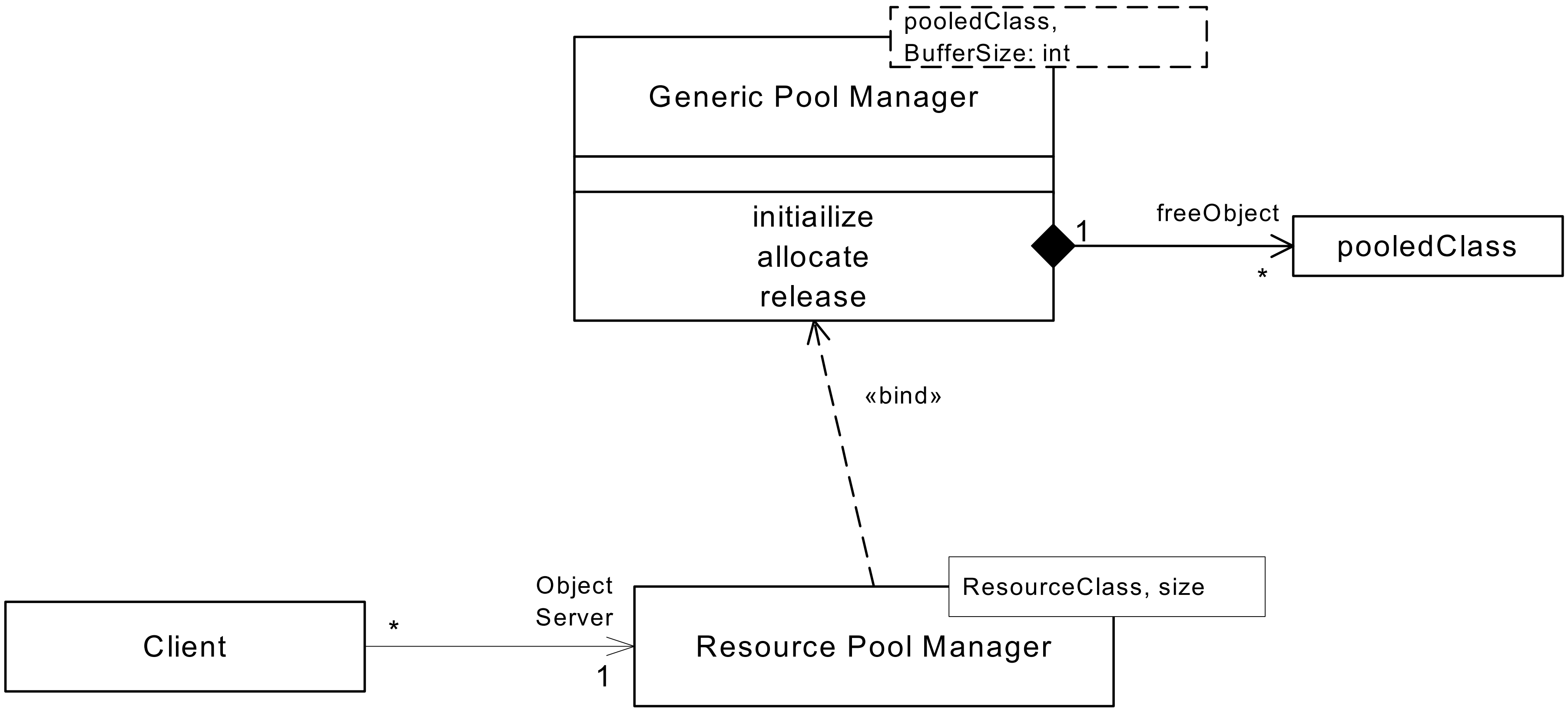
Memory Patterns

- Pooled Allocation Pattern
- Fixed Sized Buffer Allocation Pattern
- Smart Pointer Pattern

Pooled Allocation Pattern

- Problem
  - In a situation that cannot use dynamic allocation during run time, we need to use many small objects but we cannot statically allocate their ownership in the worst case, even though we CAN handle all worst cases
- Solution
  - Preallocate pools of small shared objects, giving them to the clients as necessary, who return them when done
- Consequences
  - No memory fragmentation
  - No unpredictable memory allocation
  - Handles more complex situations than the Static Allocation Pattern

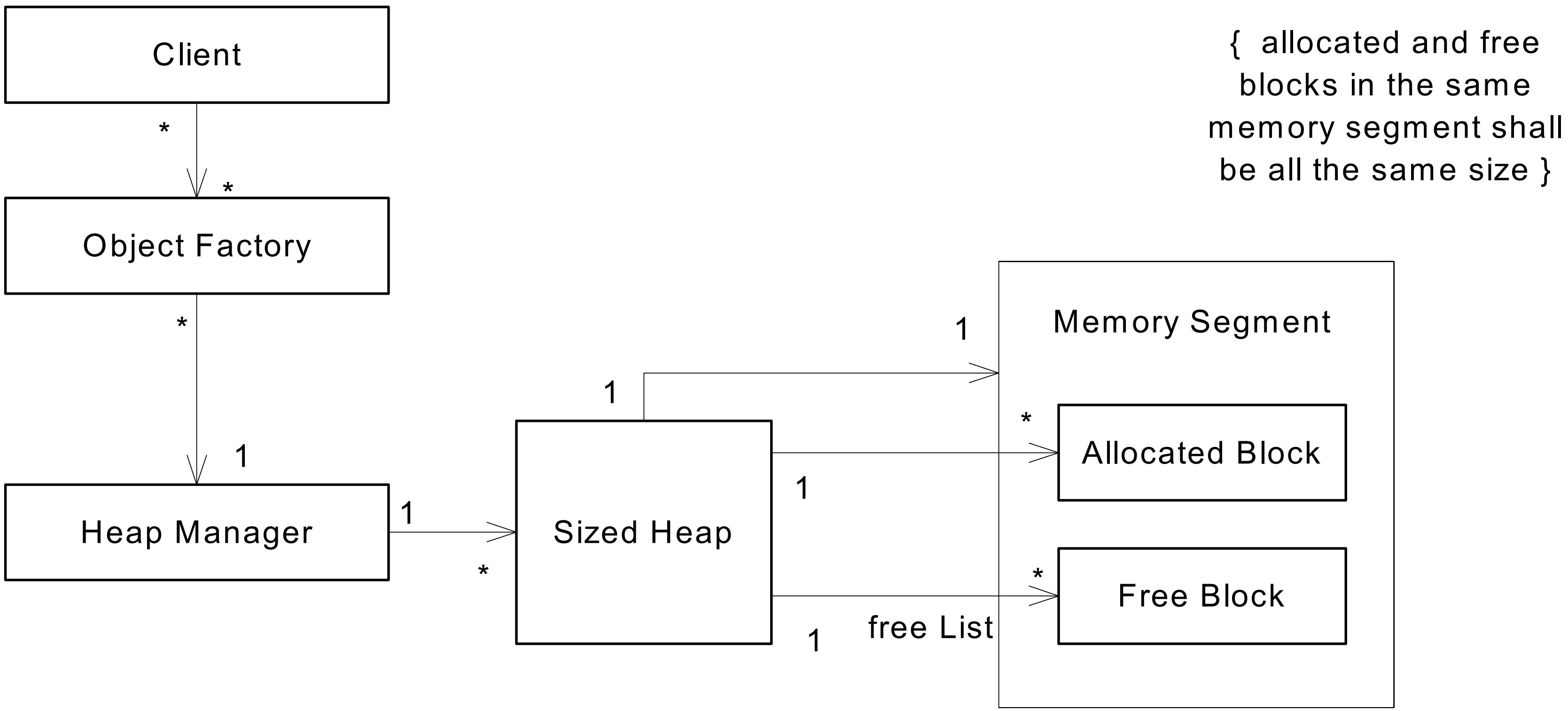
Pooled Allocation Pattern



Fixed-Sized Buffer Pattern

- Problem
  – In situations complex enough to require dynamic memory allocation, sometimes we cannot live with memory fragmentation
- Solution
  – Allocate memory dynamically in fixed sized chunks which are predetermined to allow us to *always* satisfy a memory request if any memory is available
- Consequences
  – Eliminates memory fragmentation
  – Requires static (design) analysis to determine optimal block sizes and number of sized heaps
  – Wastes memory since it is always allocated in fixed sized blocks

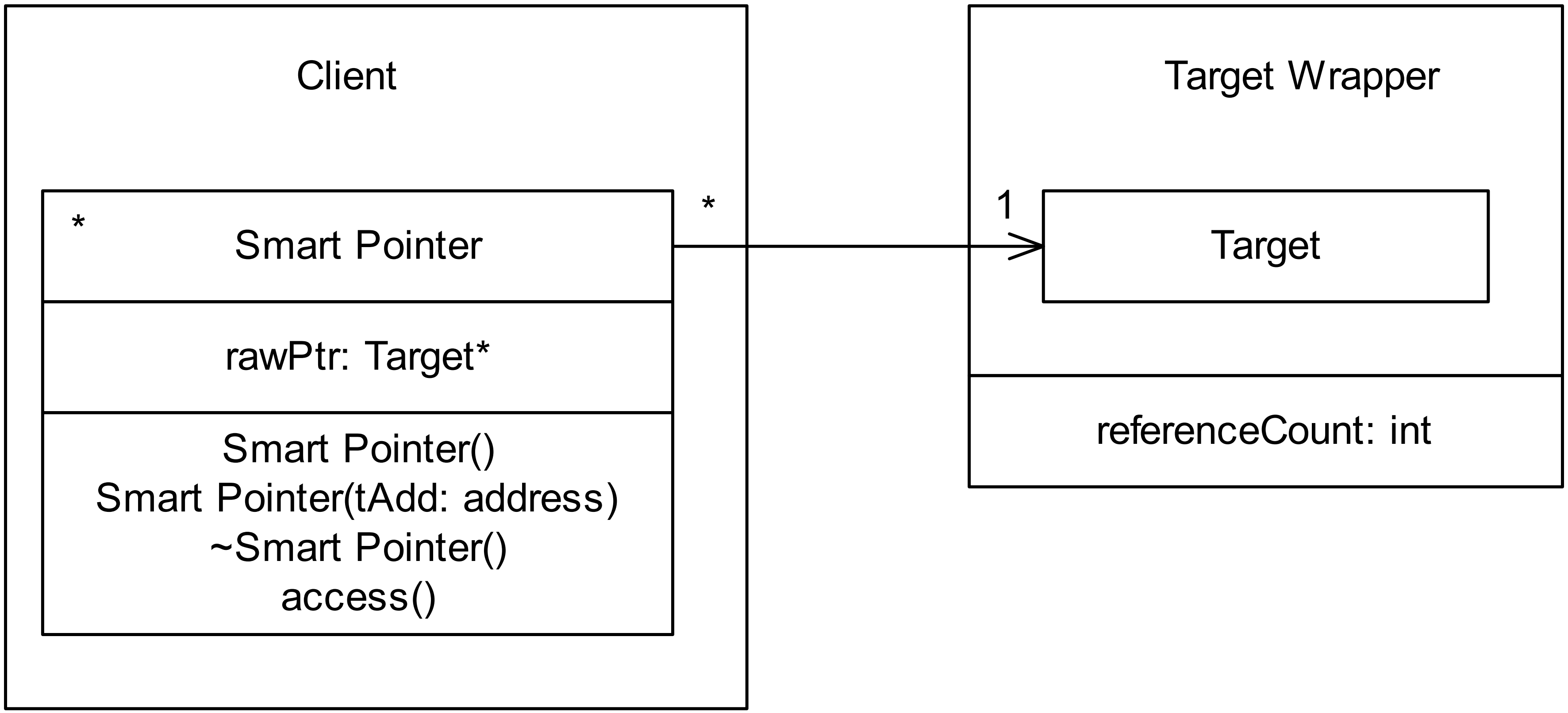
Fixed-Sized Buffer Pattern



Smart Pointer Pattern

- Problem
  - Pointers are very powerful but induce many kinds of errors. This is because they do not have a means to ensure pre- and post-conditional invariants
- Solution
  - Reify the pointers as a *class* so that access to the pointer behavior can be controlled via operations that ensure correct behavior
- Consequences
  - Stops most common pointer errors – dangling pointer, memory leaks, pointer arithmetic
  - Doesn’t work well with cyclic data structures

Smart Pointer Pattern



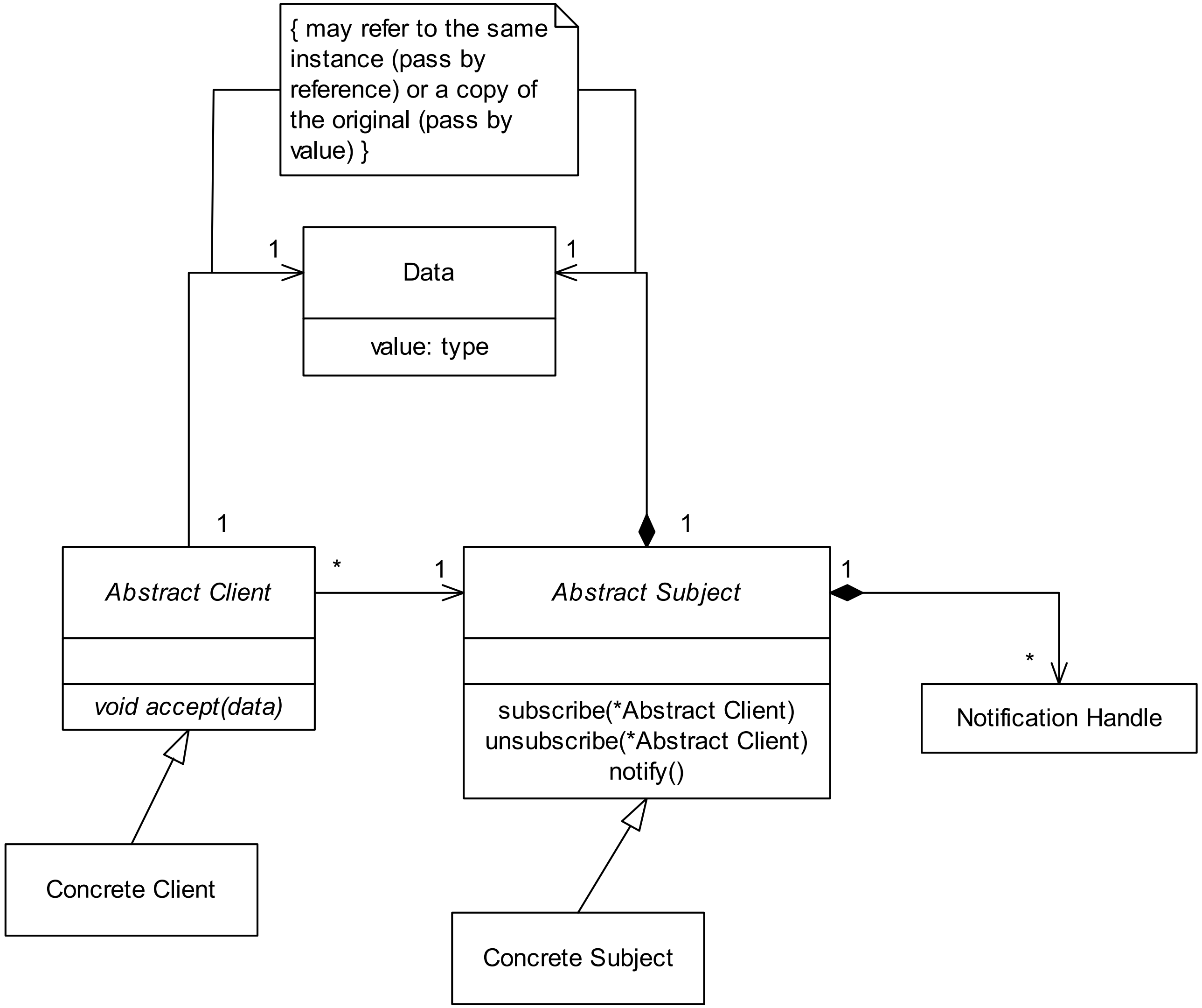
Distribution Patterns

- Observer Pattern
- Data Bus Pattern
- Proxy Pattern
- Broker Pattern

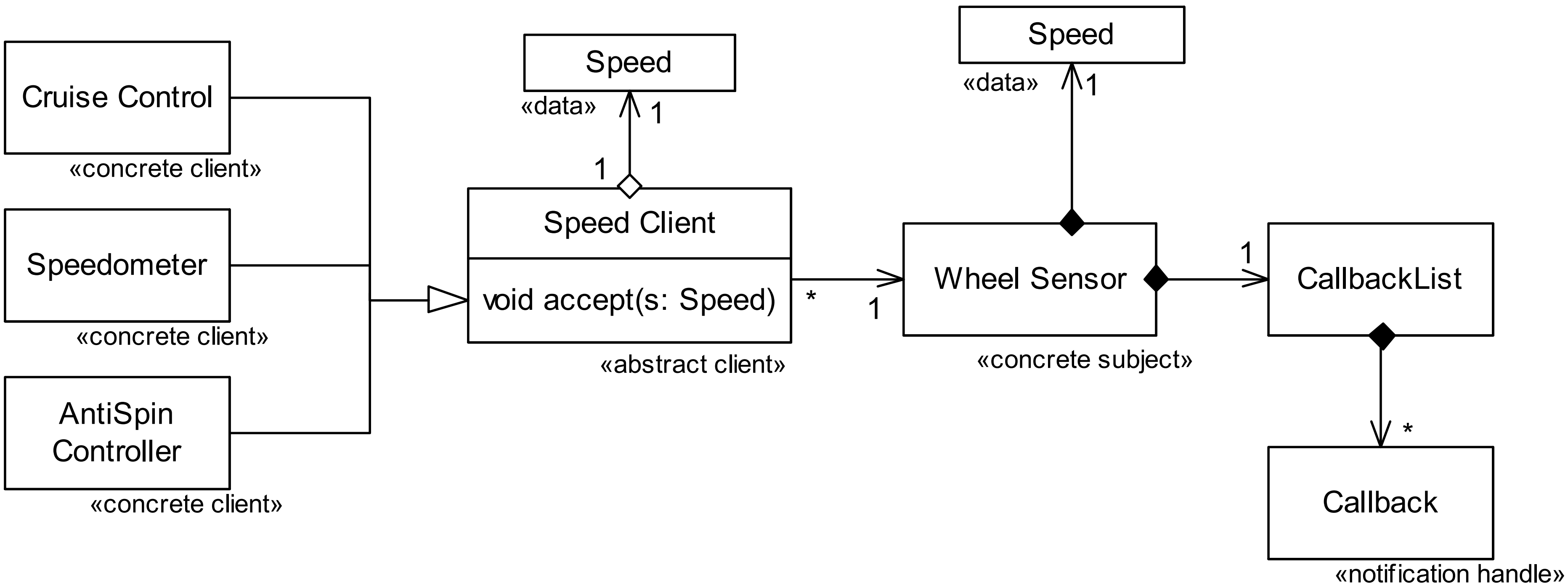
Observer Pattern

- Problem
  - How to efficiently give up-to-date data to clients in a timely way
- Solution
  - Have clients subscribe to the server. When new data is available, the server walks the client list and sends them the new data
- Consequences
  - Works well with minimal complexity

Observer Pattern



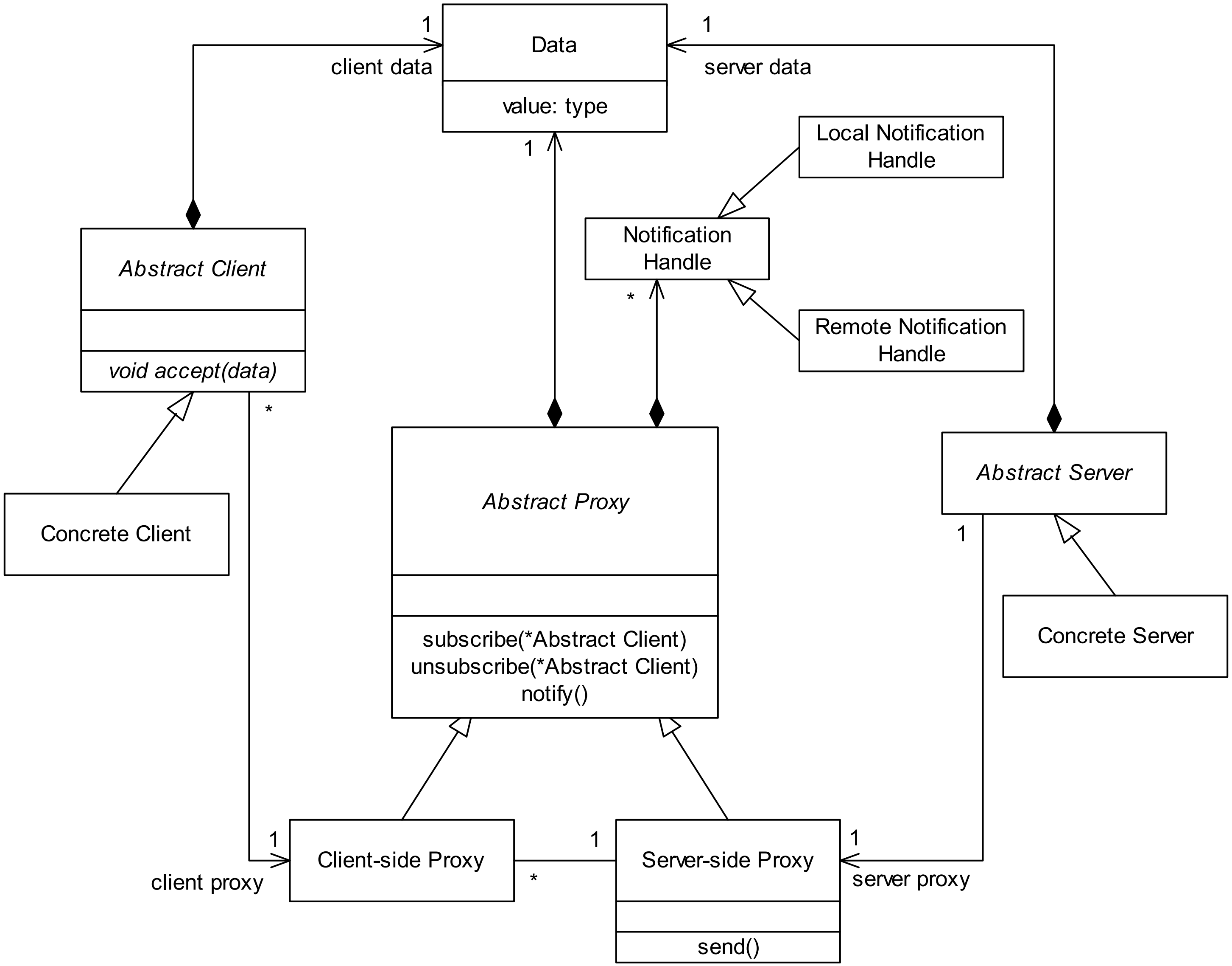
Observer Pattern Example



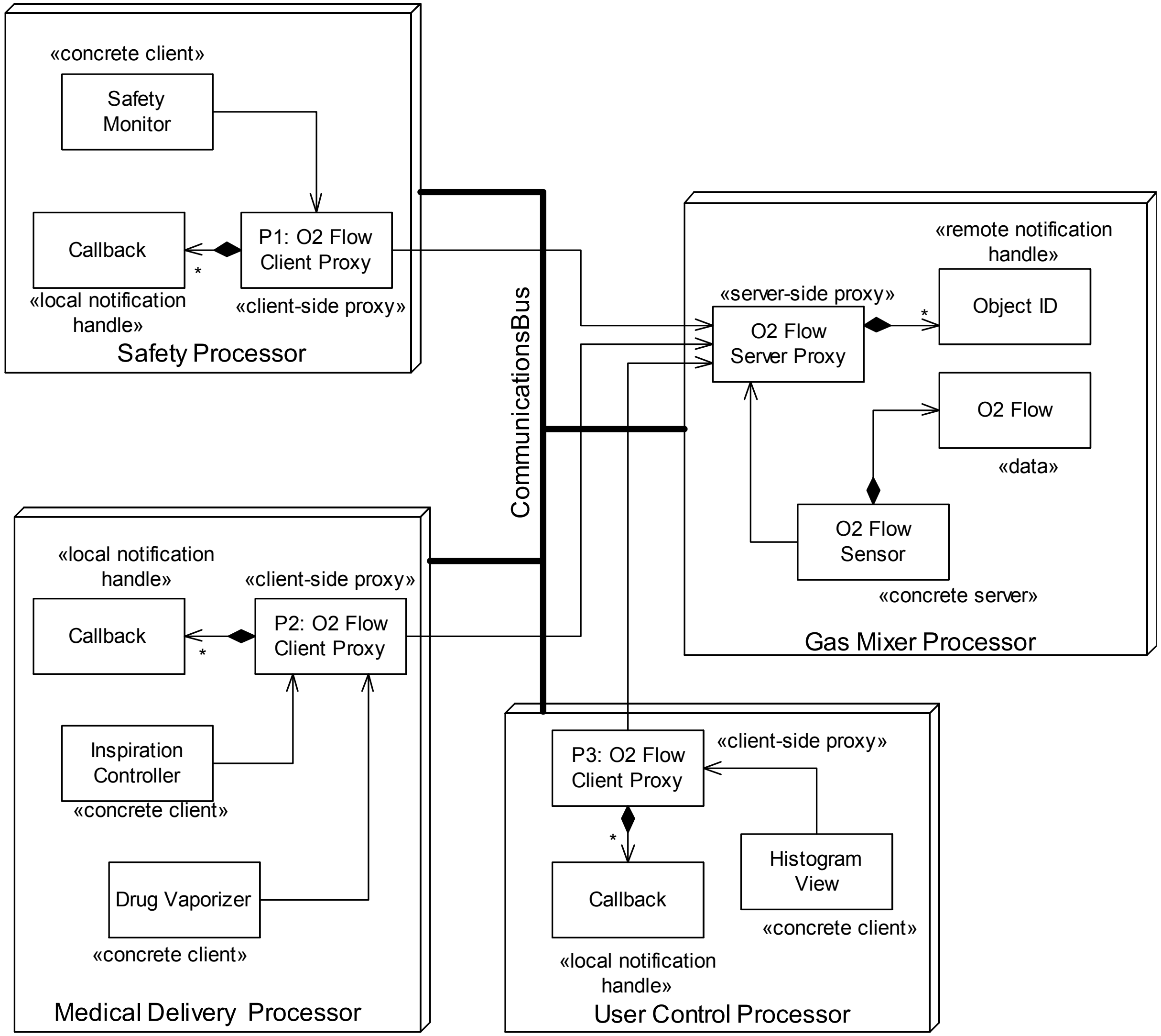
Proxy Pattern

- Problem
  - Use an observer pattern across multiple address spaces with a variety of different servers and clients, isolating away the knowledge of the infrastructure of the means to communicate
- Solution
  - Similar to a observer pattern but with client and server proxies to isolate the required infrastructure
- Consequences
  - A simple adaptation of the observer pattern
  - Good isolation of subject from communications
  - Optimizes bus traffic

Proxy Pattern



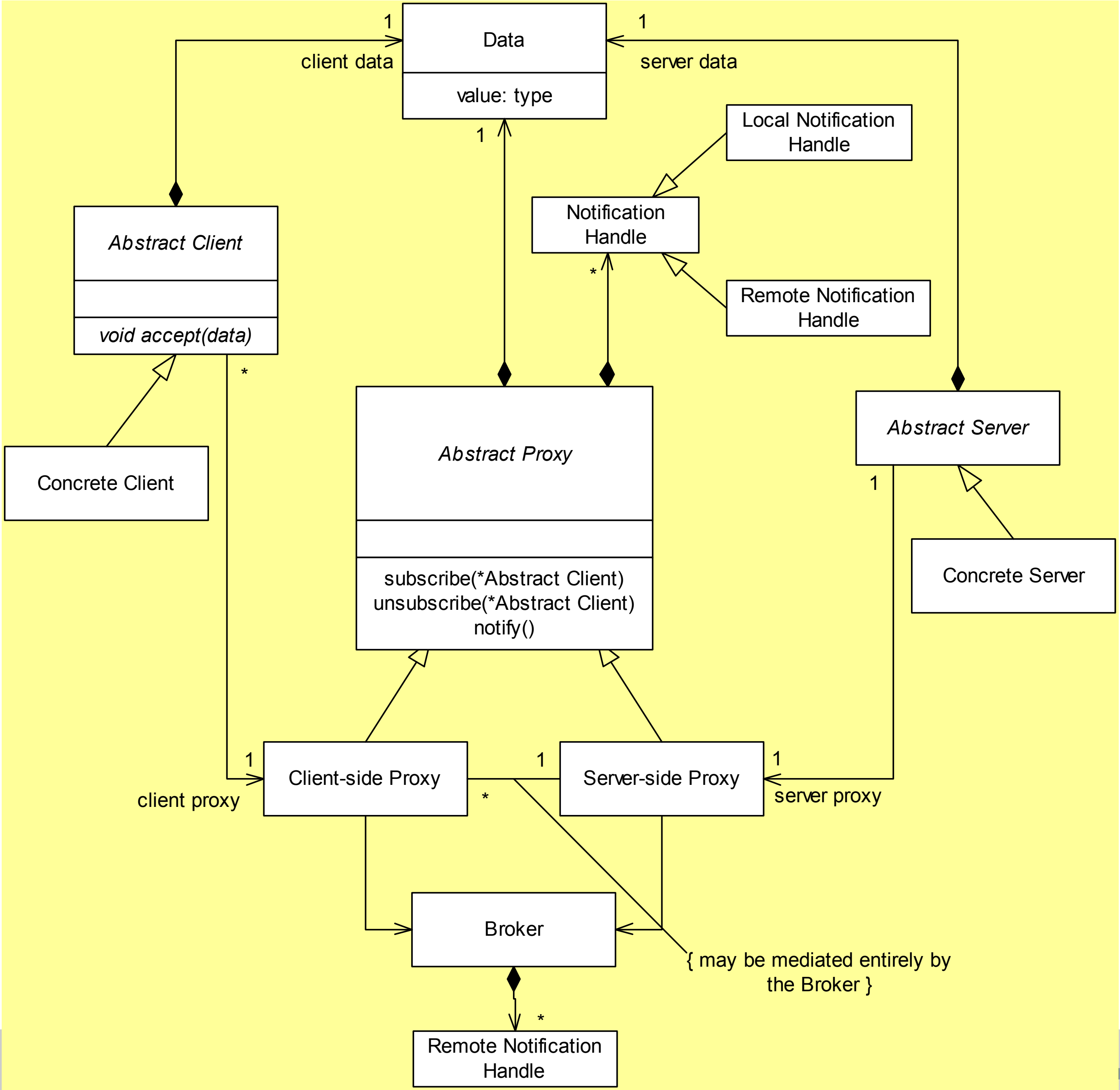
Proxy Pattern Example



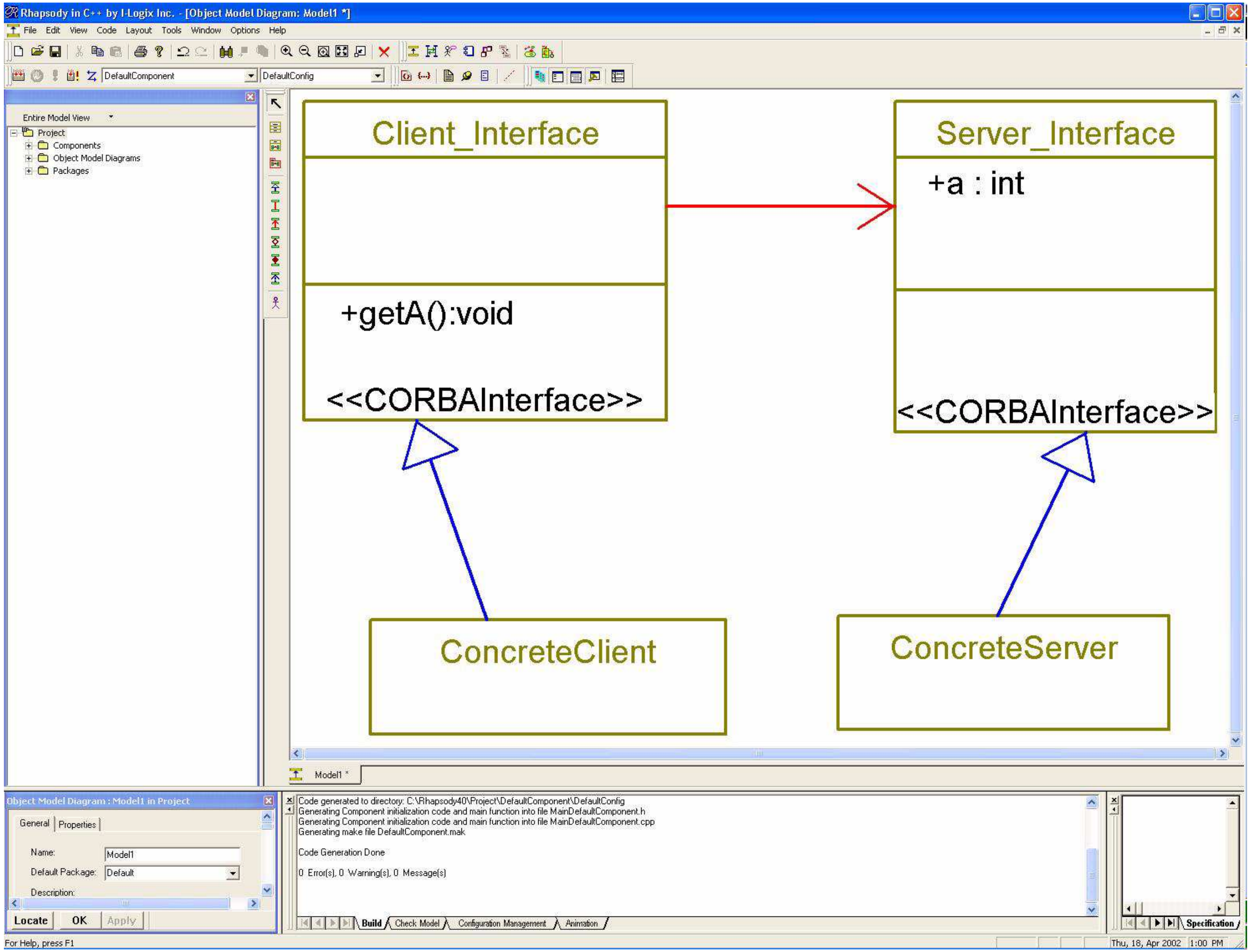
Broker Pattern

- Problem
  – Support a symmetric distribution architecture (as for dynamic load balancing), and allow objects to find each other without knowing their locations a priori
- Solution
  – Add a broker as an object repository, such that when servers run, they register with the broker; when clients need to access a server, they request the location of the server from the broker
- Consequences
  – Good support for symmetric architectures
  – Good commercial support with CORBA

Broker Pattern

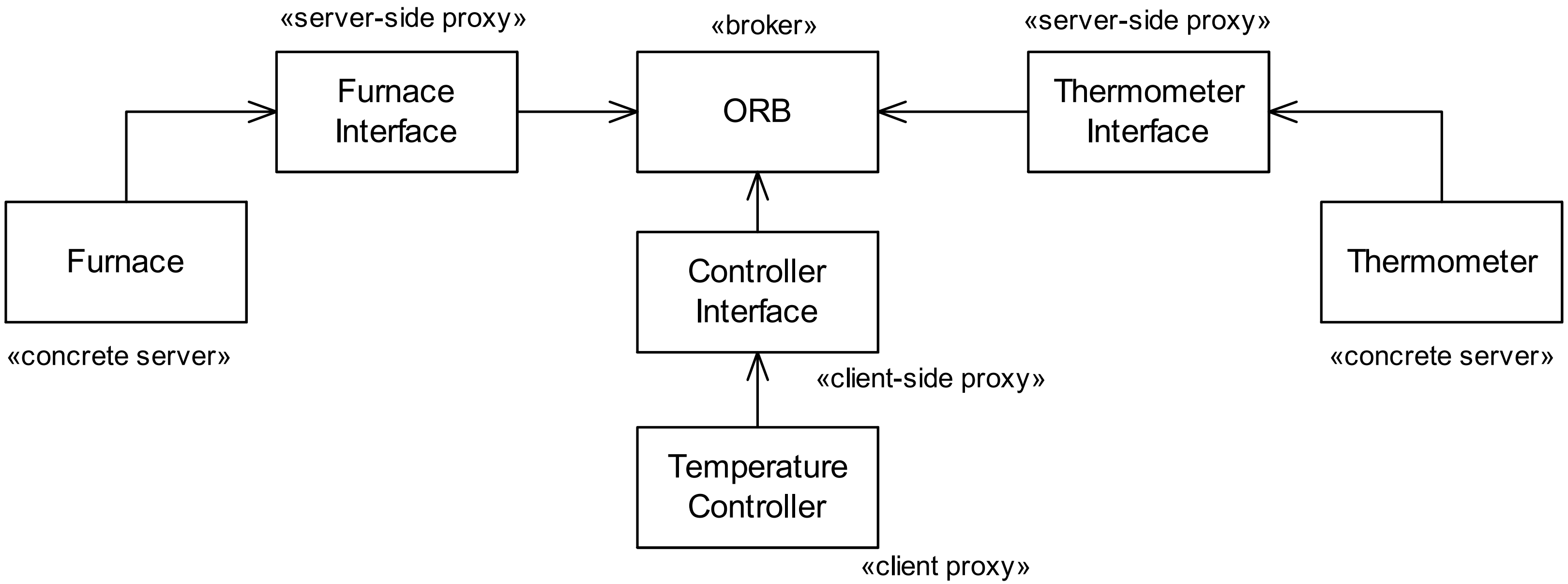


Broker Pattern Example



Telelogic

Broker Pattern Example



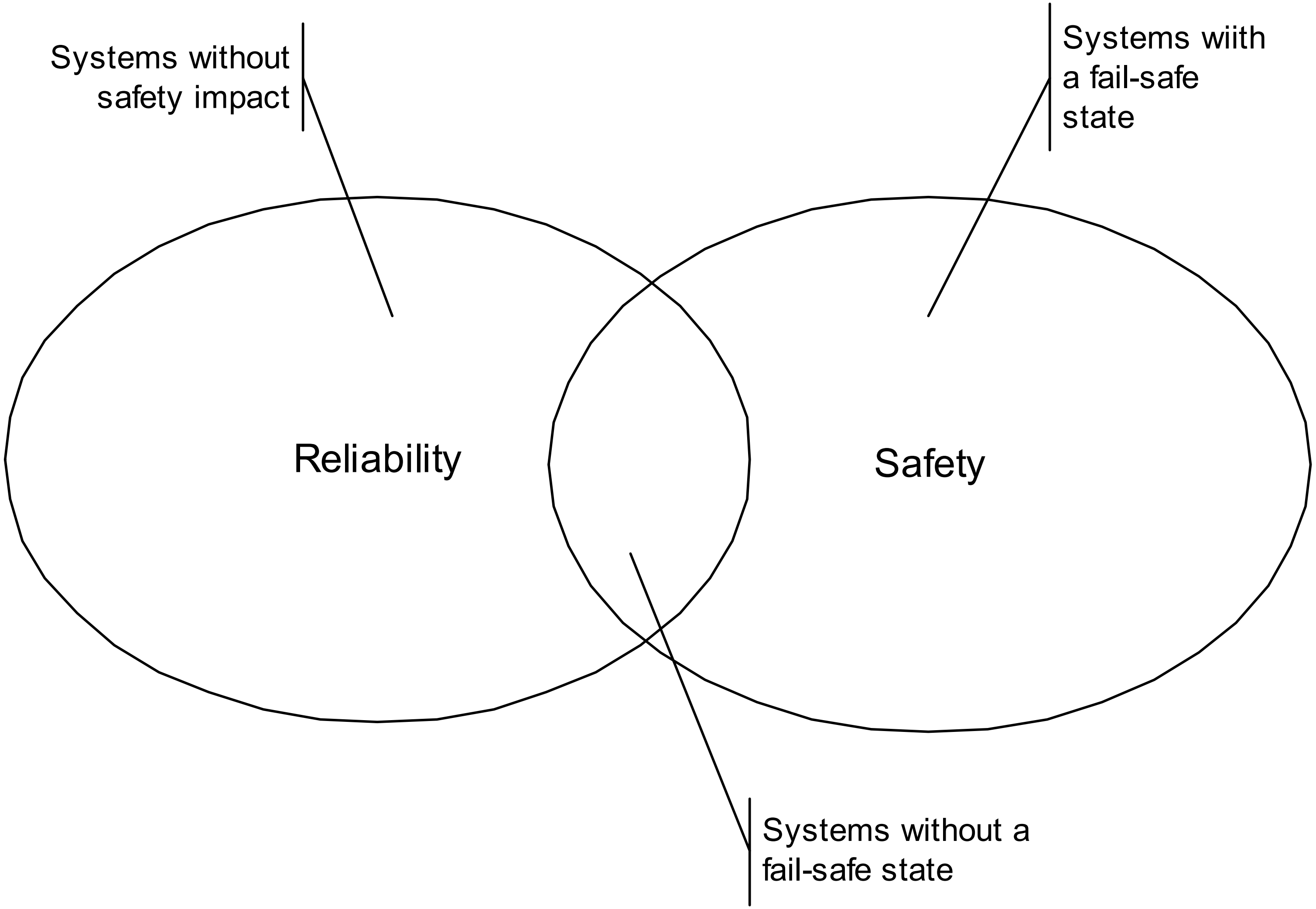
Safety and Reliability Patterns

- Protected Single Channel Pattern
- Homogeneous Redundancy Pattern
- Heterogeneous Redundancy Pattern
- Triple Modular Redundancy Pattern
- Monitor-Actuator Pattern

Safety and Reliability

- Reliability is measured by MTBF
  – MTBF = probability of fault
- Safety is measured by risk
  – risk = severity * likelihood
- Fault is nonconformant behavior
  – Error – a design or implementation flaw
  – Failure – breakage of something that was previously correct

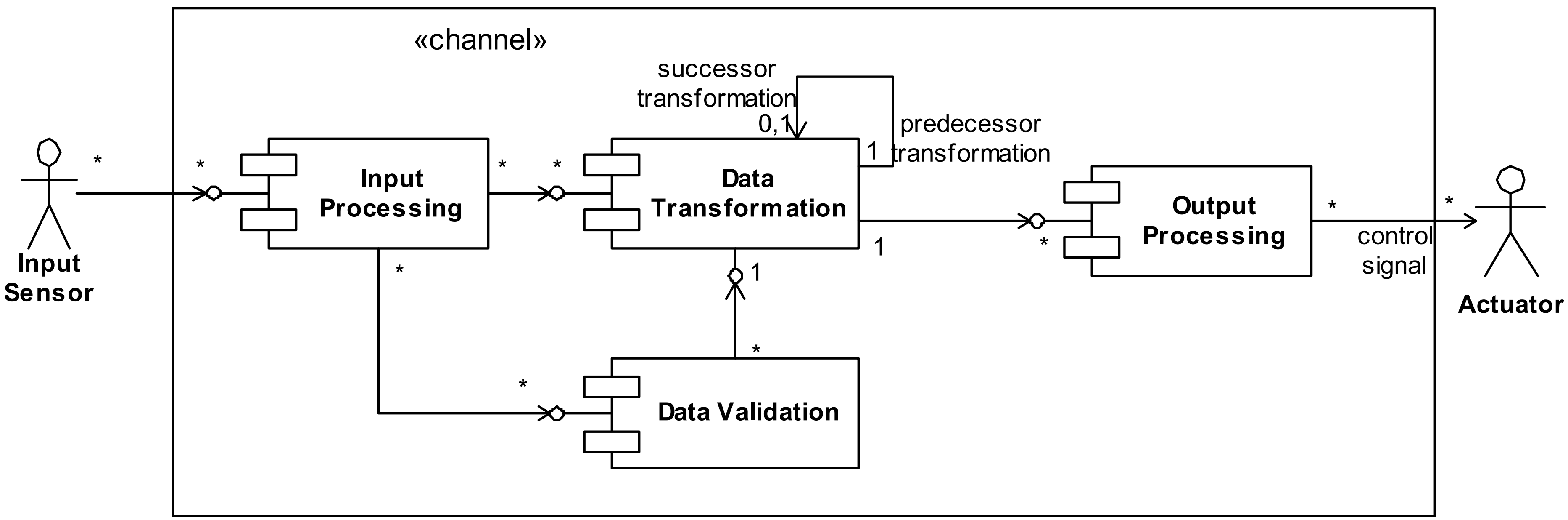
Safety vs Reliability



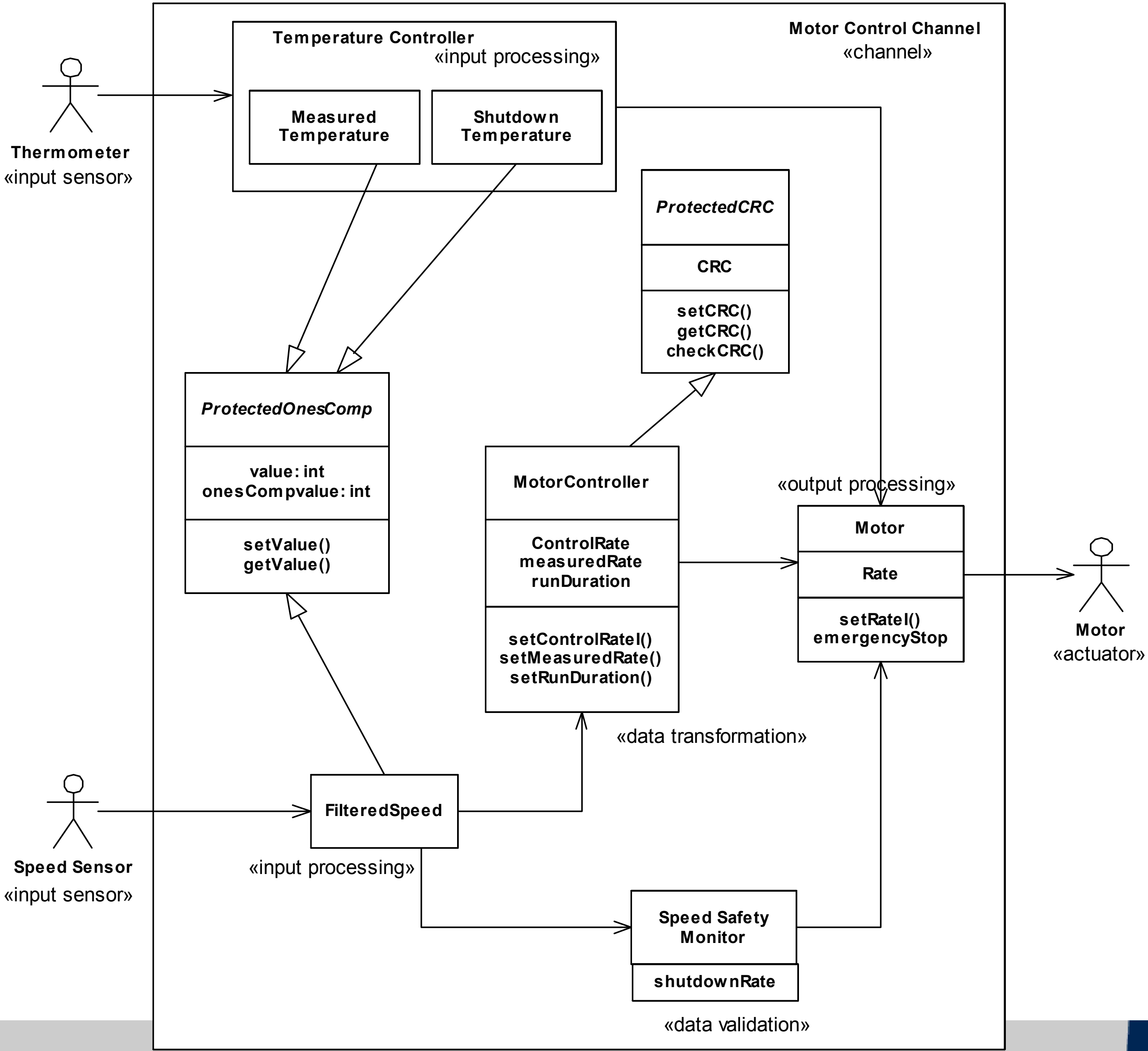
Protected Single Channel Pattern

- Problem
  - Provide protection against errors (design flaws) in a cost effective way
- Solution
  - A variant of the Channel pattern the uses light-weight redundancy to provide identification of errors
- Consequences
  - Low design cost
  - Low recurring cost
  - Not able to continue in the presence of faults

Protected Single Channel Pattern



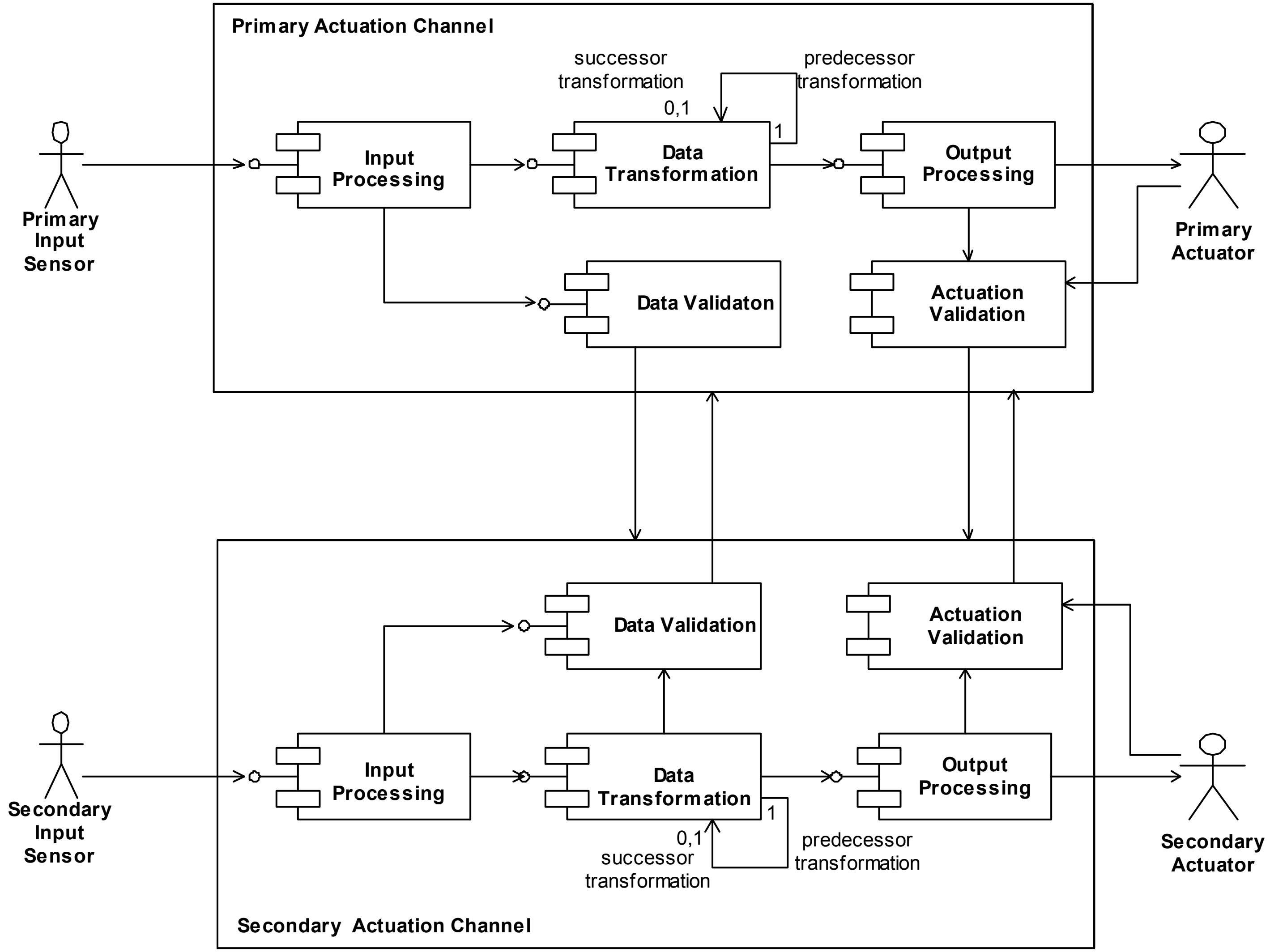
Protected Single Channel Pattern Example



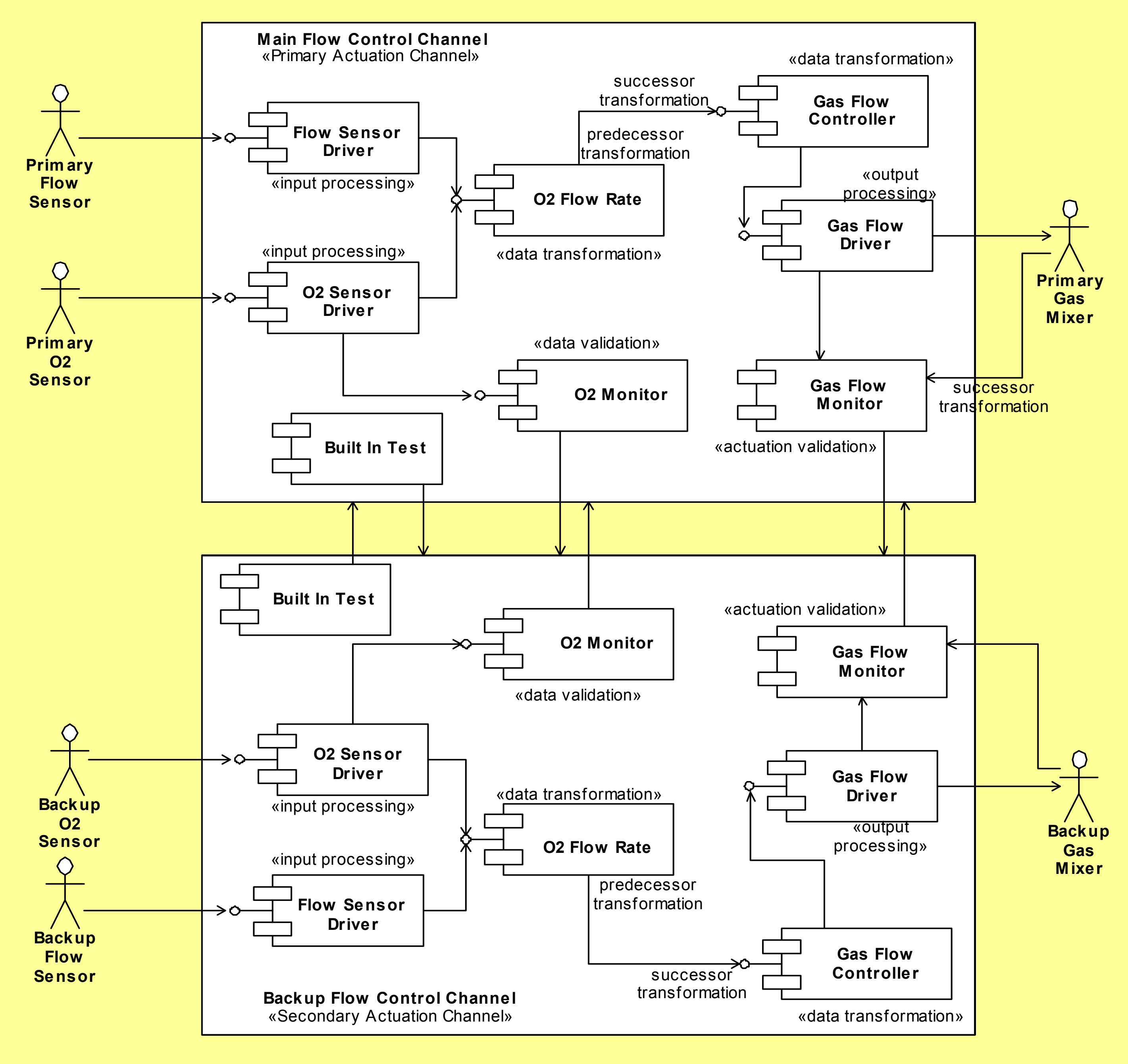
Homogeneous Redundancy Pattern

- Problem
  - Provide the ability to continue in the presence of a fault
- Solution
  - Provide “redundancy in the large” by replicating channels
- Consequences
  - Low design-time cost
  - High recurring cost
  - Able to continue in the presence of a failure
  - Does not recover from *errors*

Homogeneous Redundancy Pattern



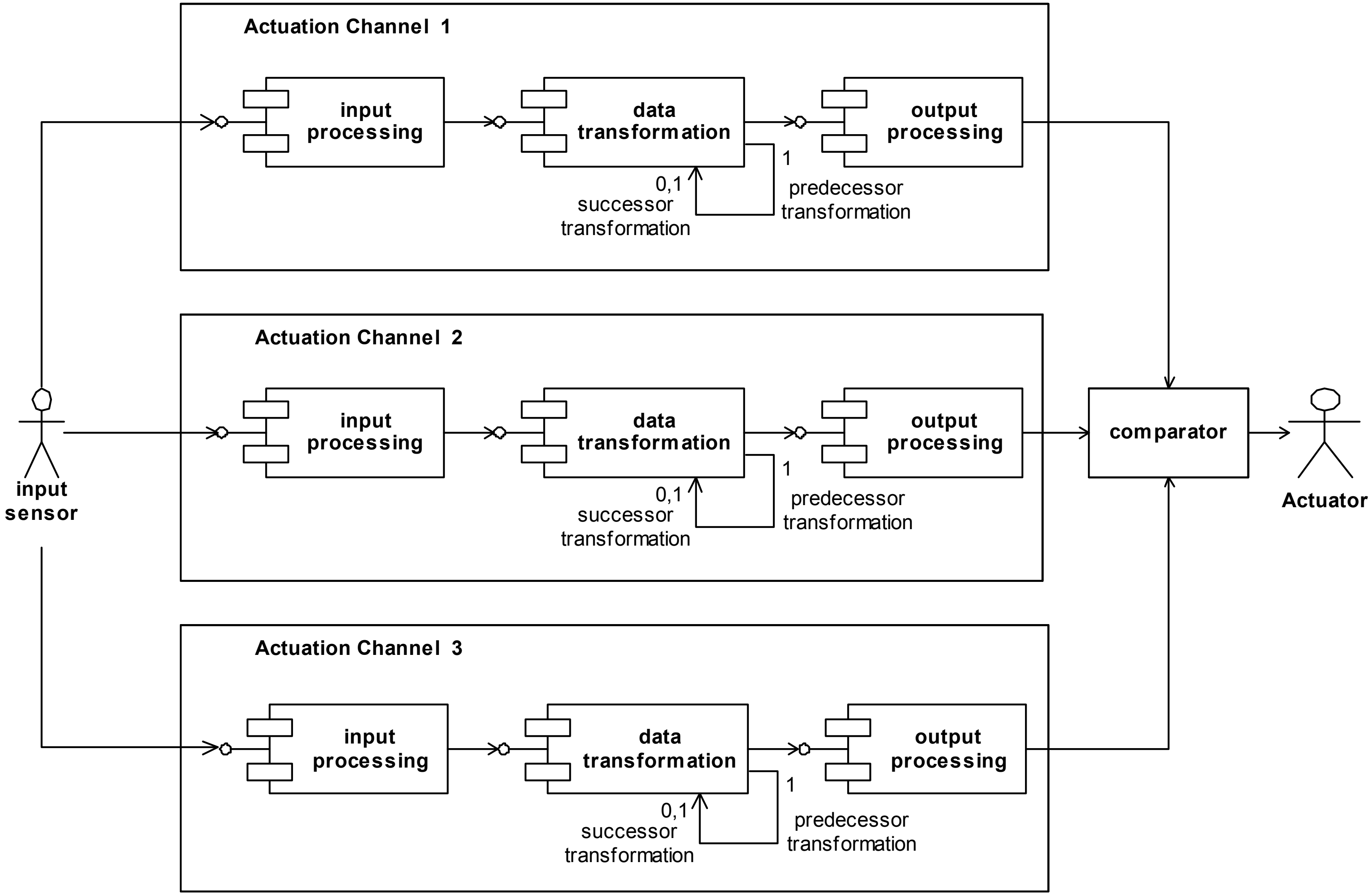
Homogeneous Redundancy Pattern Example



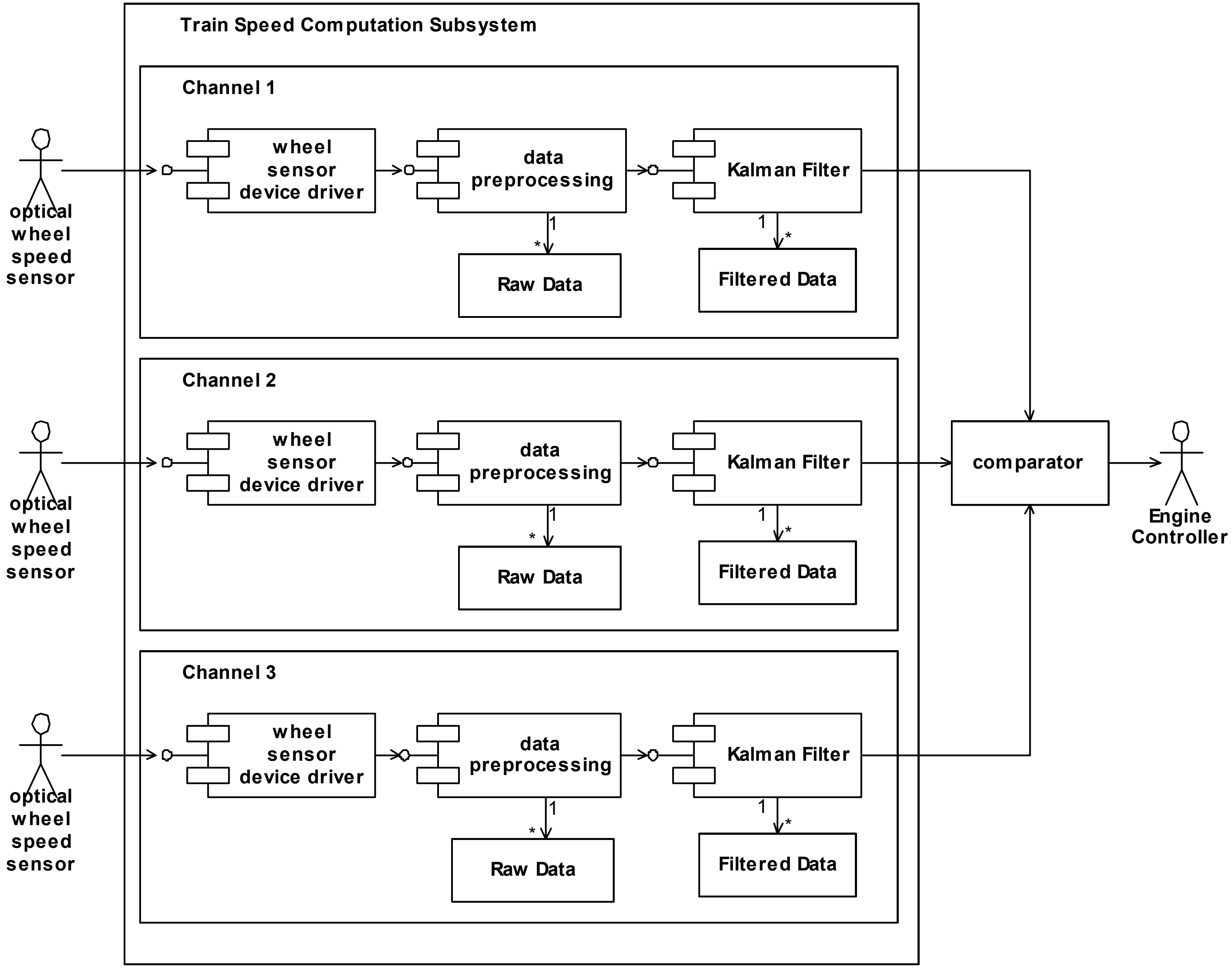
Triple Modular Redundancy Pattern

- Problem
  - Want to provide protection against single point failures and be able to continue
- Solution
  - Replicate the channel three times, run all three in parallel
  - If one channel breaks, then the other two will concur
- Consequences
  - A common solution to redundancy
  - Low design cost
  - Very high recurring cost
  - Good support for failures but not for errors

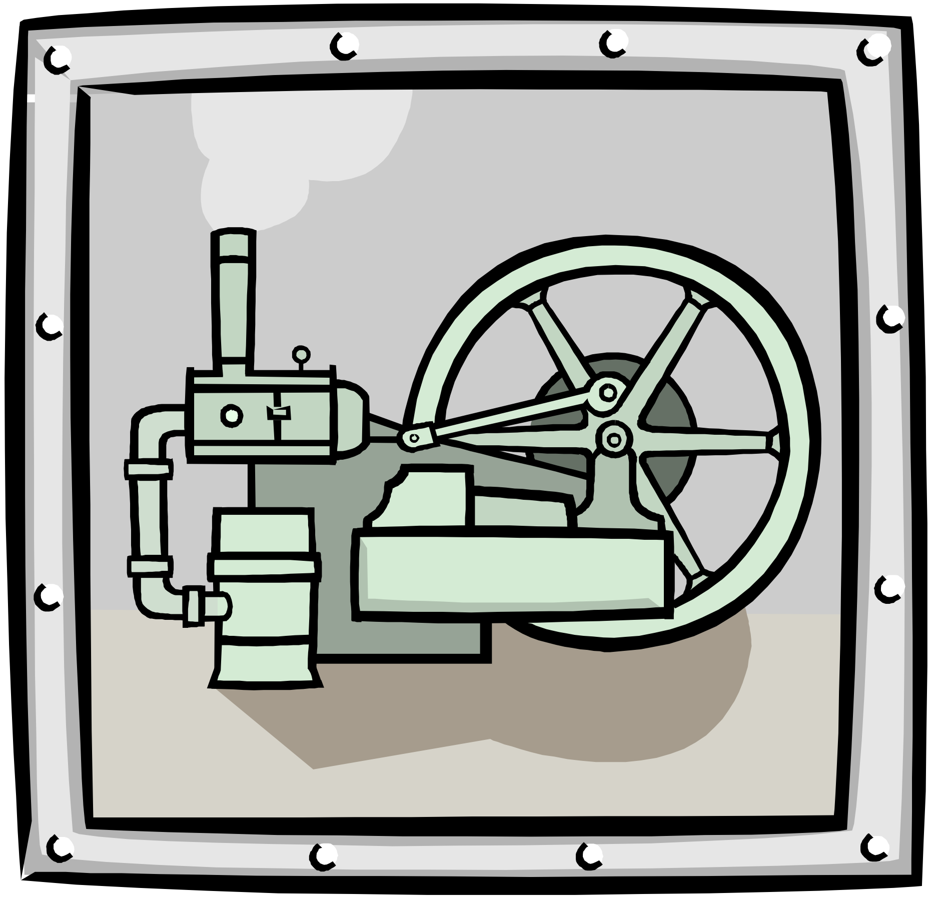
Triple Modular Redundancy Pattern



Triple Modular Redundancy Pattern Example



Mechanistic Design Patterns



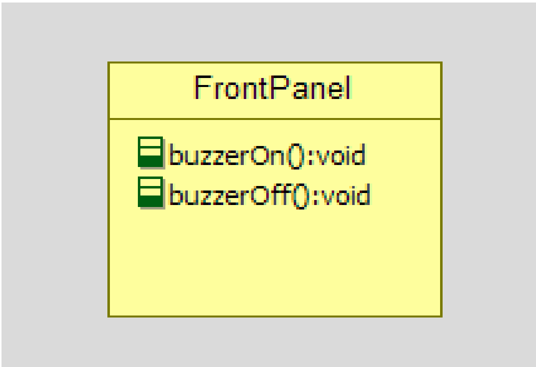
Fundamental Concepts

- Mechanistic patterns all start with “First, create an object that …”
- The vast majority of design patterns use a combination of two very fundamental concepts:
  - Delegation
  - Interface

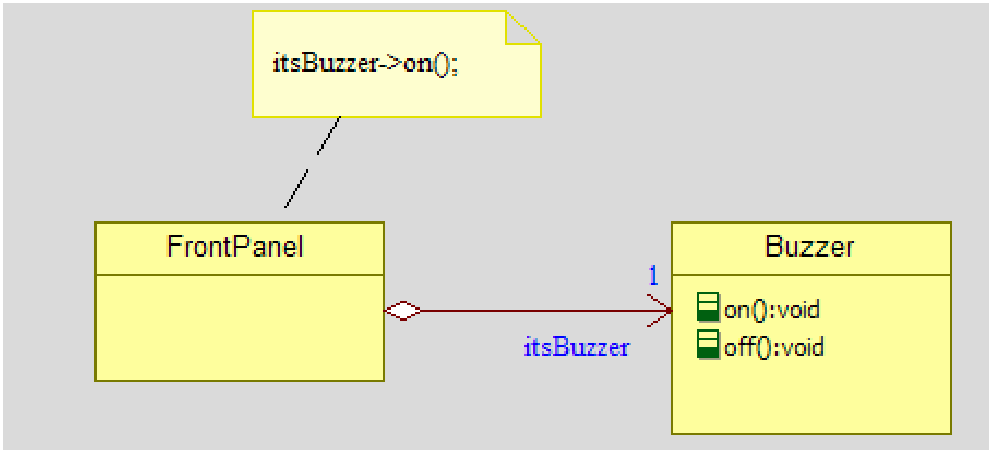
The Singleton design pattern is an exception to this rule.

Delegation

- Delegation is a way to extend the functionality of a class by using an extra class to provide additional functionality.
- It is not always necessary to use Inheritance to extend functionality, often Delegation is more preferable.



Without Delegation

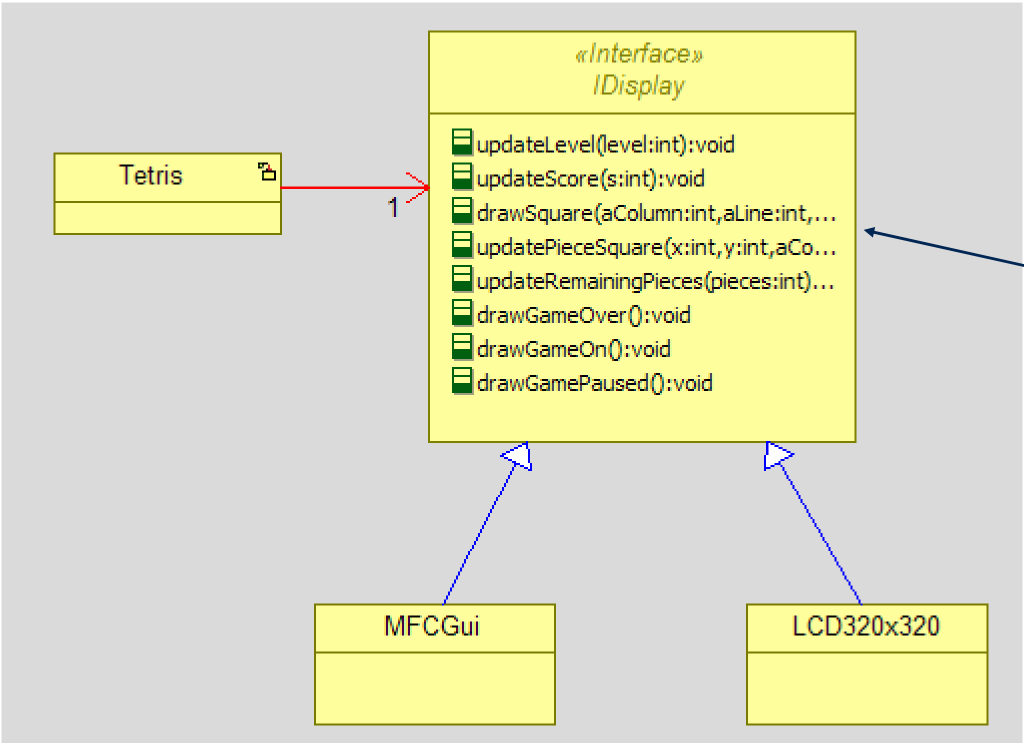


With Delegation

Telelogic

Interface

- This is really what most Design Patterns are based upon. The idea is to keep the things that change, separate from the things that don't change.



Pure Virtual operations

Telelogic

Categories of Mechanistic Design Patterns

- Creational
  - Control the instantiation or creation process
- Structural
  - Concerned with how classes and objects work together to form larger collaborations
- Behavioral
  - Concerned with algorithms and roles objects play within collaborations

The selection and categorization of patterns used here is based largely on the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma et. al, 1995

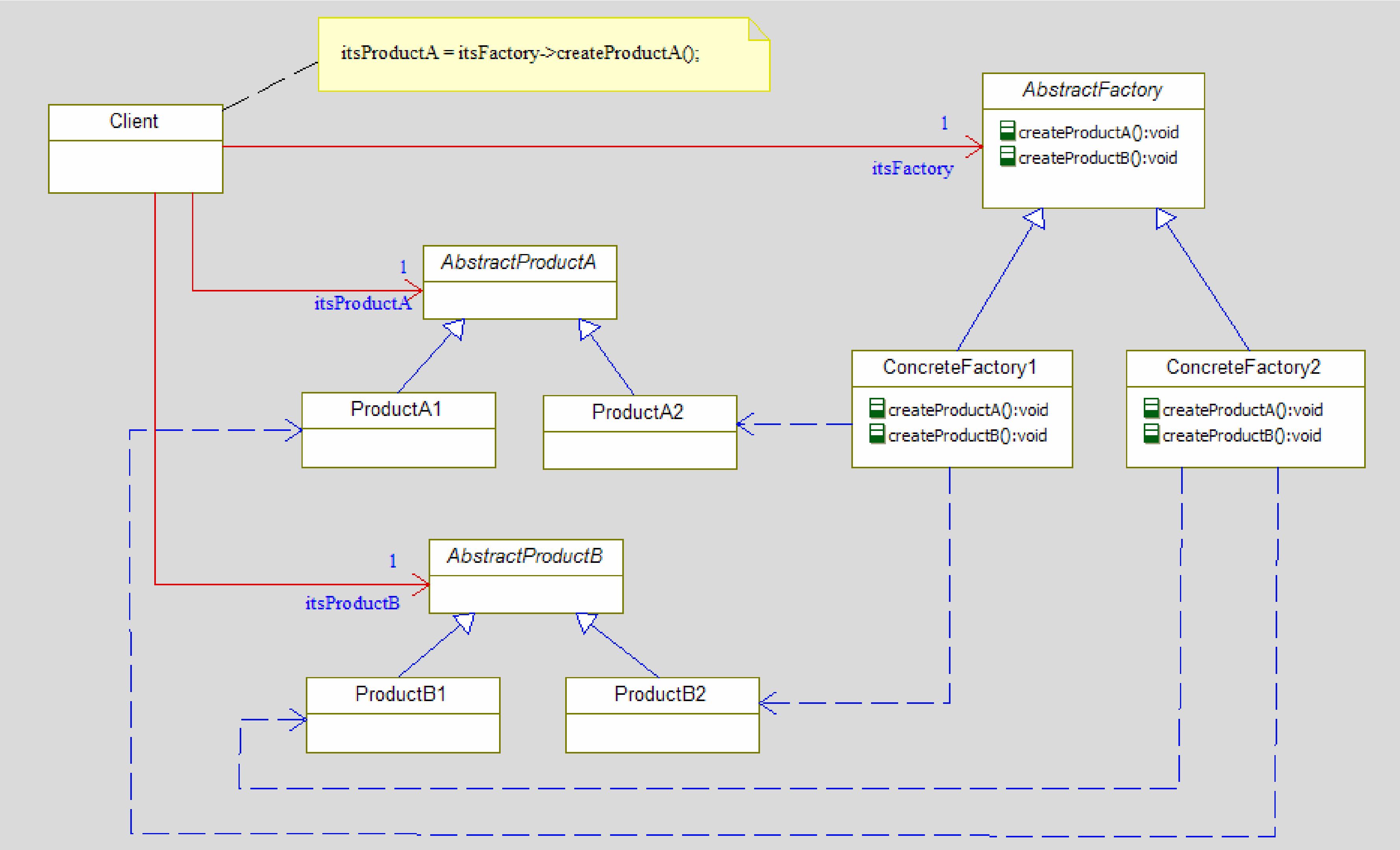
Creational Patterns

- Factory
- Factory Method
- Prototype
- Singleton

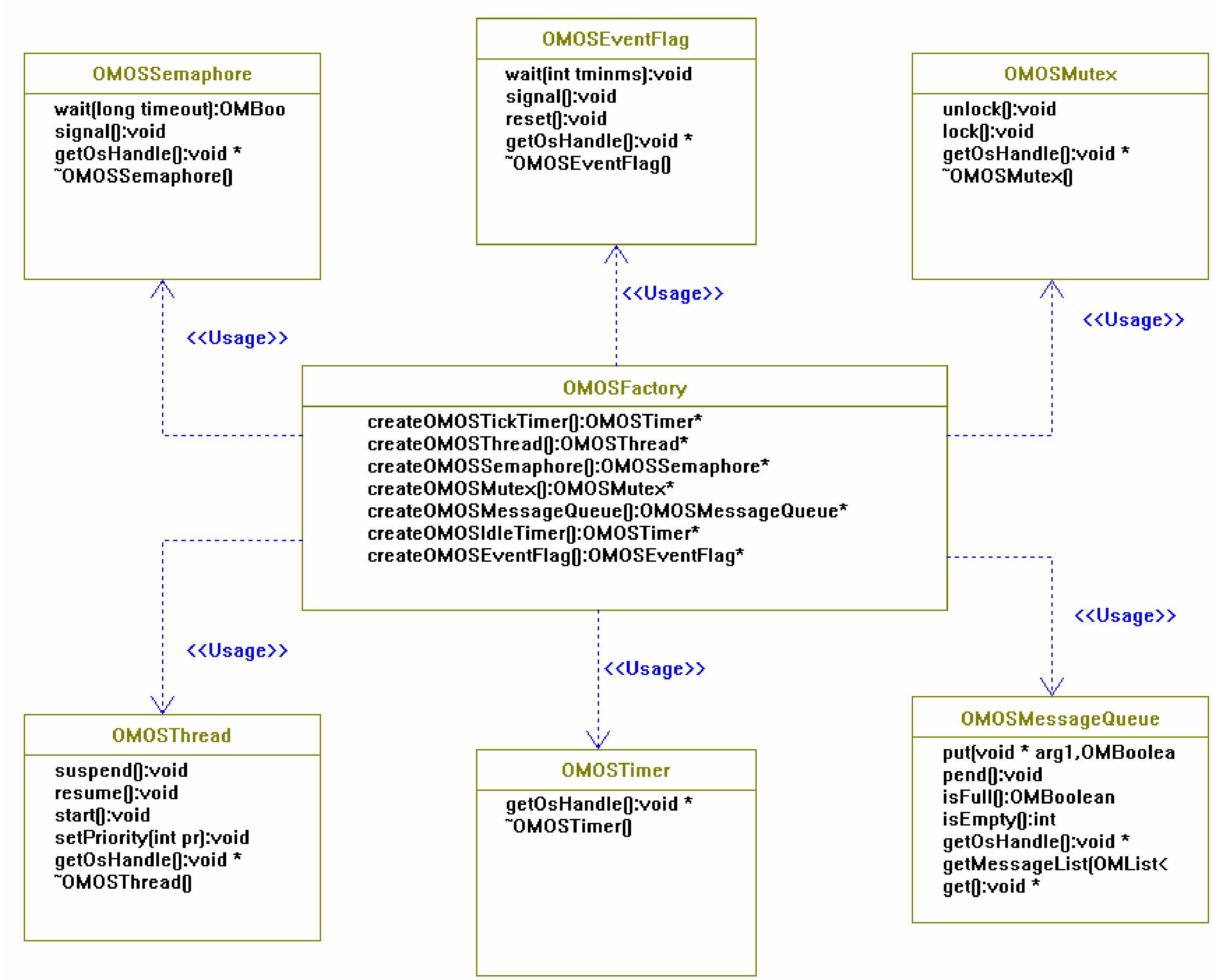
Factory Pattern

- Problem
  - Provide an interface for creating families of related or dependent objects
- Applicability
  - When you want to enforce independence of the system structure from how it is instantiated
  - When a system is to be run on different platforms parts that vary among the platforms
  - When a set of objects are meant to be used together
- Solution
  - Create an abstract factory that knows how to create all the related parts; subclasses of the factory know of to create the parts for a specific platform
- Consequences
  - It makes it easy to create platform-specific families of related classes
  - It promotes consistency among the product families
  - It aids understanding the common ground between families
  - It is easy to add new platforms for a set of objects

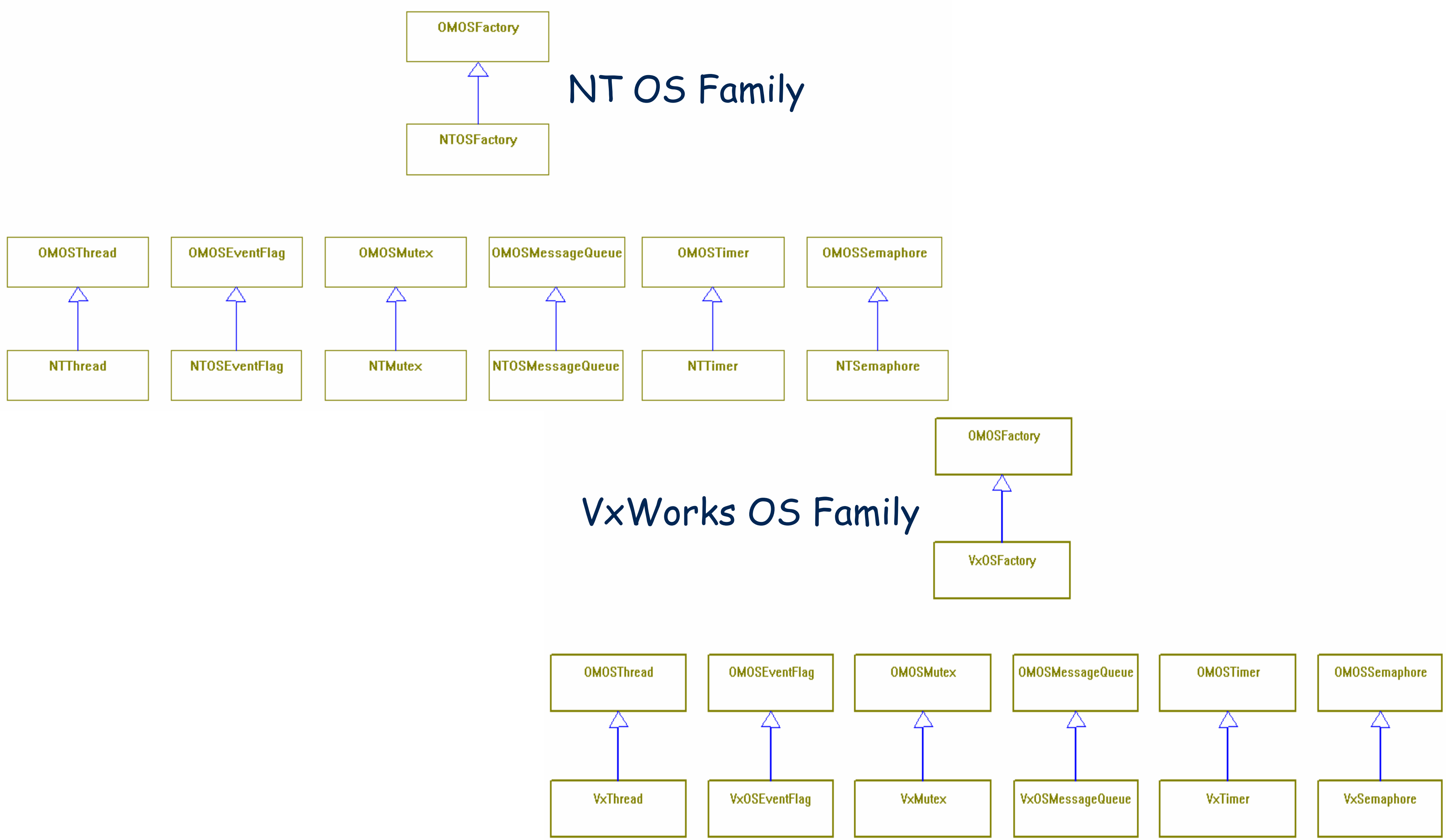
Factory Pattern



Factory Pattern Example



Factory Pattern Example



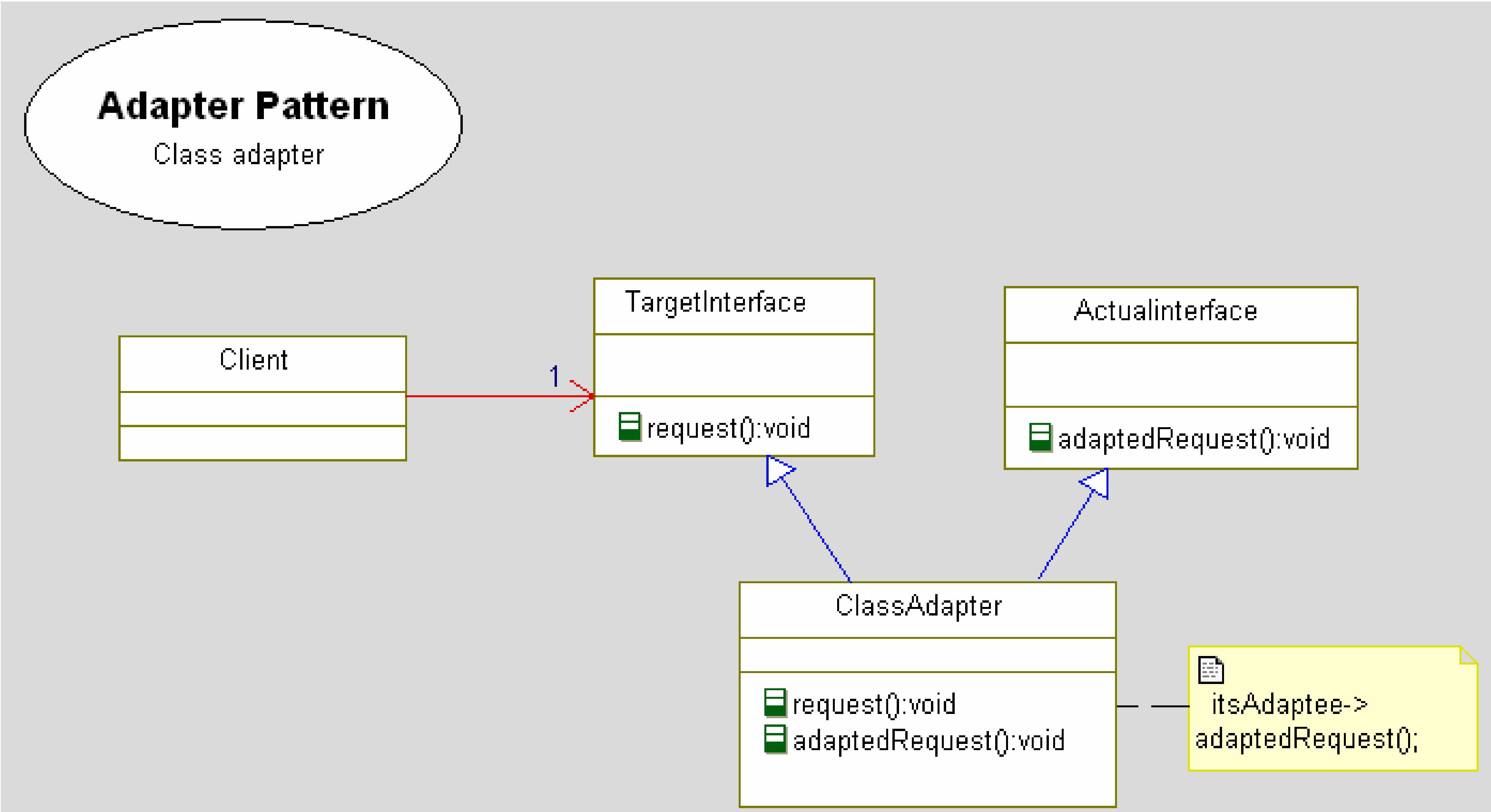
Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Flyweight

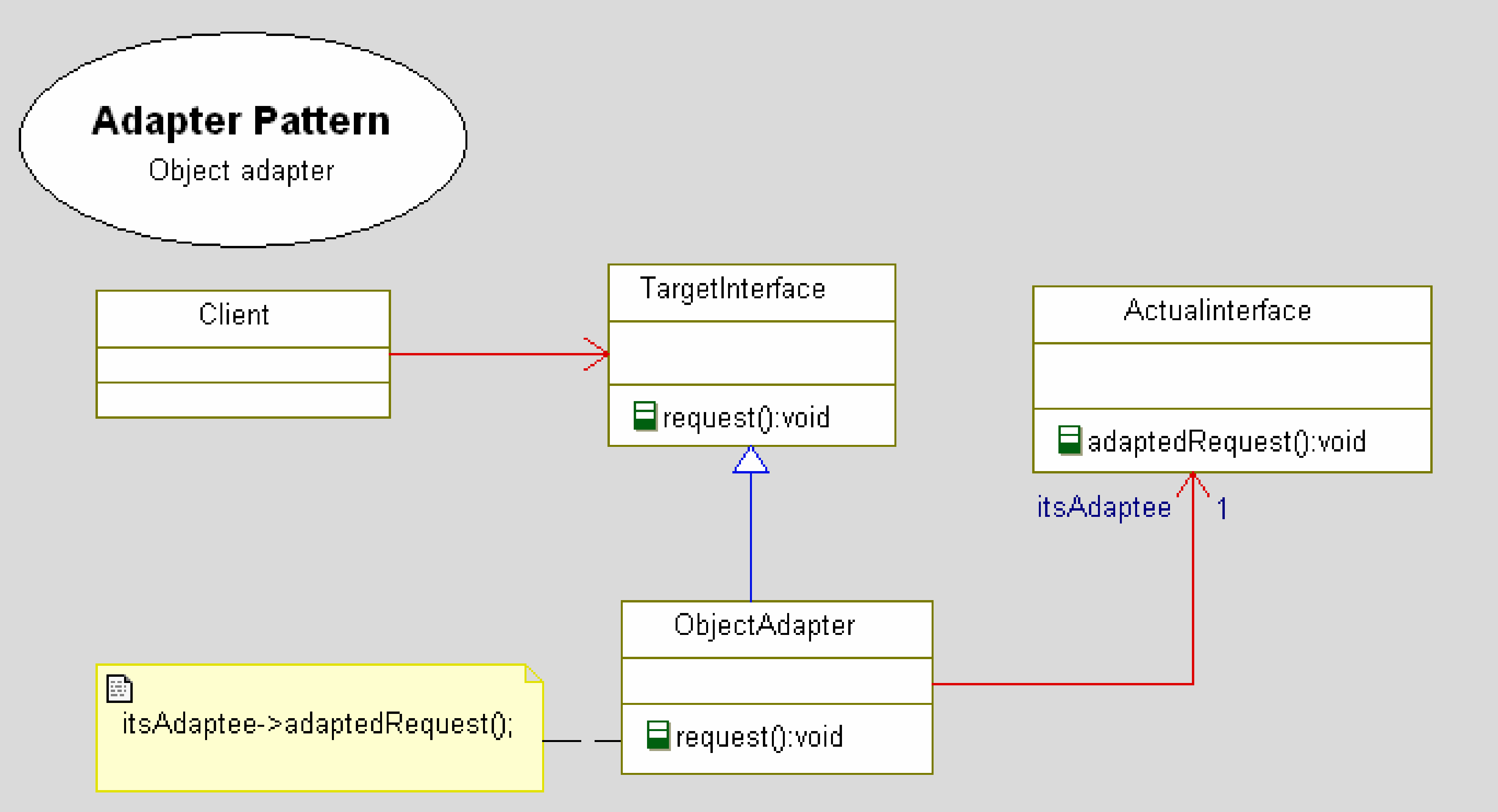
Adapter Pattern

- Problem
  - You want to adapt the interface of a class to meet a client need
- Applicability
  - (Class) When you have an existing class that meets the *semantic* need but has an incompatible interface
  - (Class) When you want to reuse an existing class in as-yet-determined circumstances
  - (Object)
- Solution
  - Create a class subclasses both the expected and actual interfaces and does the impedance matching, or
  - Create an object that refactors and forwards the requests
- Consequences
  - Class adapters work with only the specific class not subclasses
  - Object adapters work with class and subclasses
  - Class adapters introduce only a single object

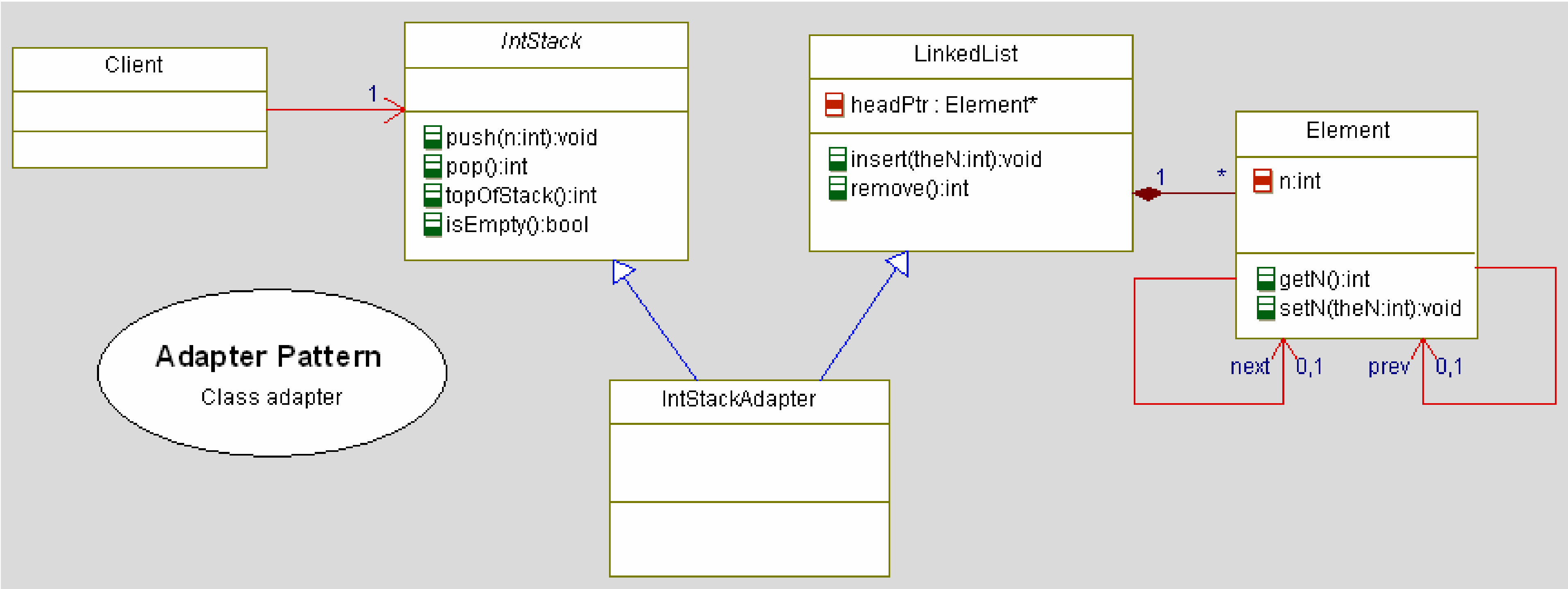
Adapter Pattern (Class)



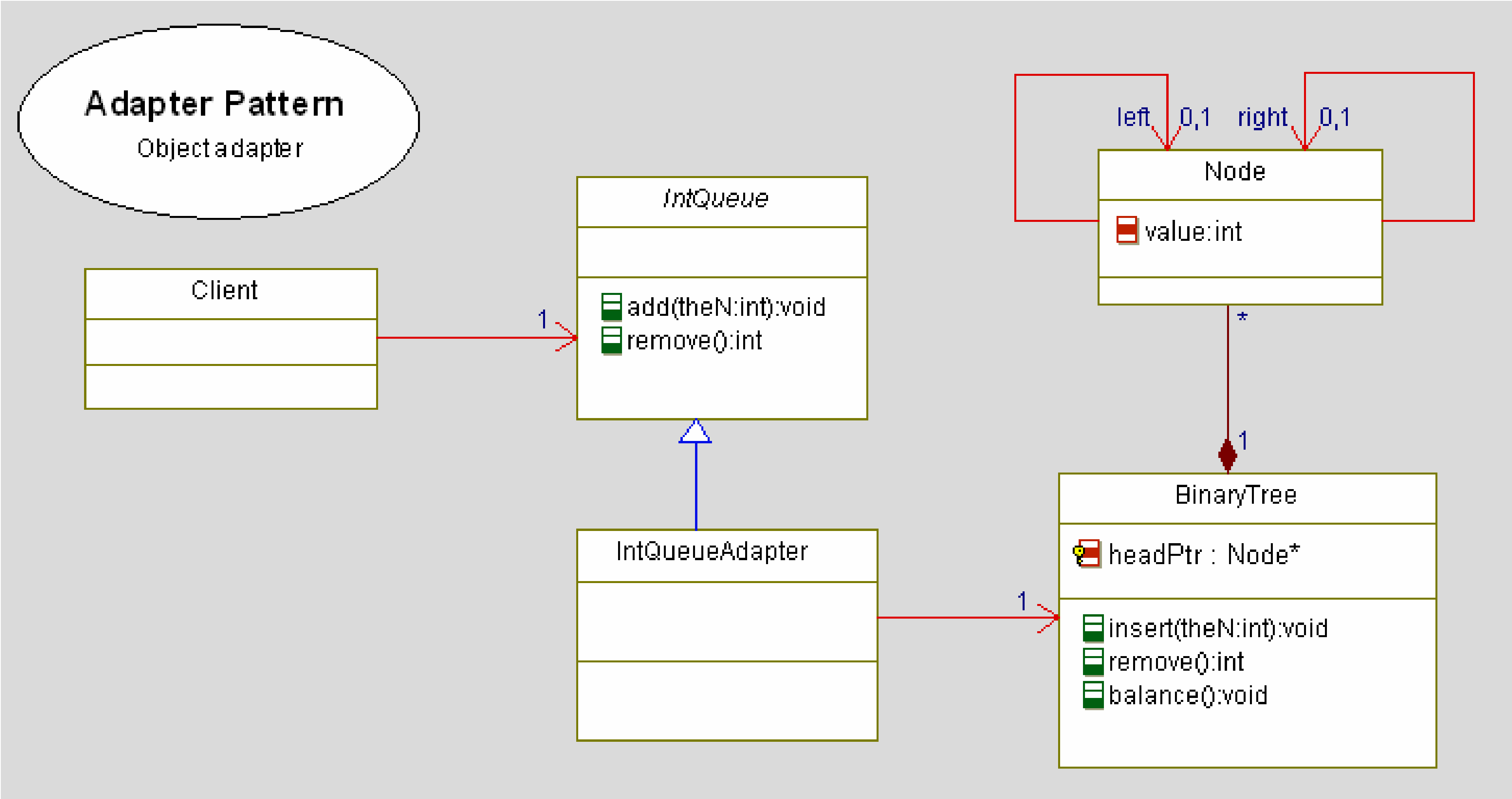
Adapter Pattern (Object)



Adapter Pattern Example (Class)



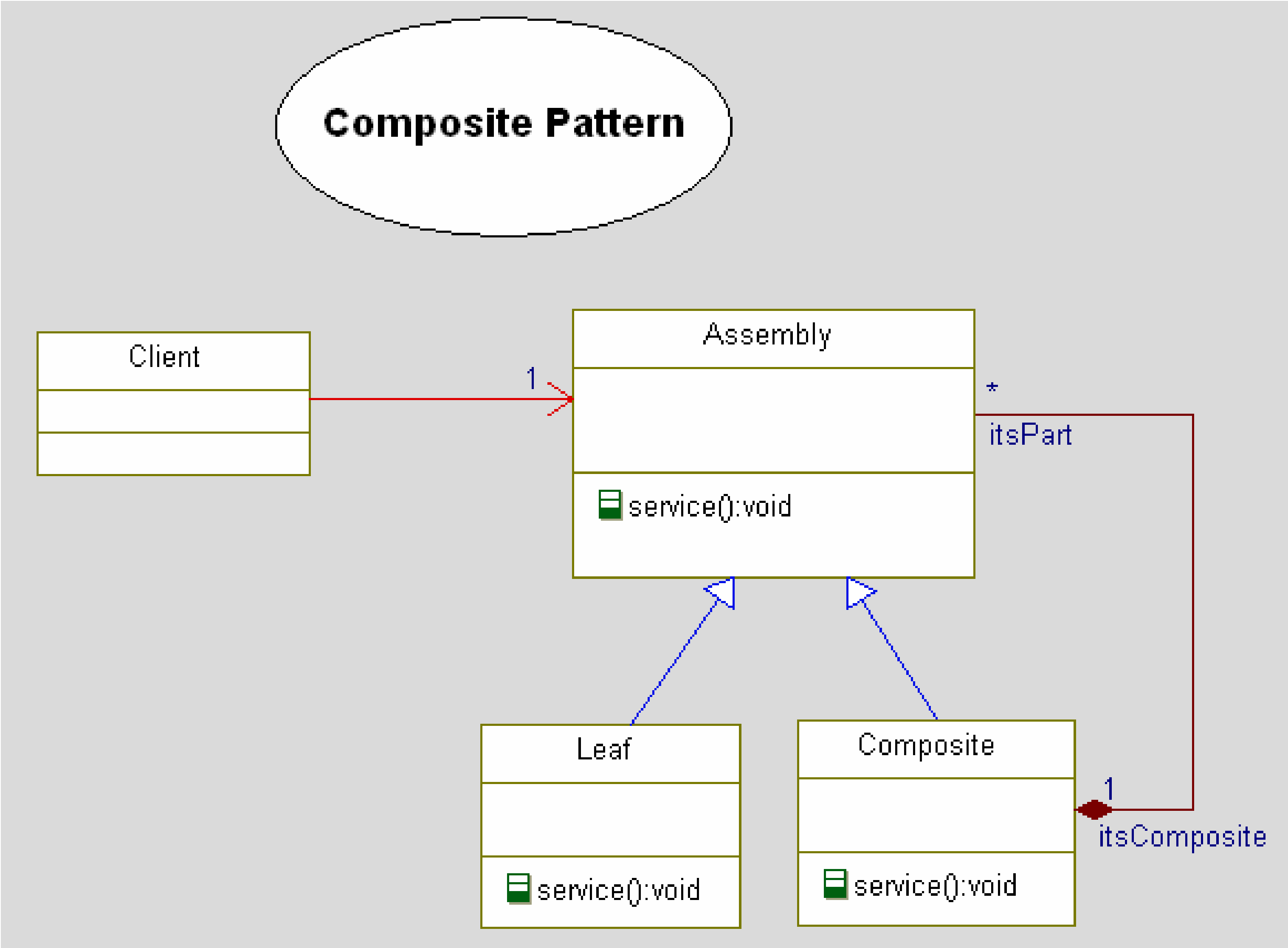
Adapter Pattern Example (Object)



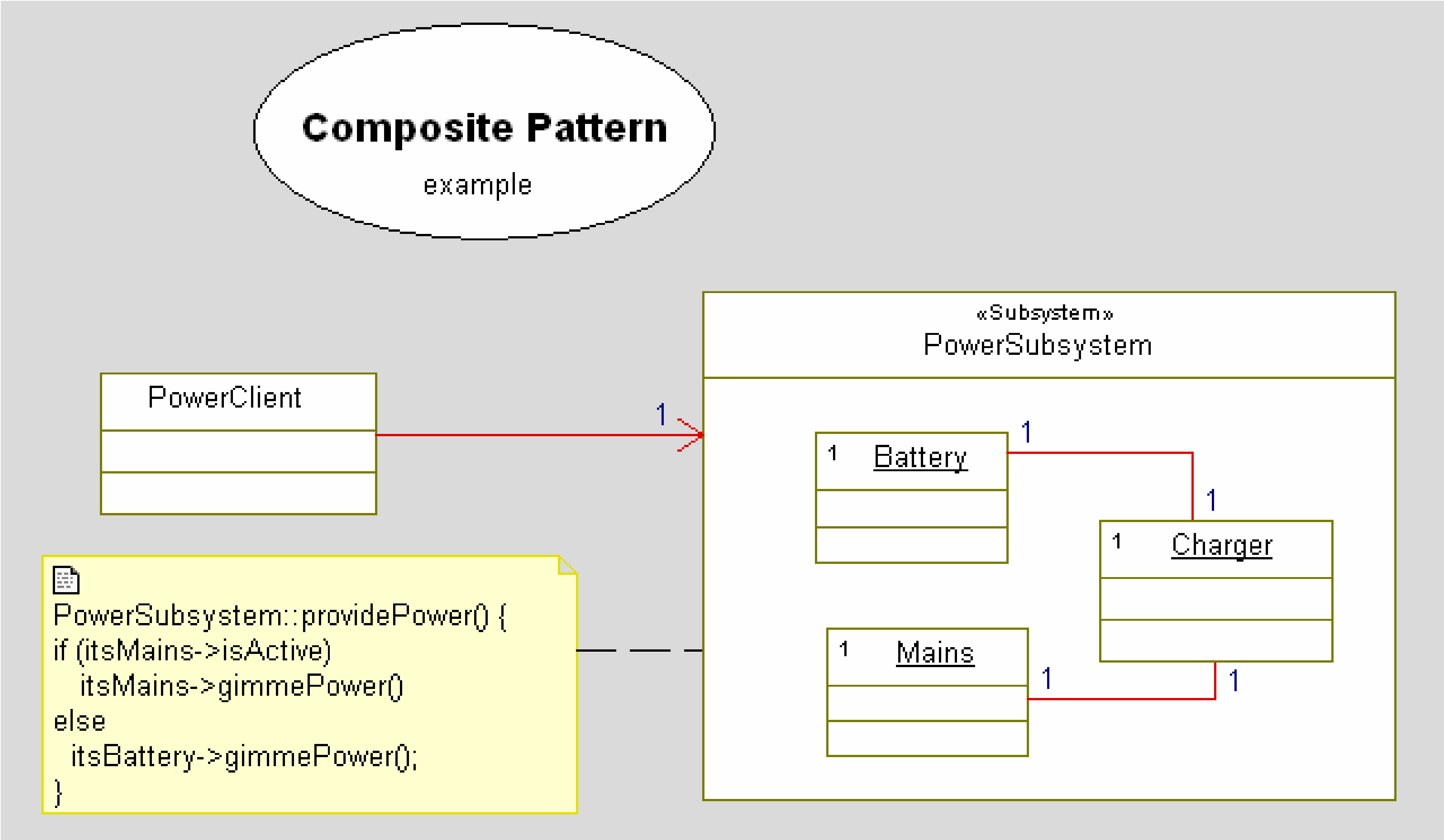
Composite Pattern

- Problem
  - Want to represent the grouping of smaller primitive semantic objects into larger assemblies
- Applicability
  - When you want to represent whole-part hierarchies
  - When you want to assign creation-destruction responsibilities
  - When the whole is at a higher level of abstraction than the parts
    - E.g. System contains subsystems contains sub-subsystems
- Solution
  - Use the Composite Class of UML 2.0/Rhapsody
- Consequences
  - Very effective way to represent hierarchies at possibly many levels
  - Very straightforward way of assigning create-destroy responsibilities
  - Specific creation-destruction sequences must be added to the structured class behaviors, if needed
  - Composite is responsible, in general, to orchestrate its parts to deliver services
  - The Port Pattern provides explicit delegation w/o Composite delegation

Composite Pattern



Composite Pattern Example



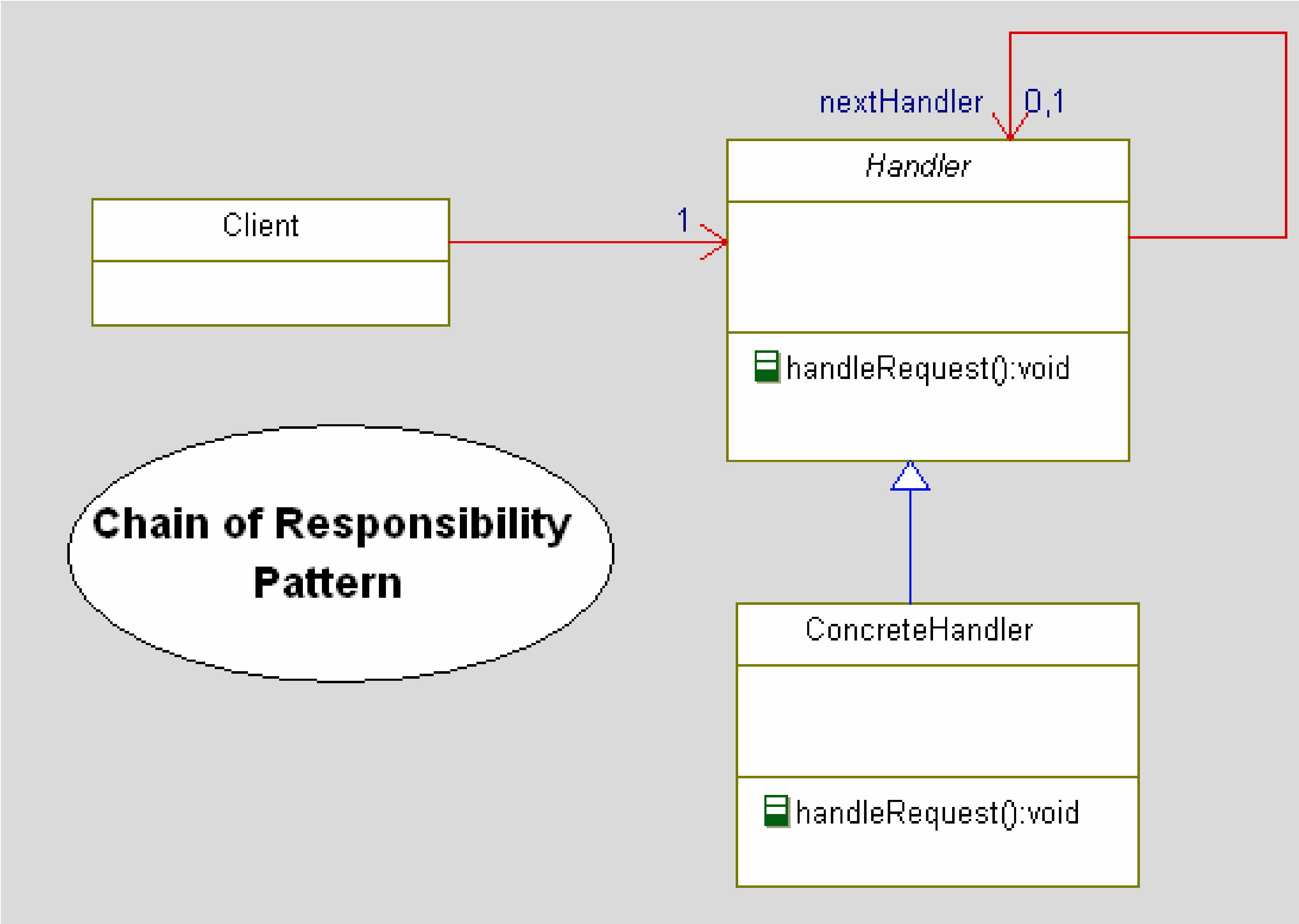
Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Container-Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Specializable ("Template") Method

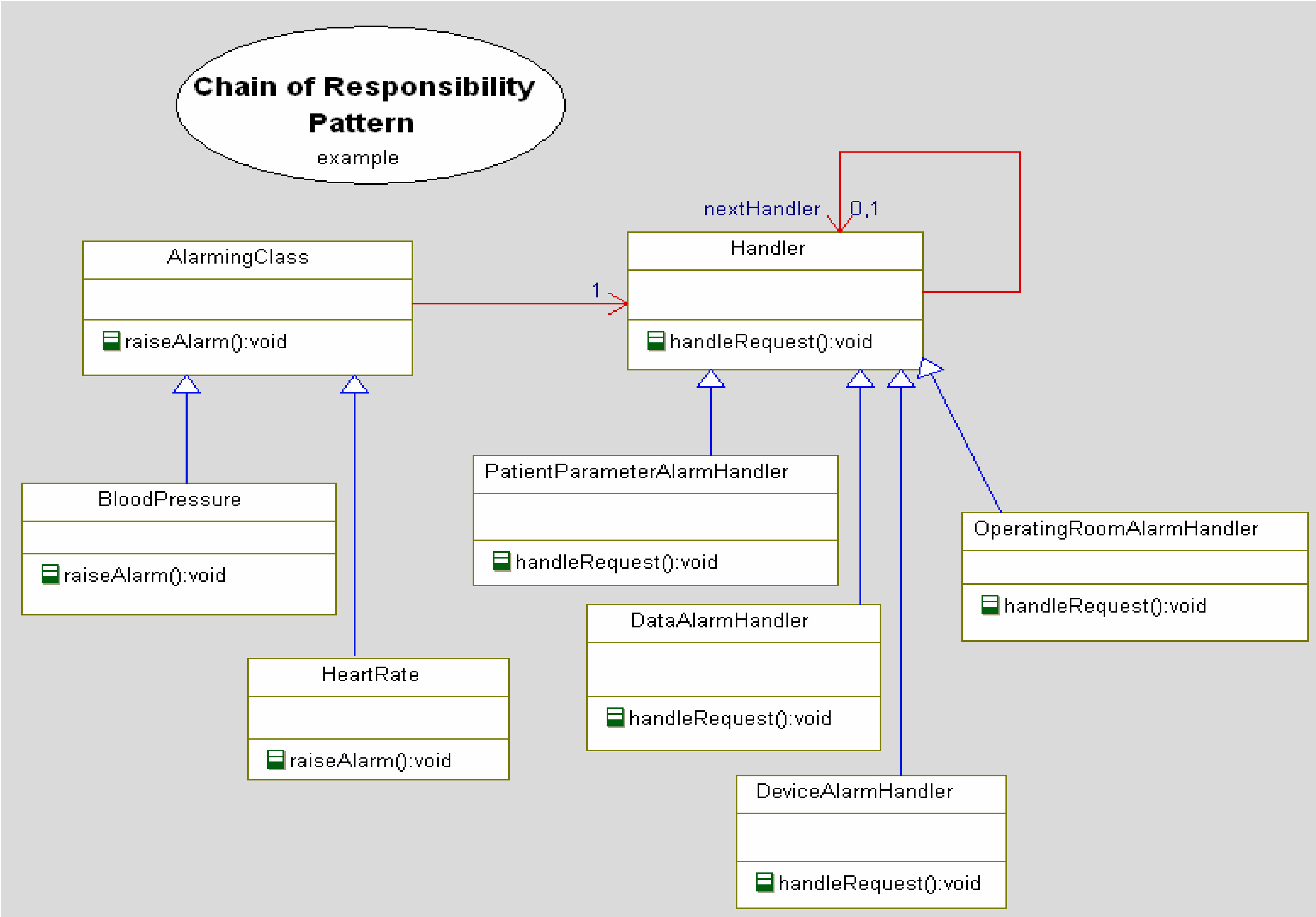
Chain of Responsibility Pattern

- Problem
  - You want to avoid coupling of a request to a specific receiver by giving more than one object a chance to fulfill the request
- Applicability
  - When the best place to handle a request isn't always known at design time
  - When you want to have multi-level response
  - When you want the request handlers to be specified dynamically
- Solution
  - Create a chain of objects and pass the request from object-to-object until the request is fulfilled or until all objects have had a chance to act
- Consequences
  - Reduced design-time coupling of classes
  - Ability to add request processing dynamically
  - There is no guarantee that a request will be handled because there is no grand oversight

Chain of Responsibility Pattern



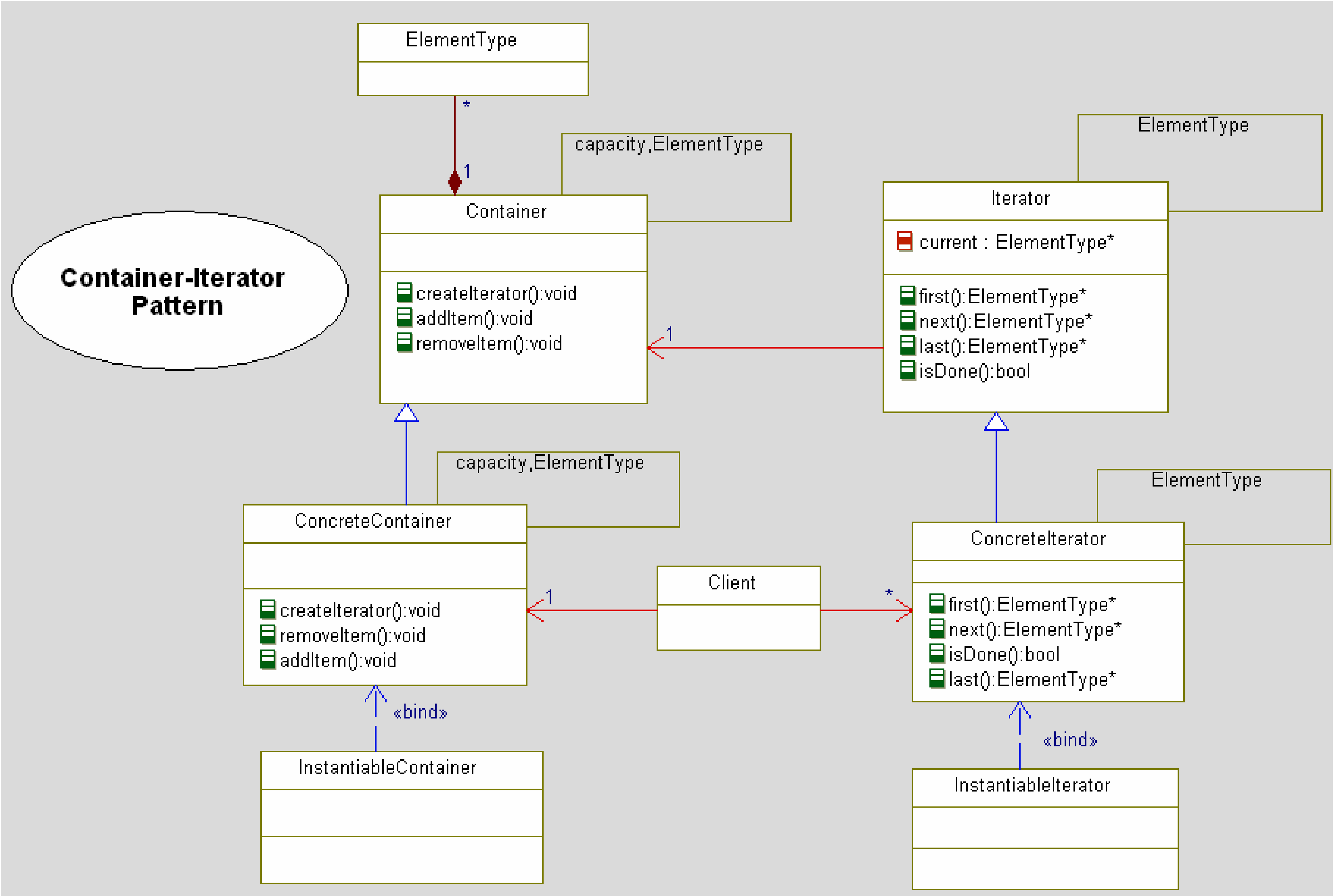
Chain of Responsibility Pattern Example



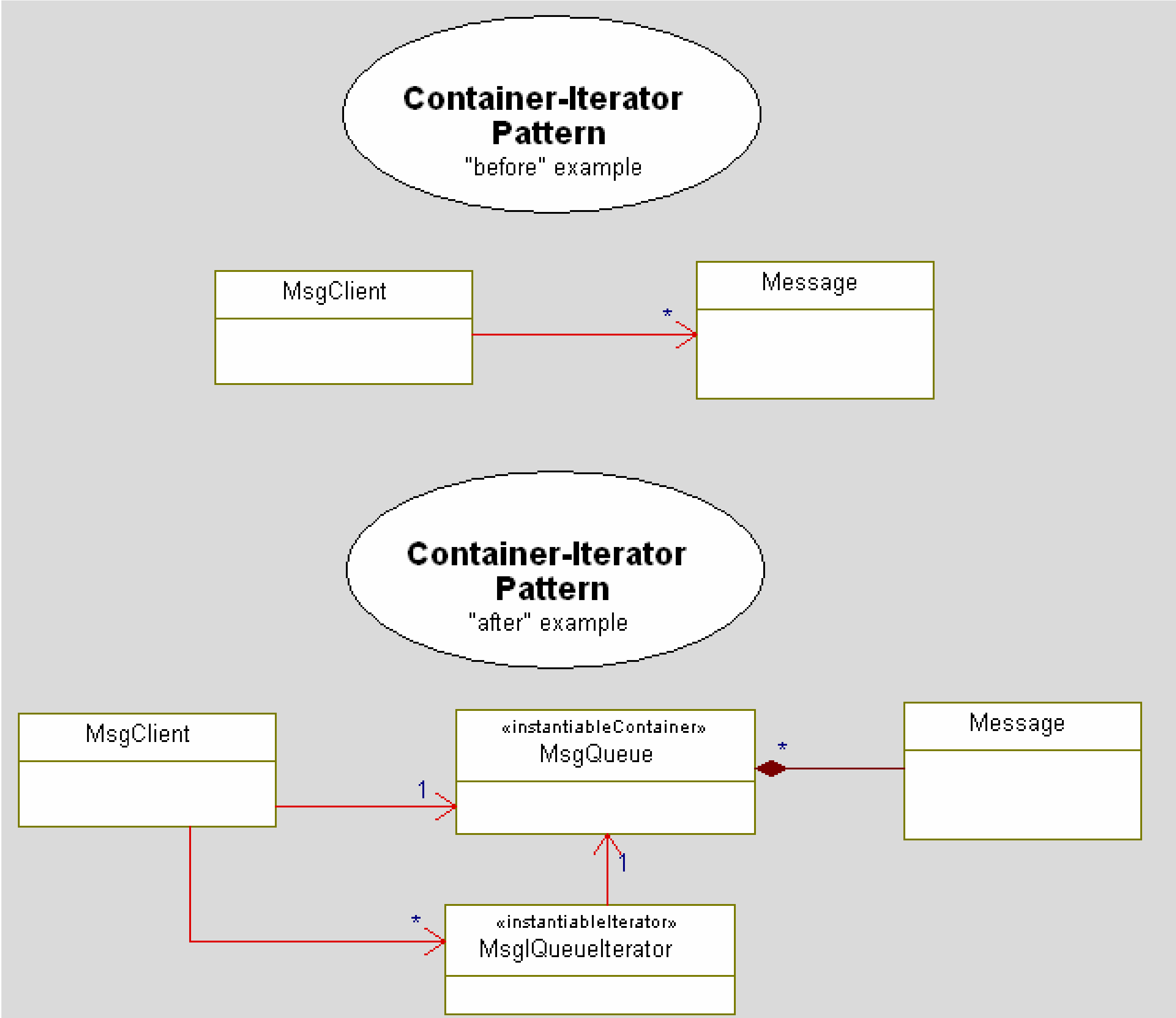
Container-Iterator Pattern

- Problem
  - You want to efficiently manage collections and hide the implementation of the collection
- Applicability
  - When containment is independent of object semantics
  - When you want to support multiple clients of the same collection
  - When you want to supply a consistent interface for collection navigation
- Solution
  - Reify the collection as an object and provide implementation-free interface for the kind of collection
  - Usually implemented as a parameterized class (template)
  - Add iterators to support multiple-client navigation over the collection
- Consequences
  - Allows extensibility in navigation of collections
  - Allows easy changes to types of collections
  - Iterators simplify the navigation over the collections
  - Multiple clients can simultaneously navigate over the collection
  - *Note: Rhapsody provides this pattern automatically*

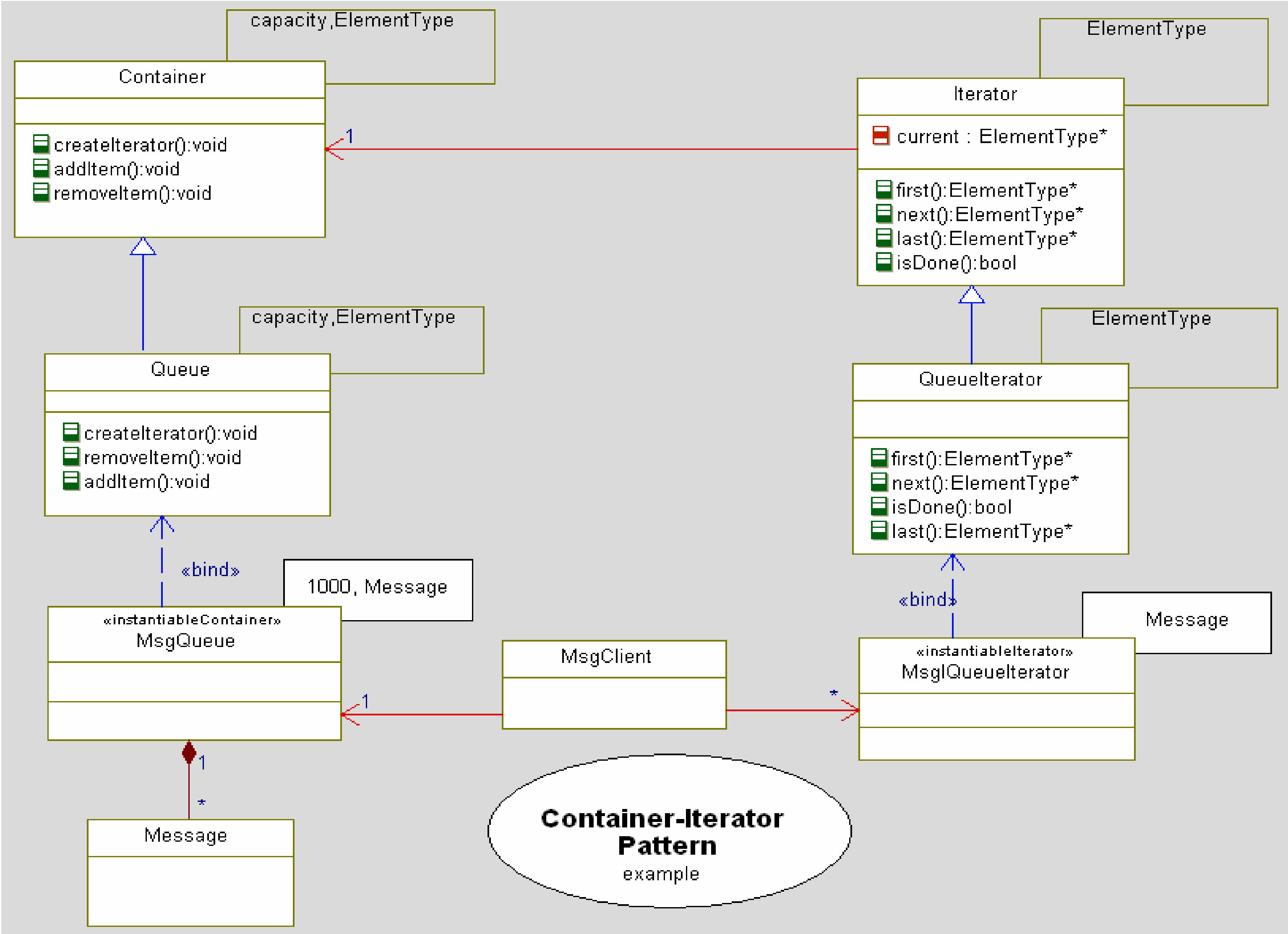
Container-Iterator Pattern



Container-Iterator Pattern Example



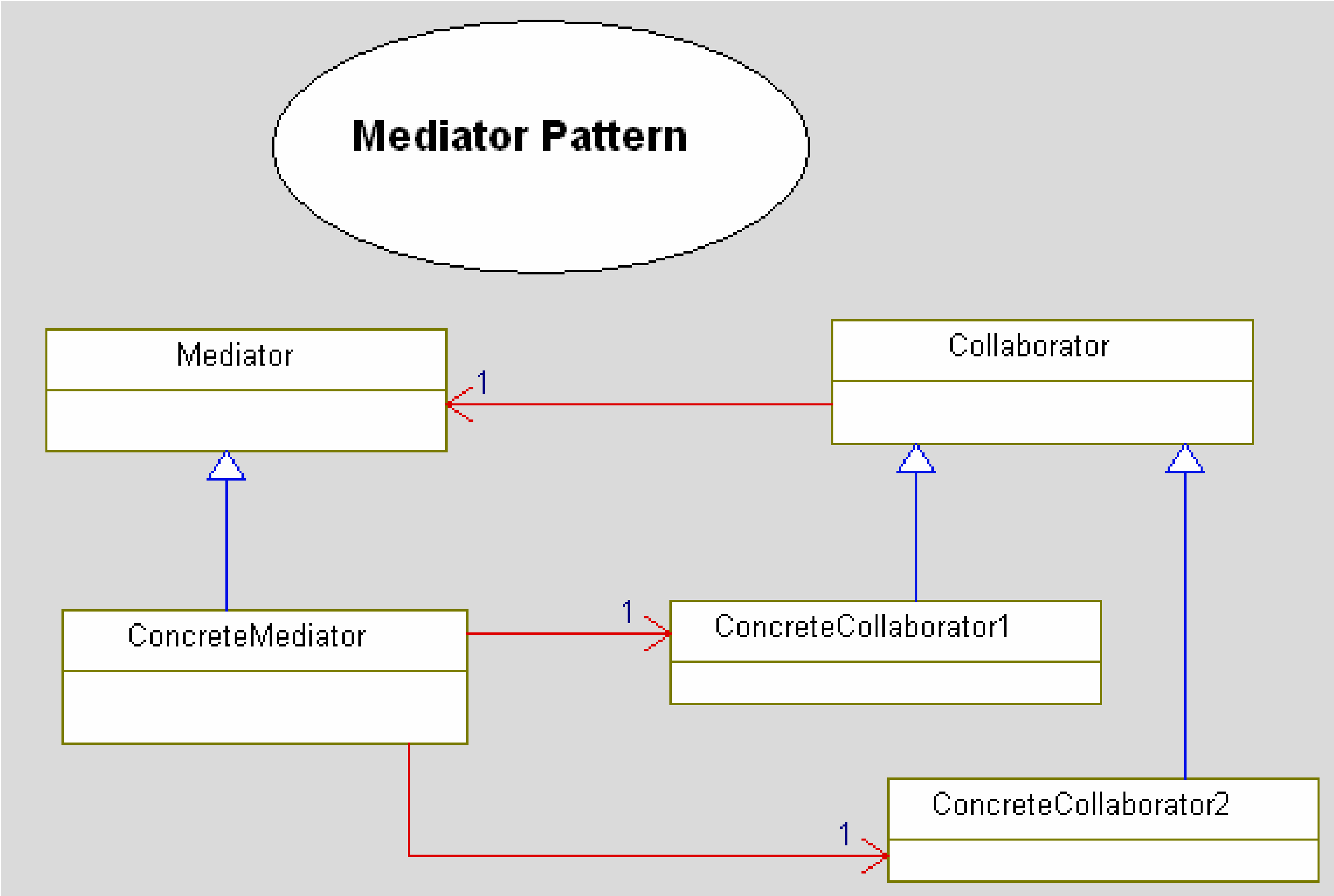
Container-Iterator Pattern Example



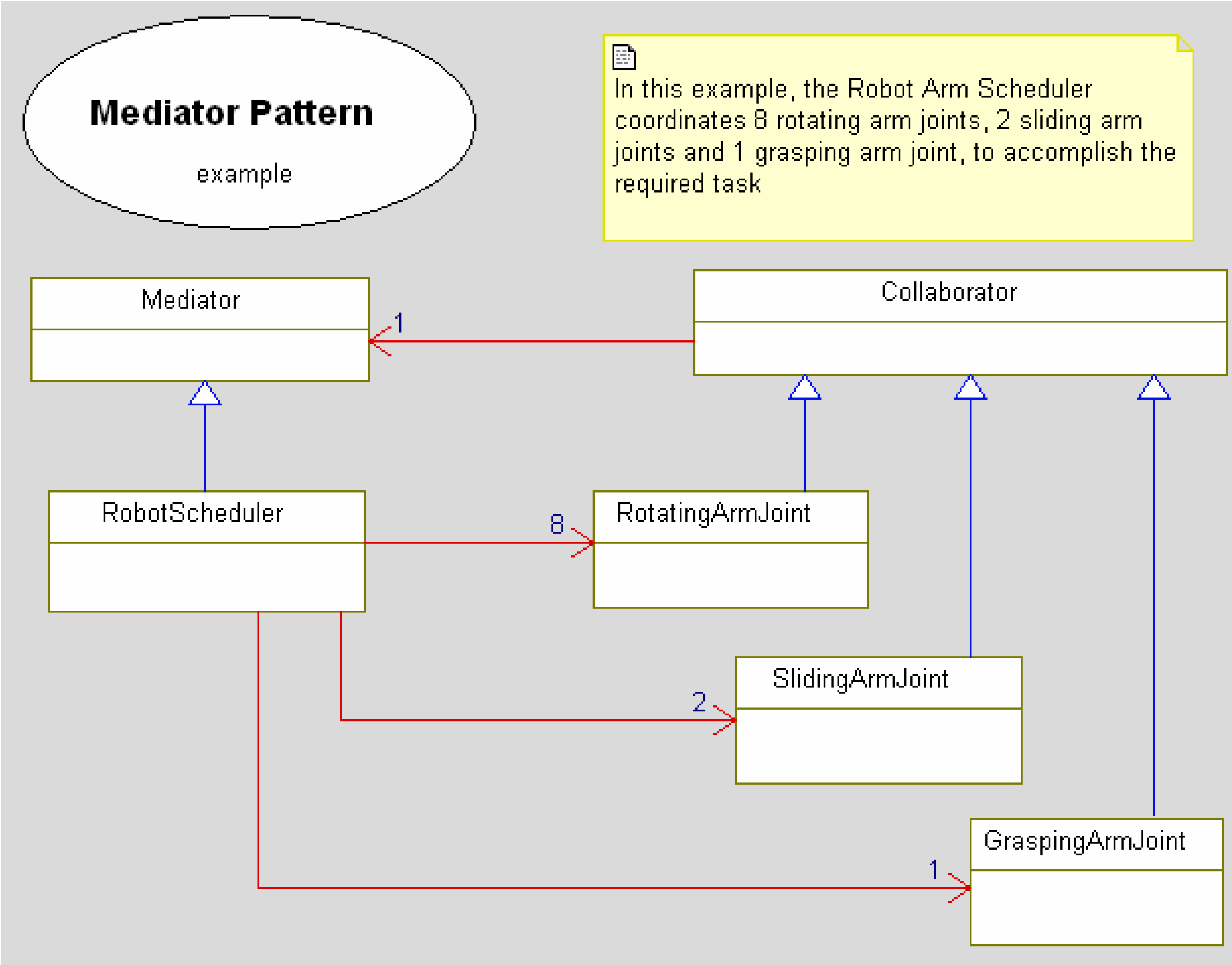
Mediator Pattern

- Problem
  - You want to coordinate a complex interaction among a set of objects
- Applicability
  - When a set of objects must coordinate in a well-defined but complex manner
  - When reusing an object is difficult because it has interactions with a large number other objects
  - When interactive behavior arising from a group of objects should be done without a lot of subclassing
- Solution
  - Reify the interaction as a class that coordinates the interaction with the objects
  - The architectural Rendezvous Pattern is a large scale example of this pattern applied to thread interaction
- Consequences
  - Limits subclassing
  - Decouples classes within the interacting set
  - Abstracts and simplifies object interaction protocols

Mediator Pattern



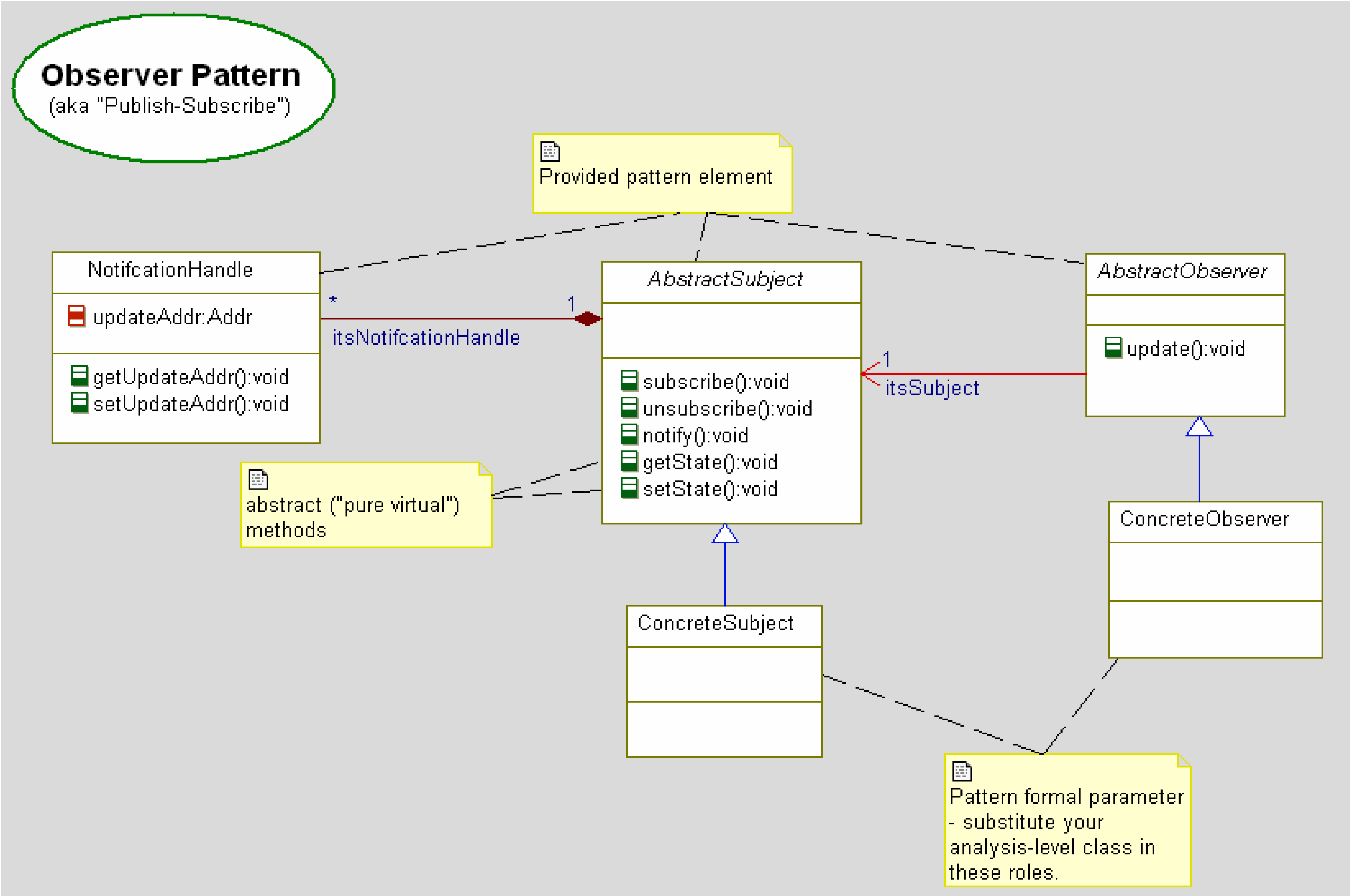
Mediator Pattern Example



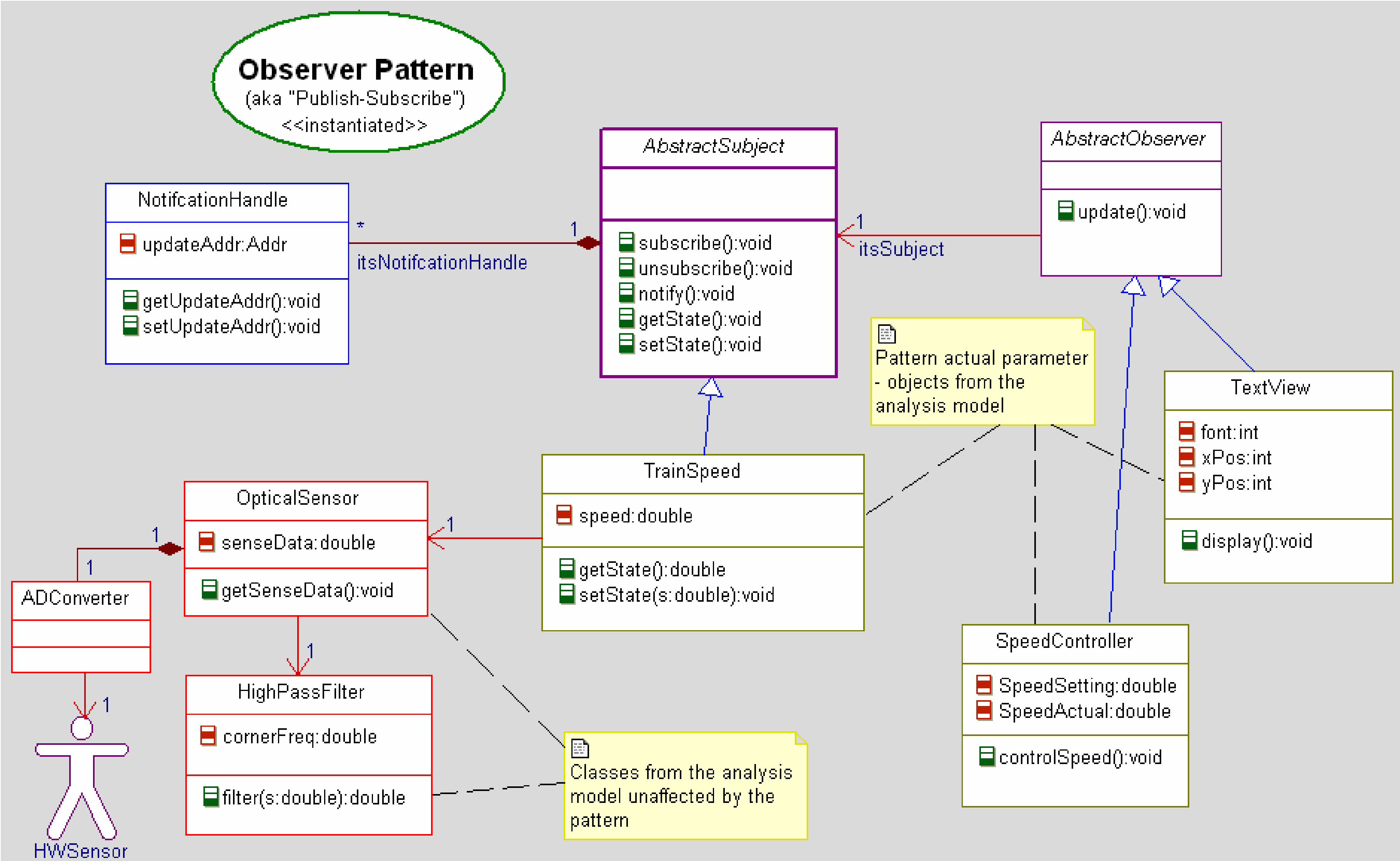
Observer Pattern

- Problem
  - You want to efficiently notify a set of clients that relevant data has changed
- Applicability
  - When you want to efficiently notify a set of objects about a value or state change
  - When you want to dynamically add or remove objects to be notified
  - When the server should have no knowledge of the client(s)
- Solution
  - Create a Subject class that provides subscribe() and unsubscribe operations
  - When data changes, notify all subscribed objects
- Consequences
  - Easy to dynamically add or remove observers
  - Supports “broadcast” of state information

Observer Pattern



Observer Pattern Example

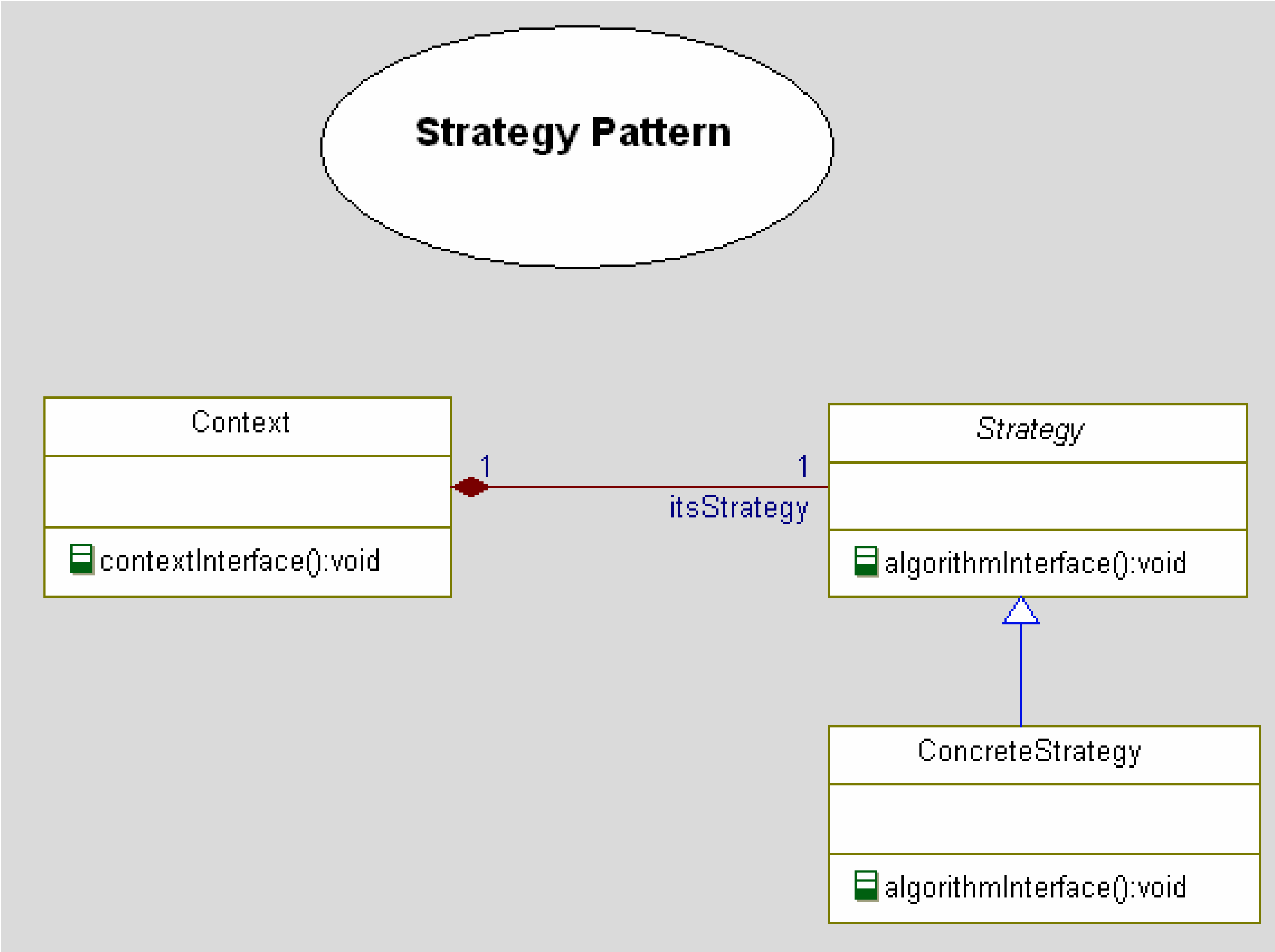


Telelogic

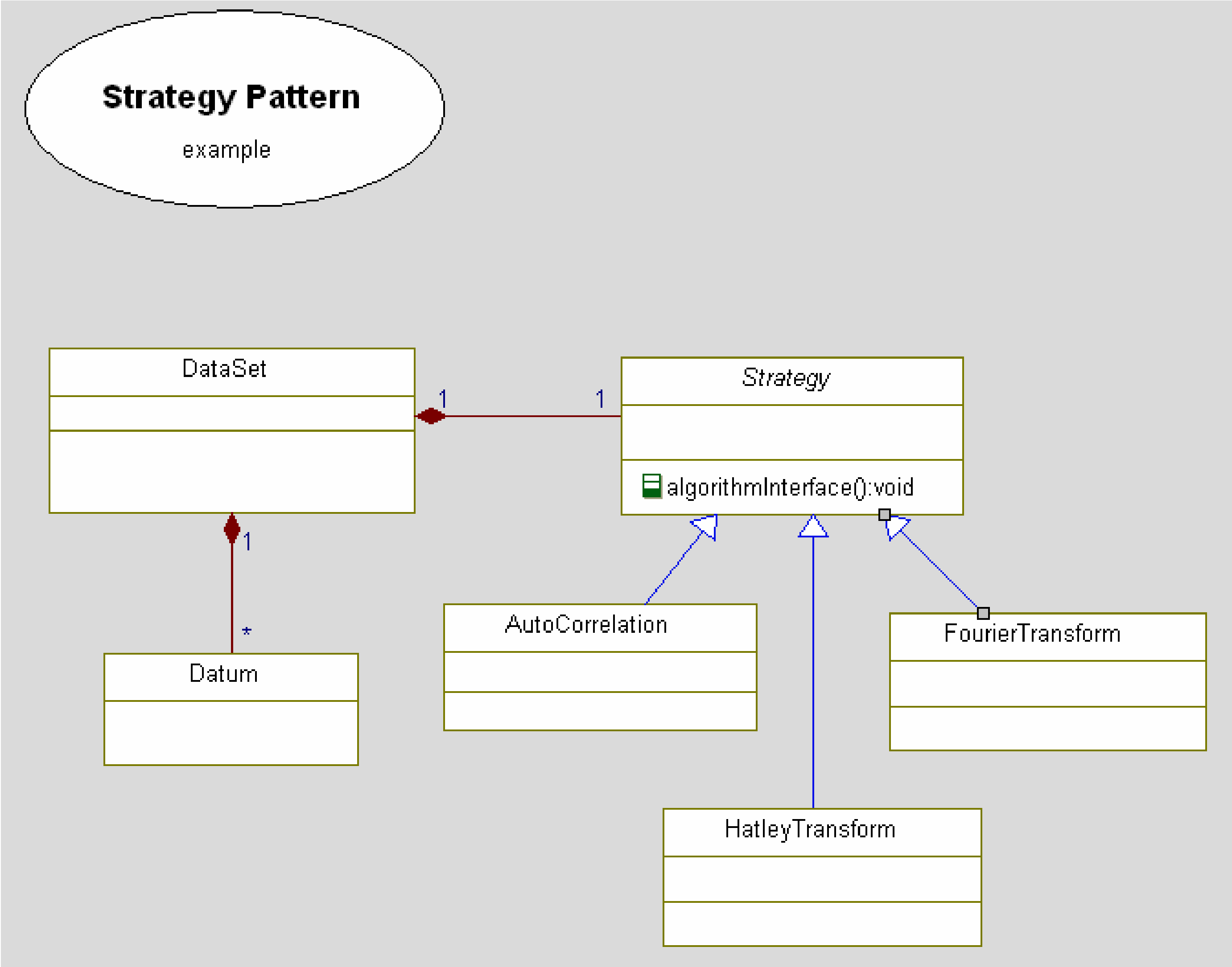
Strategy Pattern

- Problem
  - You want to apply a family of algorithms and make them interchangeable, even dynamically so
- Applicability
  - When you want to apply an algorithm to different classes
  - When you need algorithmic variants
  - When you need to vary a strategy or algorithm at run-time
- Solution
  - Reify the strategy(ies) as objects and attach them to the appropriate context objects
- Consequences
  - Easy management and application of related algorithms
  - Alternative to subclassing the context
  - Eliminates conditional statements in algorithms
  - Allows selection of an *optimal* algorithmic strategy depending on circumstances
  - Some overhead for collaboration between context and strategy

Strategy Pattern



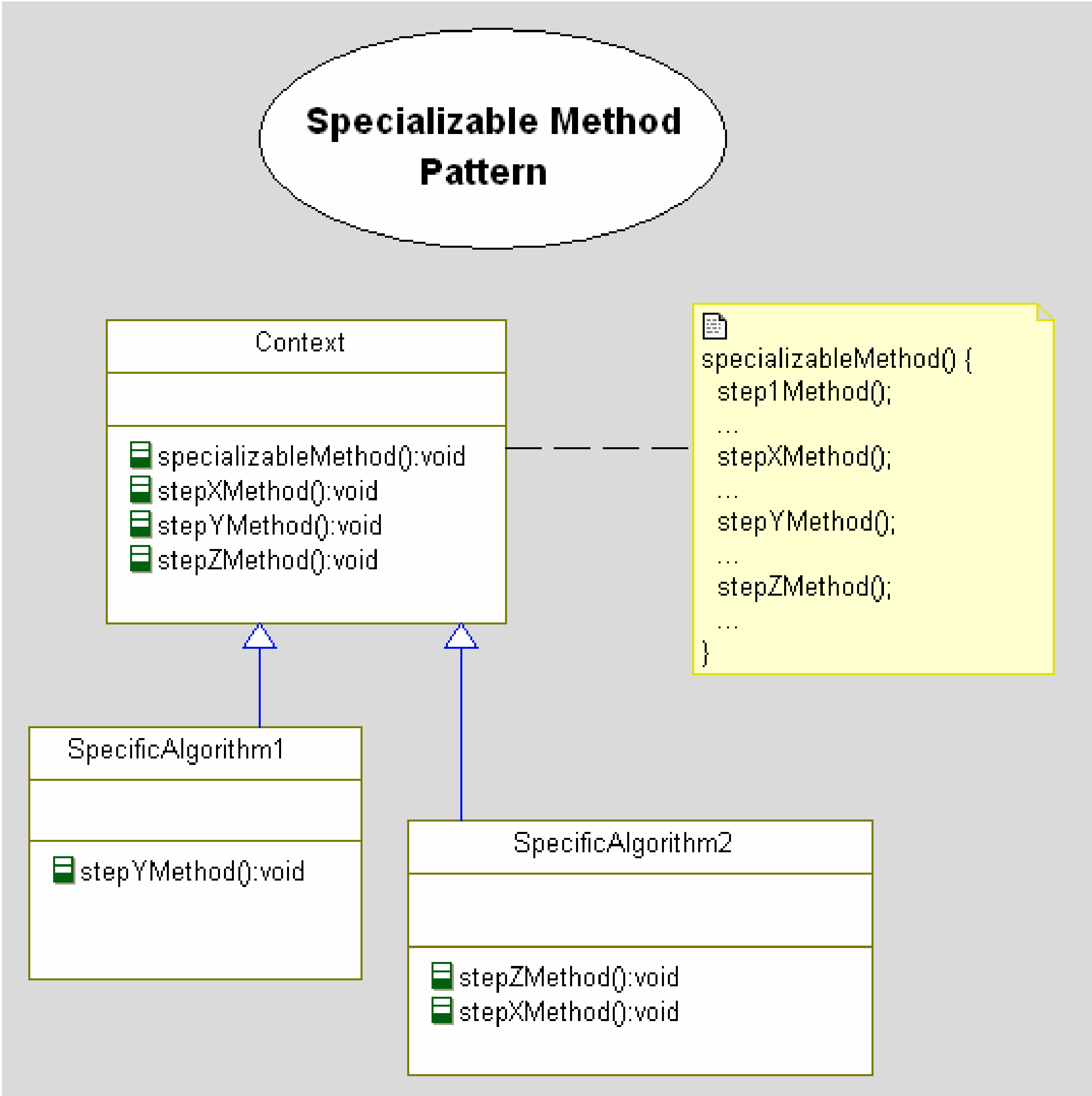
Strategy Pattern Example



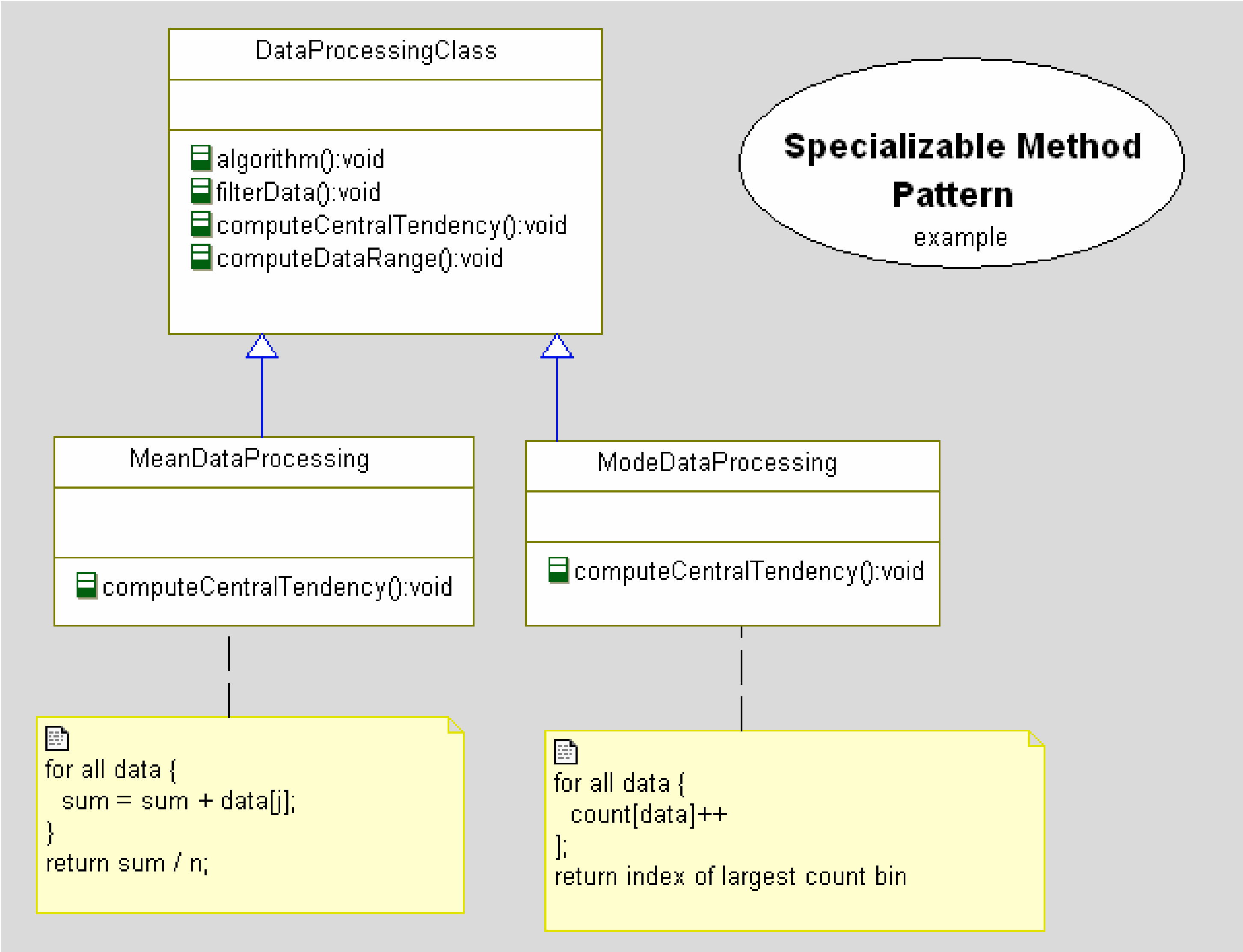
Specializable Method Pattern

- Problem
  - You want to define a skeleton of an algorithm and allow specialized steps to be subclassed
- Applicability
  - When the algorithm has a static structure but some steps may differ depending on the need
  - When the algorithm is linearly factorable
- Solution
  - Define a context class for the algorithm structure and allow subclasses to modify the steps that may vary
  - Aka “Template Method”
- Consequences
  - Good algorithmic reuse
  - Useful for class libraries

Specializable Method Pattern



Specializable Pattern Example



Real-Time Pattern References

White papers on real-time, objects, and the UML at www.ilogix.com