



Real-Time Architectural Design Patterns

*Bruce Powel Douglass, PhD
Chief Evangelist
Telelogic, Inc.*

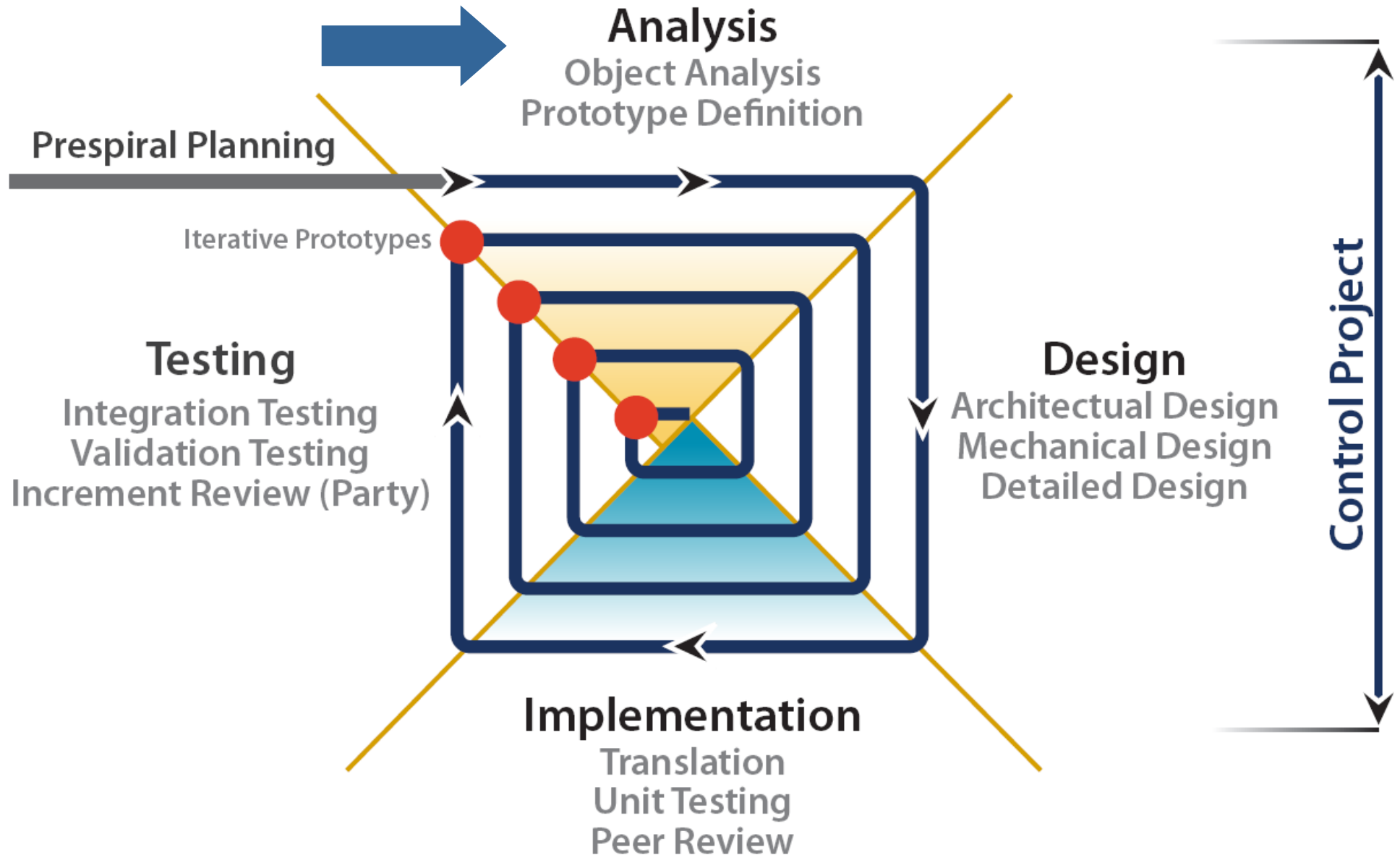
www.telelogic.com/modeling

Partner in Switzerland – www.evocean.ch

UML is a trademark or registered trademark of Object Management Group, Inc. in the U.S. and other countries.



Analysis in Harmony ESW®

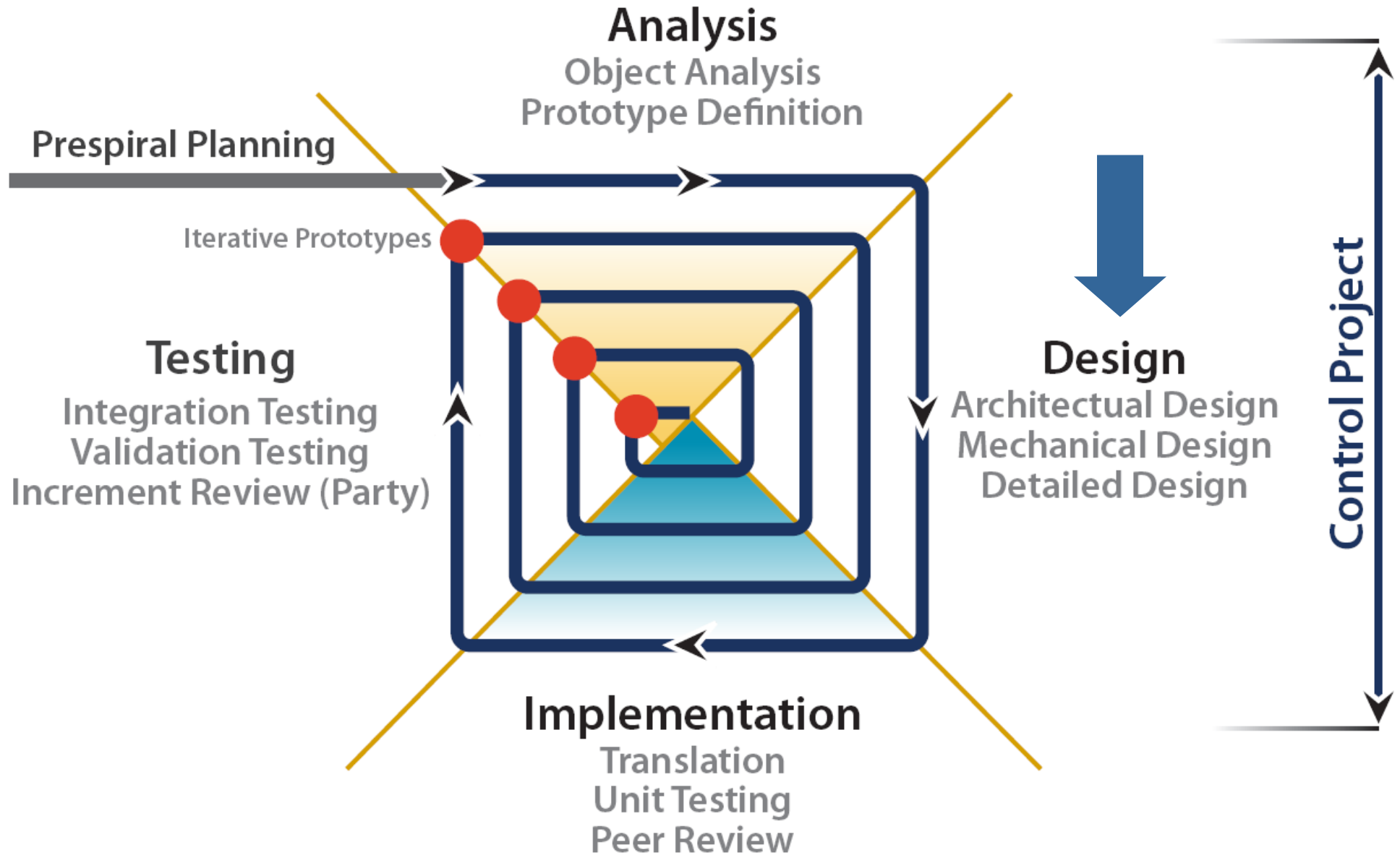


Analysis



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

Design in Harmony ESW®



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 (optimal) solution*
- Two approaches
 - *Elaborative Design*
Refine analysis model by adding more detail
 - *Translative Design*
Build a translator that embodies design decisions

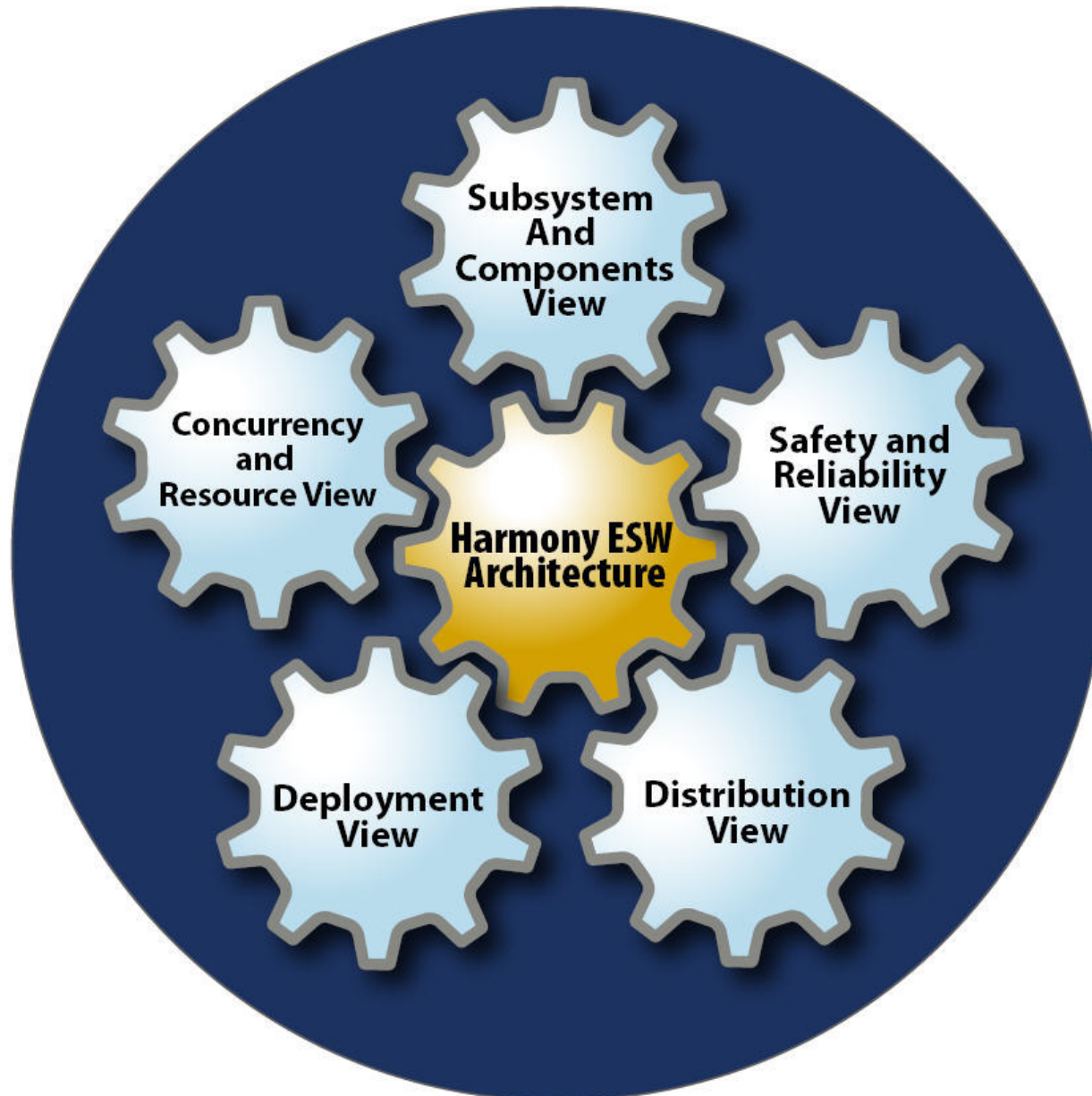
(Physical) Architectural Design

- (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.

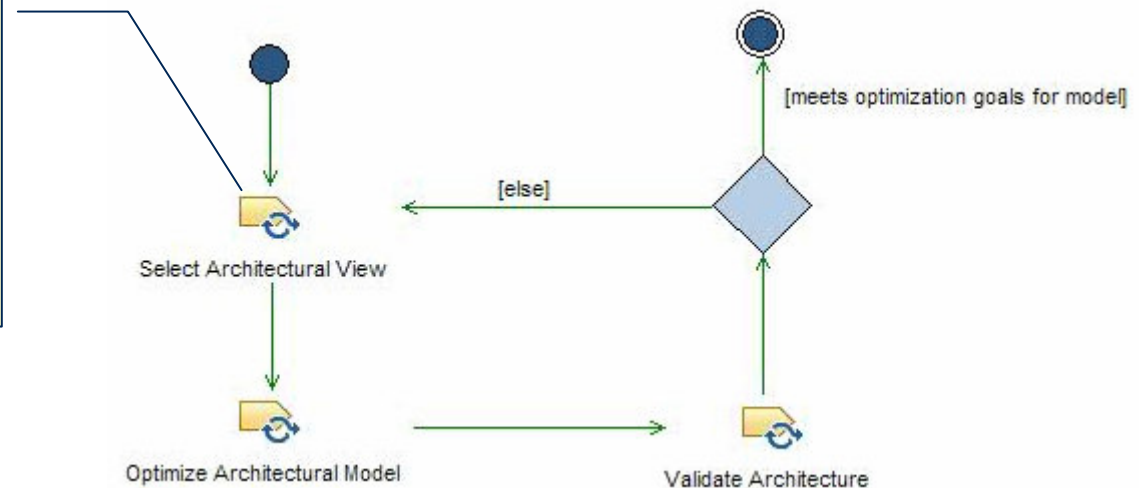
Aspects of Architecture



Architectural Design Workflow

Key views include:

- Subsystem and component view
- Concurrency and resource view
- Distribution view
- Safety and Reliability view
- Deployment view



Work Breakdown												
Breakdown Element	Steps	Index	Predecessors	Model Info	Type	Planned	Repeatable	Multiple Occurrences	Ongoing	Event-Driven	Optional	Team
+ Select Architectural View	●	26			Task	✓	✓	✓				
+ Optimize Architectural Model	●●●●●	27	26		Task	✓	✓	✓				
+ Validate Architecture	●●○	28	27		Task	✓	✓					



Not all views may be relevant within the mission of a given prototype, but all will be addressed before the project is completed.

Architecture: Optimize Architectural Model

Task: Optimize Architectural Model



This task focuses on the optimization of the entire system based upon a specific architectural view.

[Expand All Sections](#) [Collapse All Sections](#)

Purpose

The purpose of this task is to optimize the model by adding in strategic (i.e. architectural) design decisions.

[Back to top](#)

Relationships

Roles	Main: <ul style="list-style-type: none">• Software Architect	Additional: <ul style="list-style-type: none">• Reliability Czar• Safety Czar• Software Modeler• Subject Matter Expert	Assisting:
Inputs	Mandatory: <ul style="list-style-type: none">• Model	Optional: <ul style="list-style-type: none">• Architecture• Platform Independent Model	External: <ul style="list-style-type: none">• None
Outputs	<ul style="list-style-type: none">• Architecture• Platform Specific Model		

[Back to top](#)

Main Description

The input to the task is the un-optimized PIM (plus architectural decisions from previous prototypes). This task adds additional relevant architectural optimization decisions. Architecture is divided up into different subject matter views (see Guidance, below) each of which has its own specific design patterns. This step optimizes the system by selecting and applying patterns in one or more views.

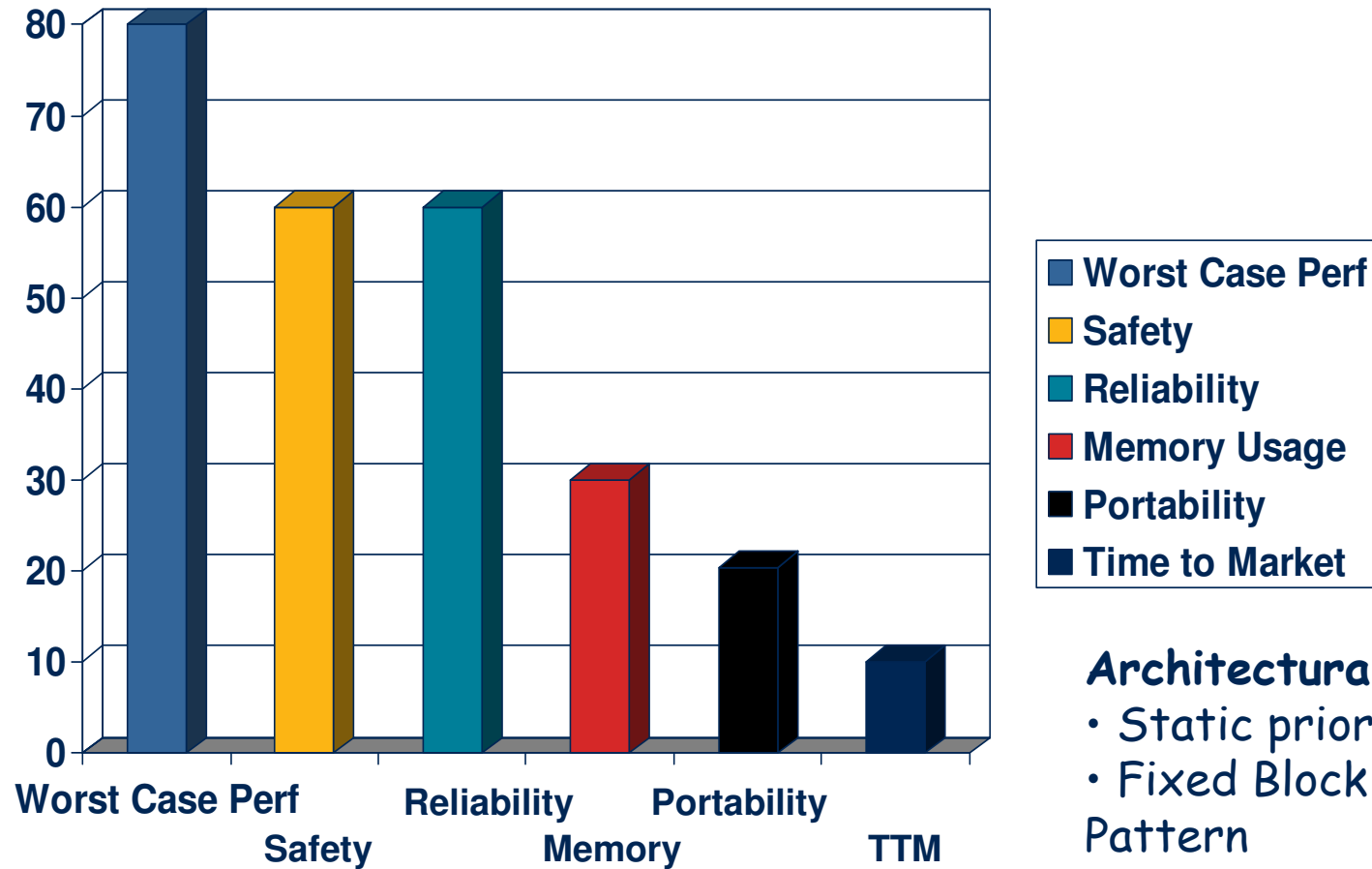
[Back to top](#)

Steps

[Expand All Steps](#) [Collapse All Steps](#)

- [Identify design criteria](#)
- [Rank design criteria](#)
- [Select design approach](#)
- [Apply design patterns](#)
- [Refine scenarios](#)

Example of Pattern Selection

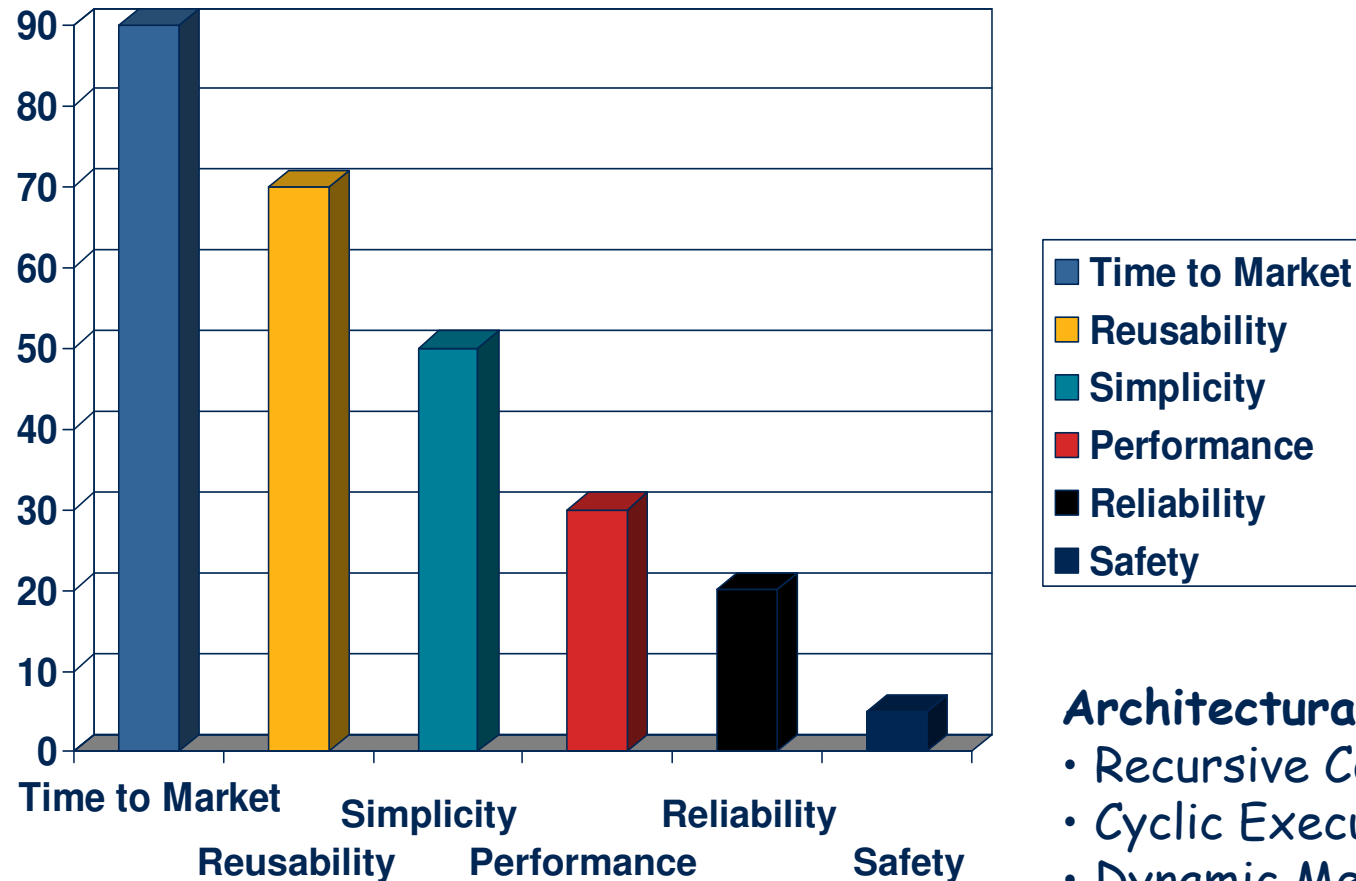


Design Criteria ranked by criticality

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

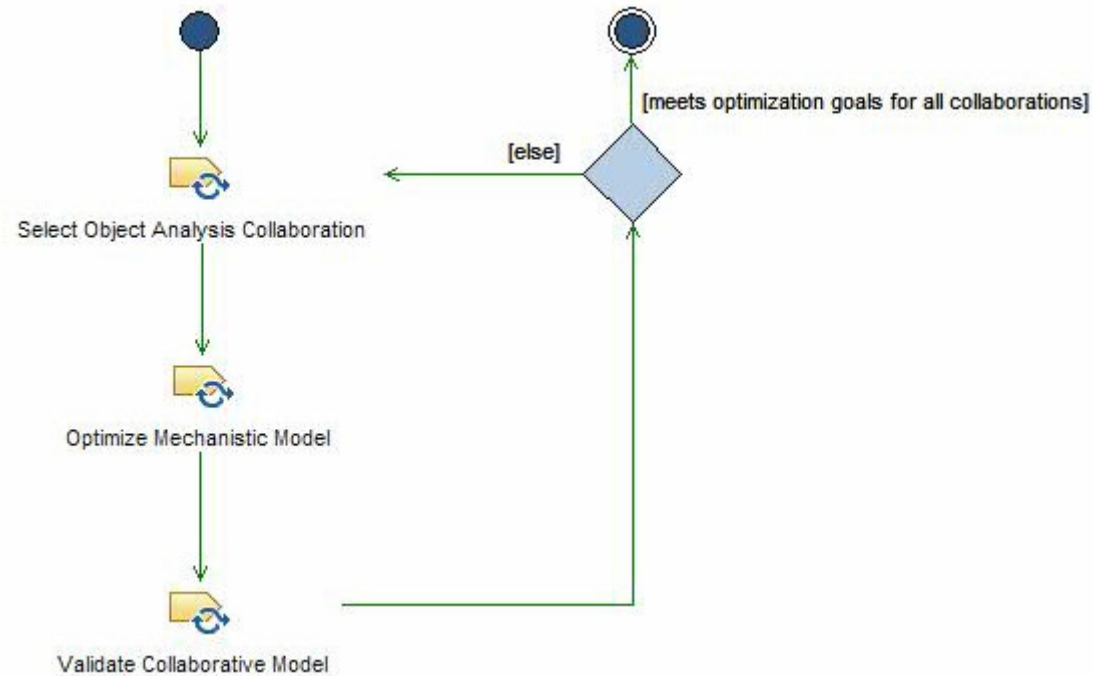
Architectural Patterns:

- 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

Mechanistic Design Workflow



Work Breakdown													
Breakdown Element	Steps	Index	Predecessors	Model Info	Type	Planned	Repeatable	Multiple Occurrences	Ongoing	Event-Driven	Optional	Team	
⊕ Select Object Analysis Collaboration	••	34			Task	✓		✓					
⊕ Optimize Mechanistic Model	•••••	35	34		Task	✓	✓	✓					
⊕ Validate Collaborative Model	•••	36	35		Task	✓	✓	✓					

Mechanistic Design: Optimize Collaboration

Task: Optimize Mechanistic Model



This task focuses on the optimization of a single design collaboration.

[Expand All Sections](#) [Collapse All Sections](#)

Purpose

The purpose of this task is to optimize the system, at some level of abstraction, to make the system more usable and to meet quality of service requirements.

[Back to top](#)

Relationships

Roles	Main: <ul style="list-style-type: none">Software Modeler	Additional: <ul style="list-style-type: none">Reliability CzarSafety CzarSoftware ArchitectSubject Matter Expert	Assisting:
Inputs	Mandatory: <ul style="list-style-type: none">Model	Optional: <ul style="list-style-type: none">Platform Independent Model	External: <ul style="list-style-type: none">None
Outputs	<ul style="list-style-type: none">Platform Specific Model		

[Back to top](#)

Main Description

The Harmony/Embedded process is very design-pattern driven. A design pattern is a generalized solution to a commonly occurring problem; as such each pattern optimizes some aspects of the system against others. The optimization process must first identify against which criteria the optimization should be done, rank them in order of importance, and select design solutions that provide the desired optimizations at an acceptable cost.

[Back to top](#)

Steps

[Expand All Steps](#) [Collapse All Steps](#)

- [Understand the design collaboration that the model is being optimized around](#)
- [Identify design criteria](#)
- [Rank design criteria](#)
- [Select design approach](#)
- [Apply design patterns](#)
- [Refine scenarios](#)

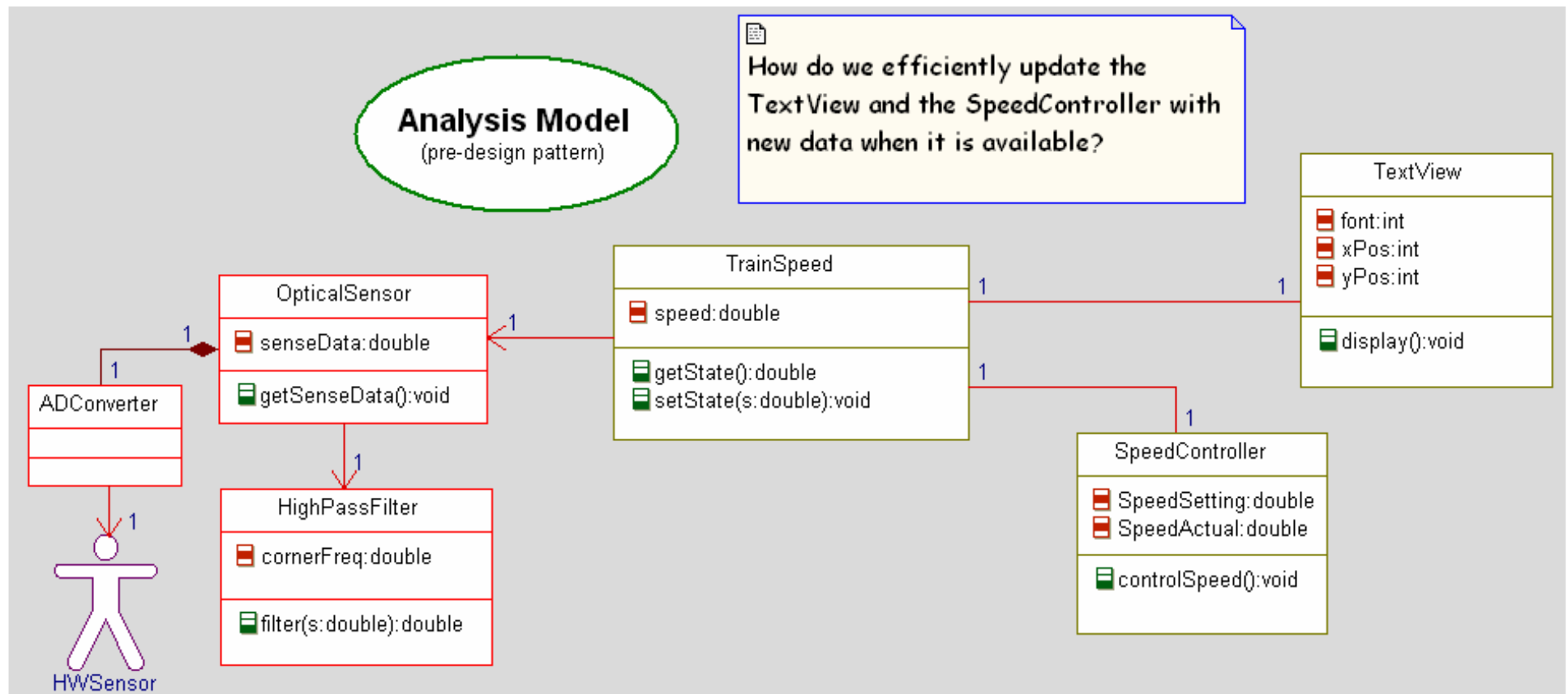
Detailed Design

- 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

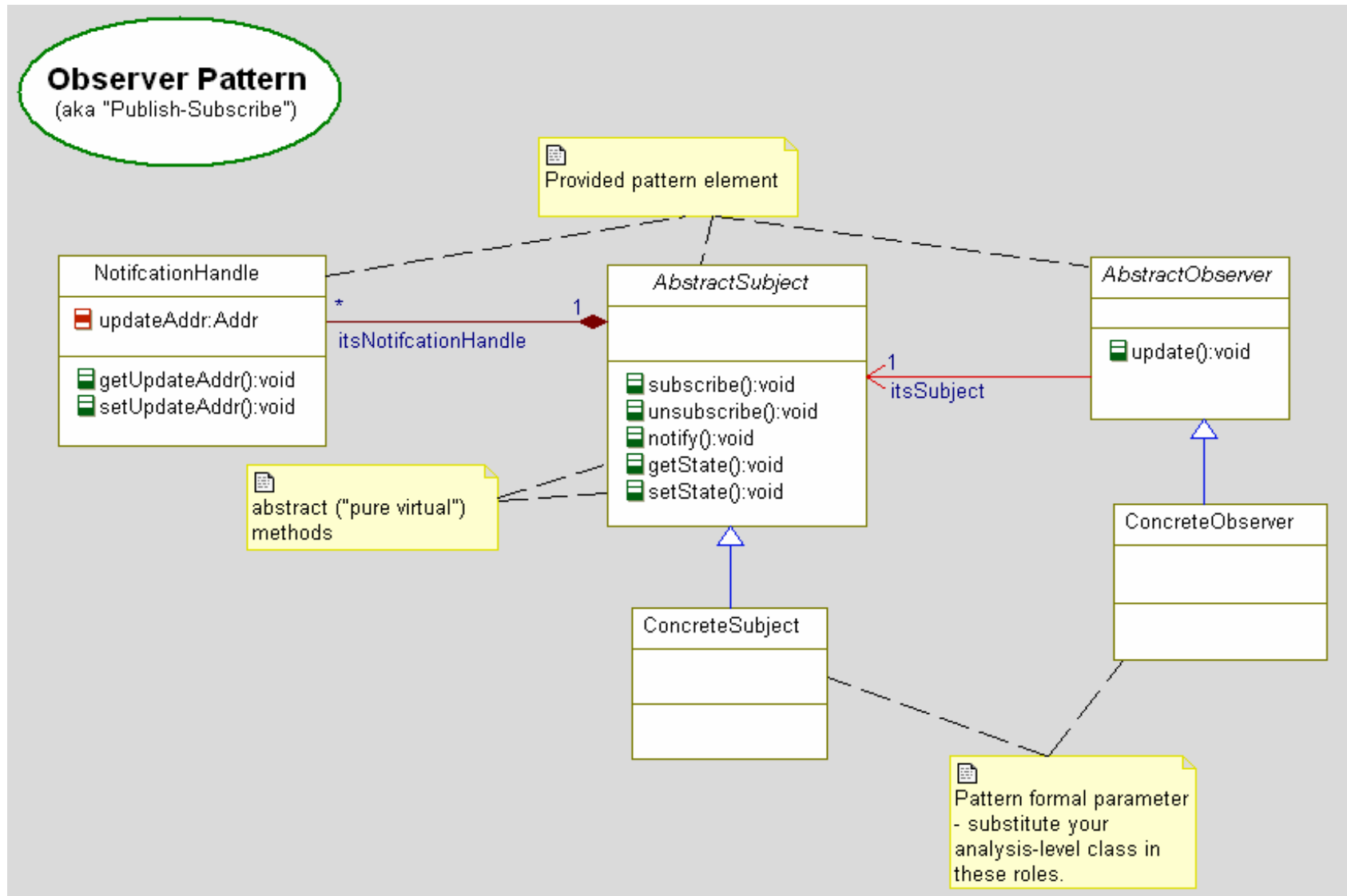
Design Patterns

- 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

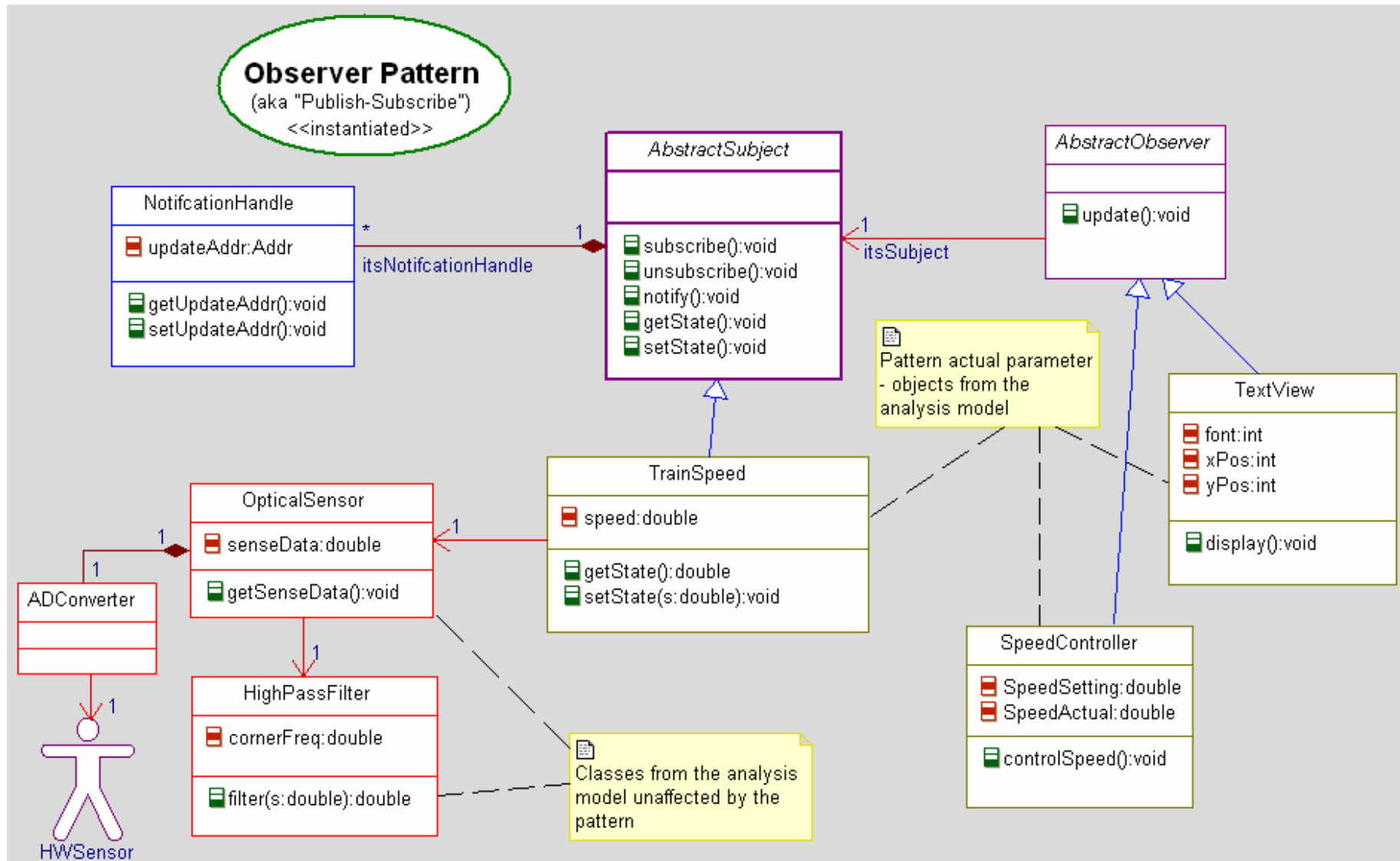
Example Analysis Model Collaboration



Design Pattern: Observer Pattern Specification



Design Pattern: Observer Pattern Instantiation



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

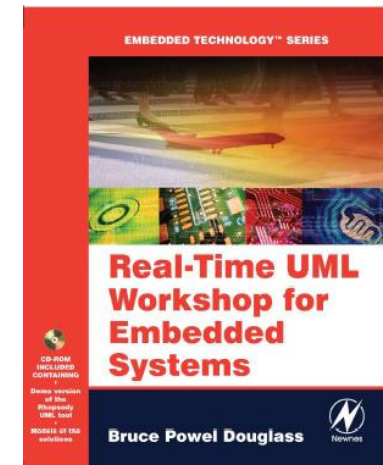
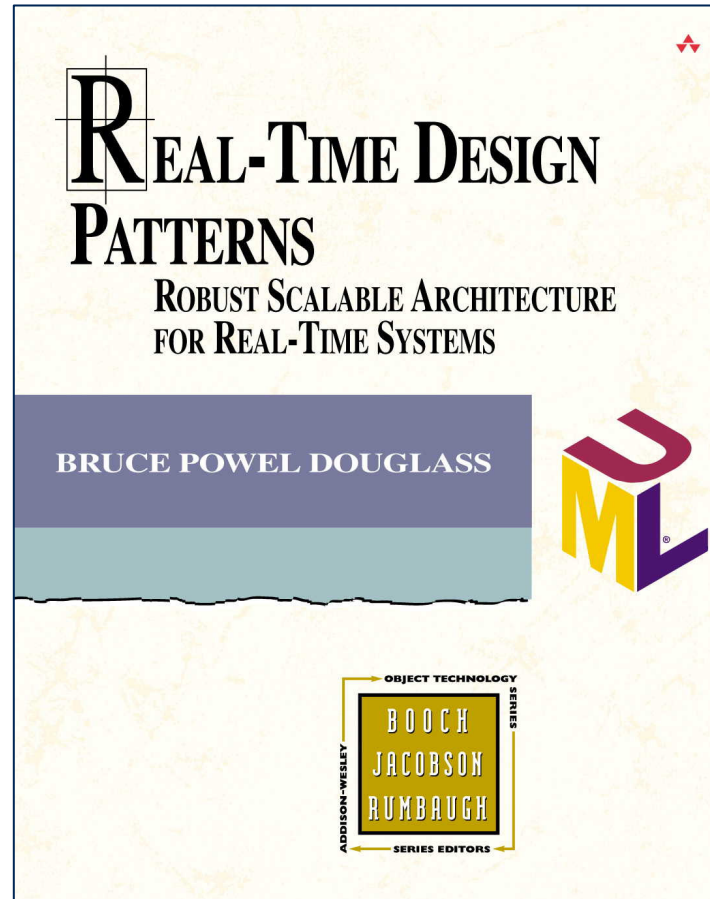
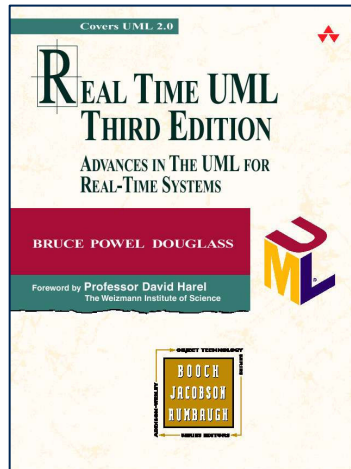
Properties of Design Patterns

- Problem Context
 - Statement of the characteristics of a problem context that make the solution useful
 - Statement of the characteristics optimized by the solution
- Solution
 - The design pattern *per se*
 - Often shown as a class diagram with accompanying sequence and state diagrams
 - Often shown with a particular example application
- Consequences
 - Design is all about *optimization*
 - *Every* pattern has benefits and costs
 - *Benefits* are optimizations that can be achieved using the pattern
 - *Costs* are deoptimizations that result from the use of the pattern

How do I Apply Design Patterns???

1. Construct the (working) initial model (PIM)
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
 - Architectural patterns apply *system-wide*
 - Mechanistic patterns apply *collaboration-wide*
 - Detailed design patterns apply *class-wide*
5. Apply the design patterns
6. Test the (working) design solution (PSM)
 - Is the original functionality retained?
 - Are the desired optimizations achieved?

A Brief Selection of Important 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

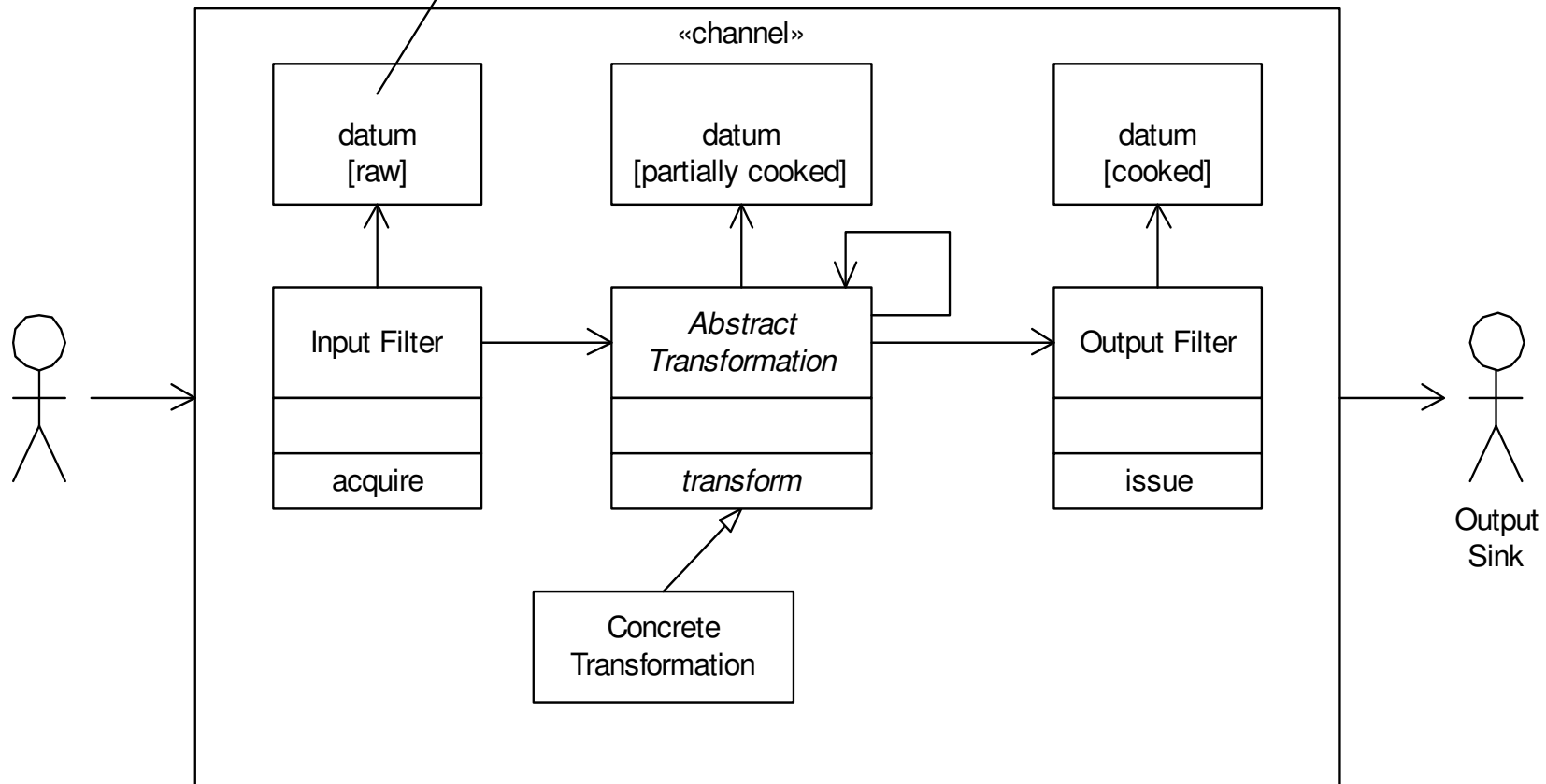
- Construct the system as a channel, a large scale subsystem which handles data from acquisition all the way through dependent actuation. Provide as many independent channels as necessary.

- Consequences

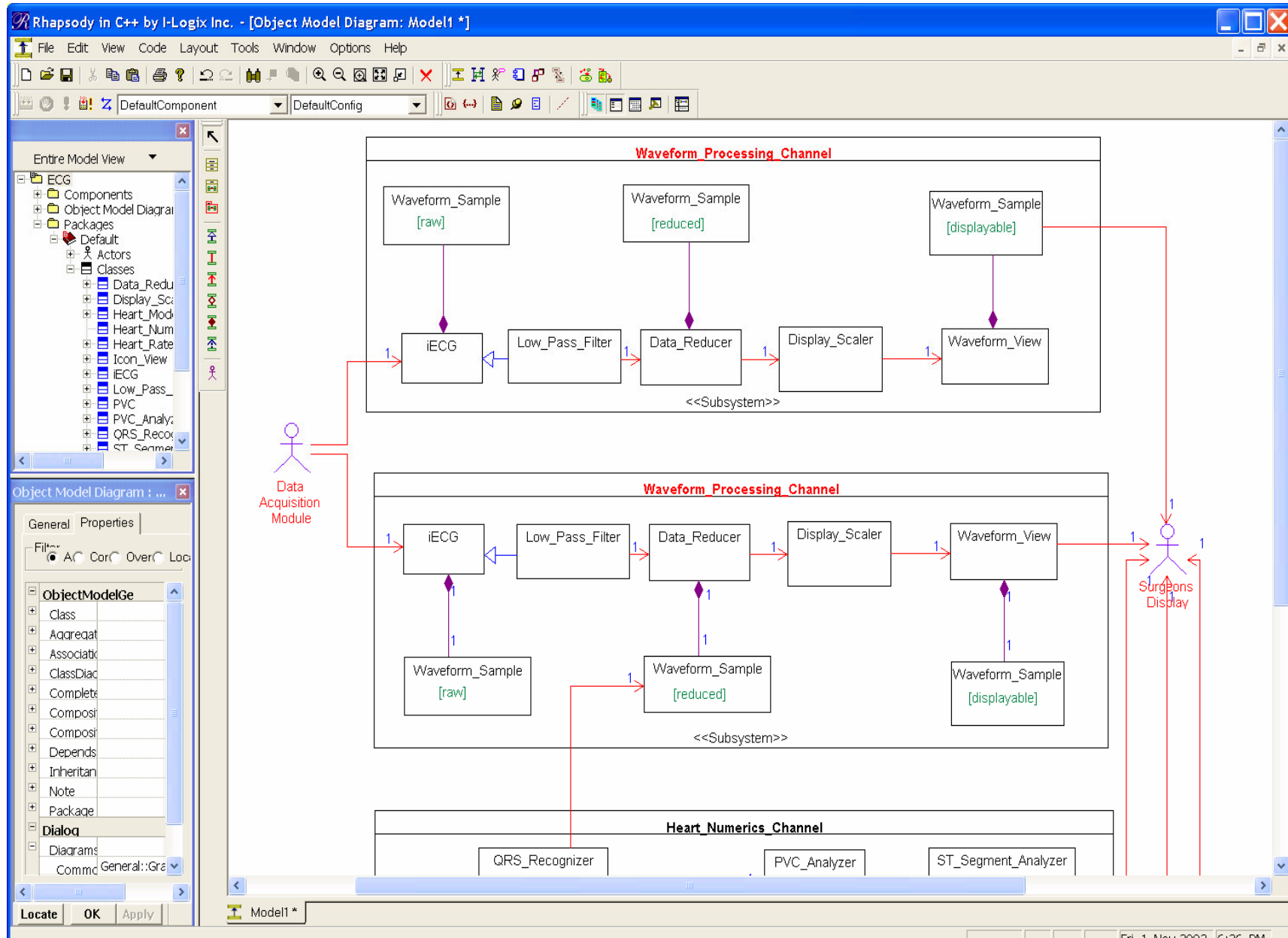
- A simple organizational pattern that permits redundancy to be easily added.
- May use additional memory since channels are designed to be independent, requiring replication (redundancy)

Channel Pattern

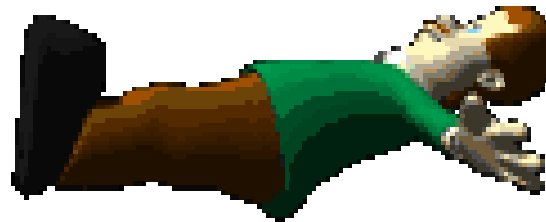
"Object in State" notation in the UML. State is in square brackets.



Channel Pattern Example



Concurrency Architecture Patterns



Message Queuing Pattern

- Problem

- A simple way to request services be performed across thread boundaries in a thread-robust way

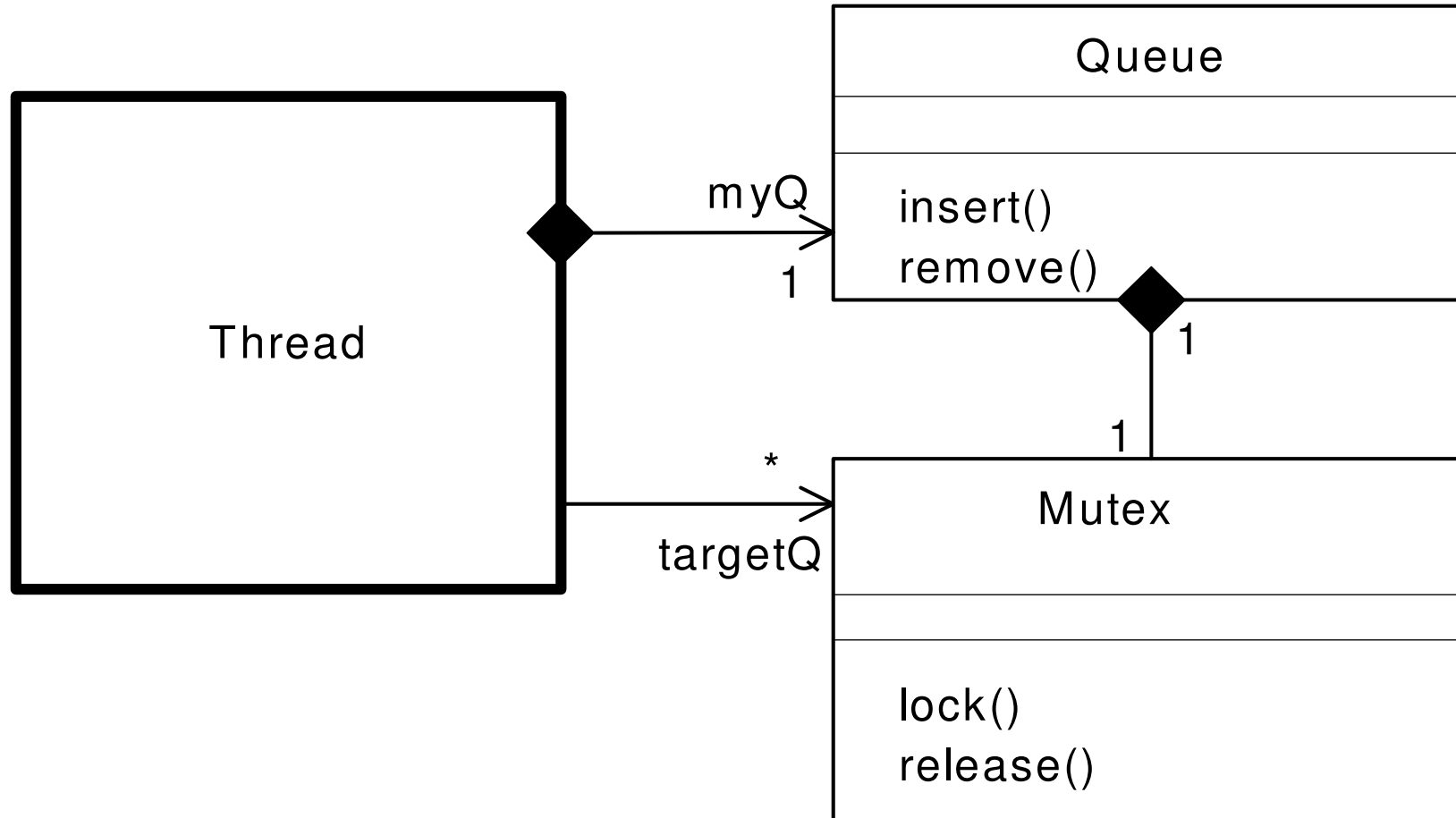
- Solution

- 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

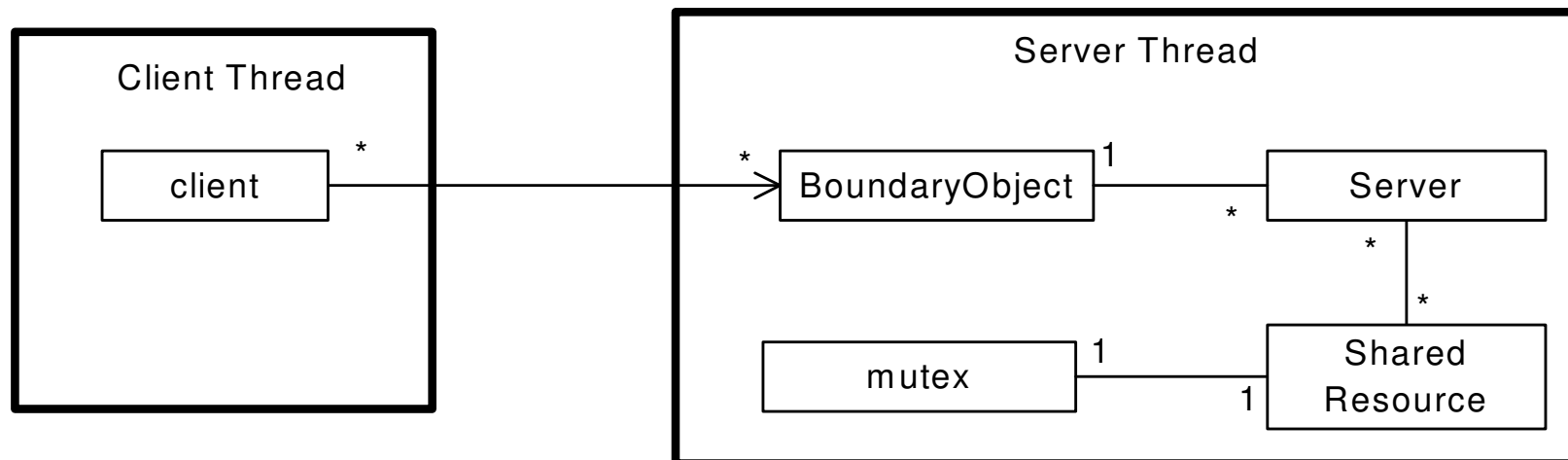
Message Queuing Pattern



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
 - Leads 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.

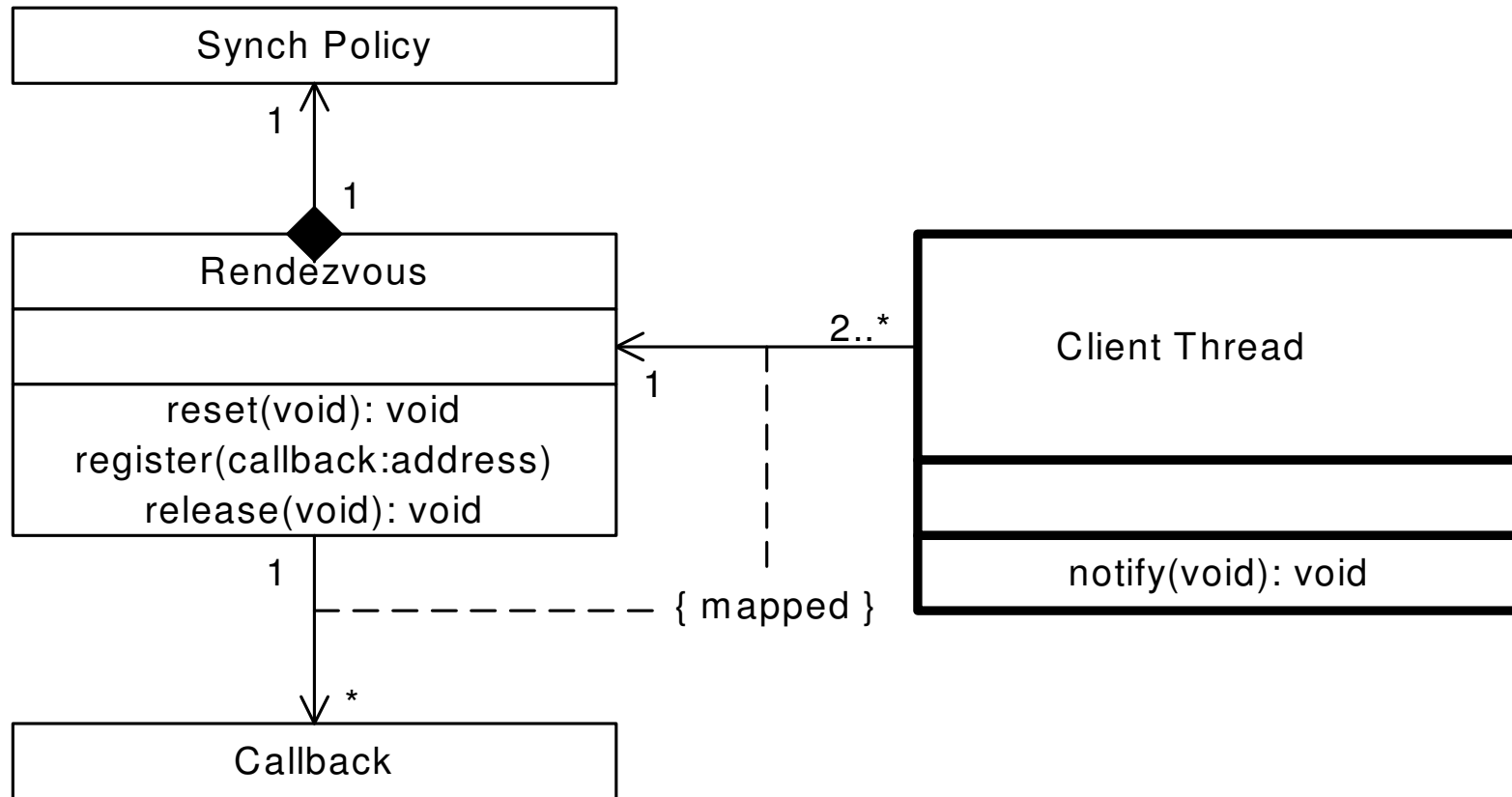
- Solution

- Reify the rendezvous policy as a class that mediates how the threads collaborate

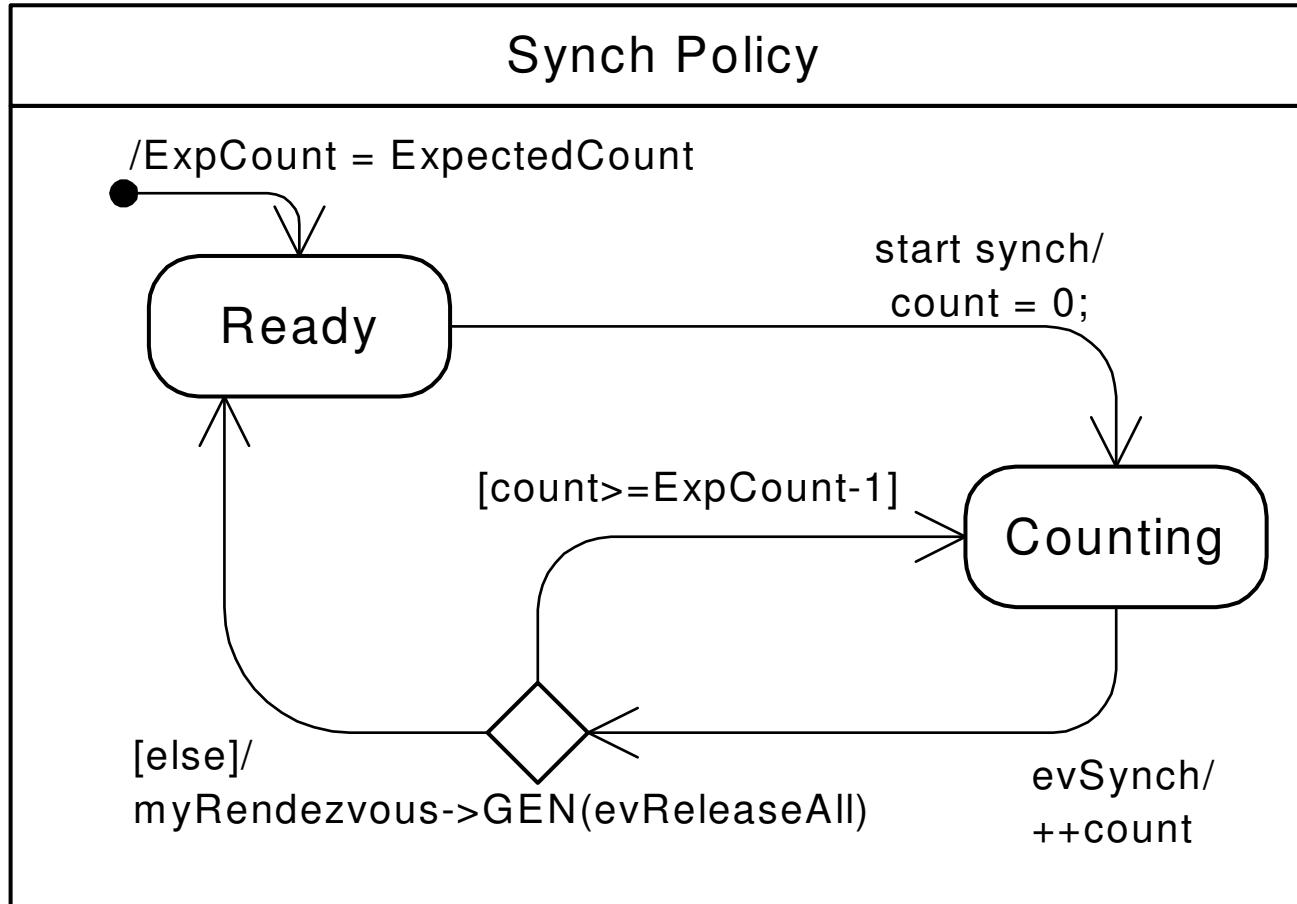
- Consequences

- 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



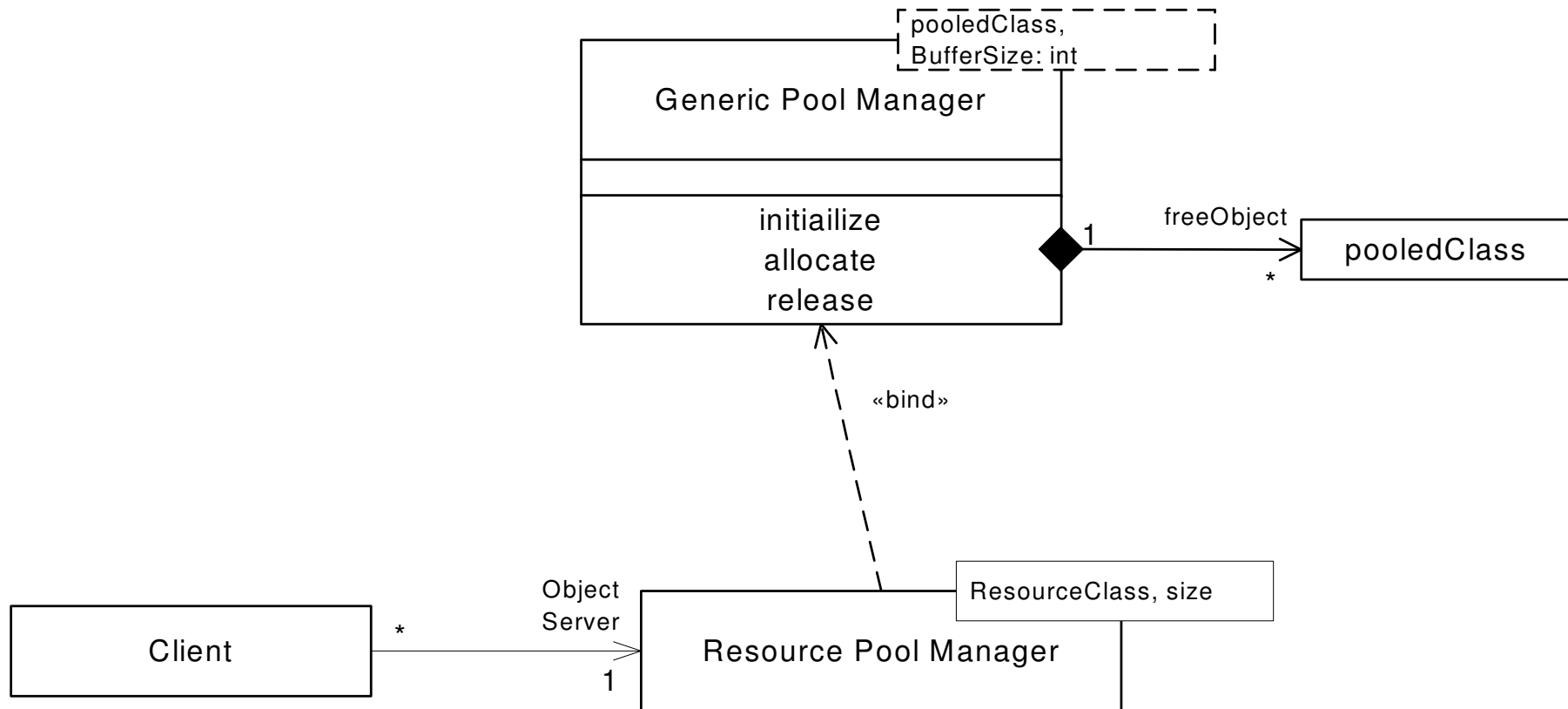
Memory Patterns

- 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

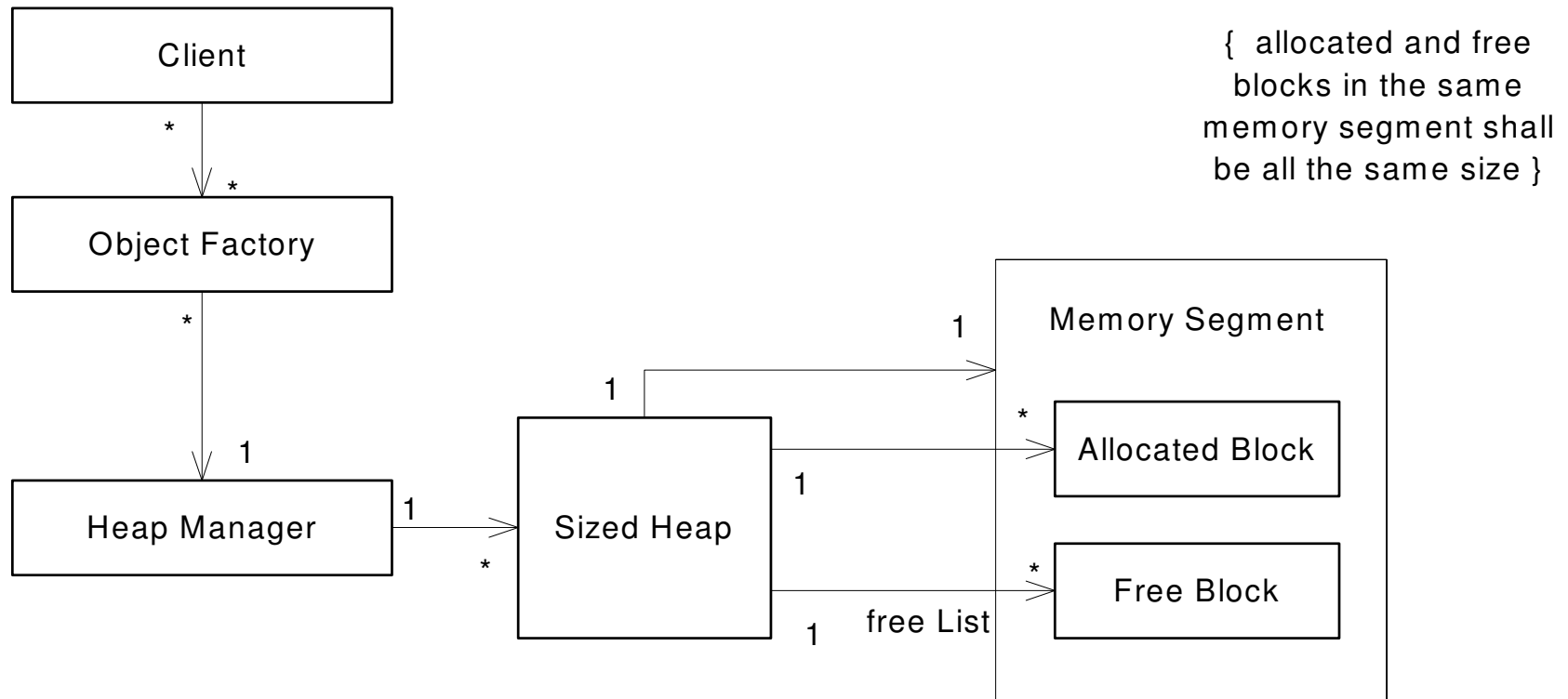
- Solution

- 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

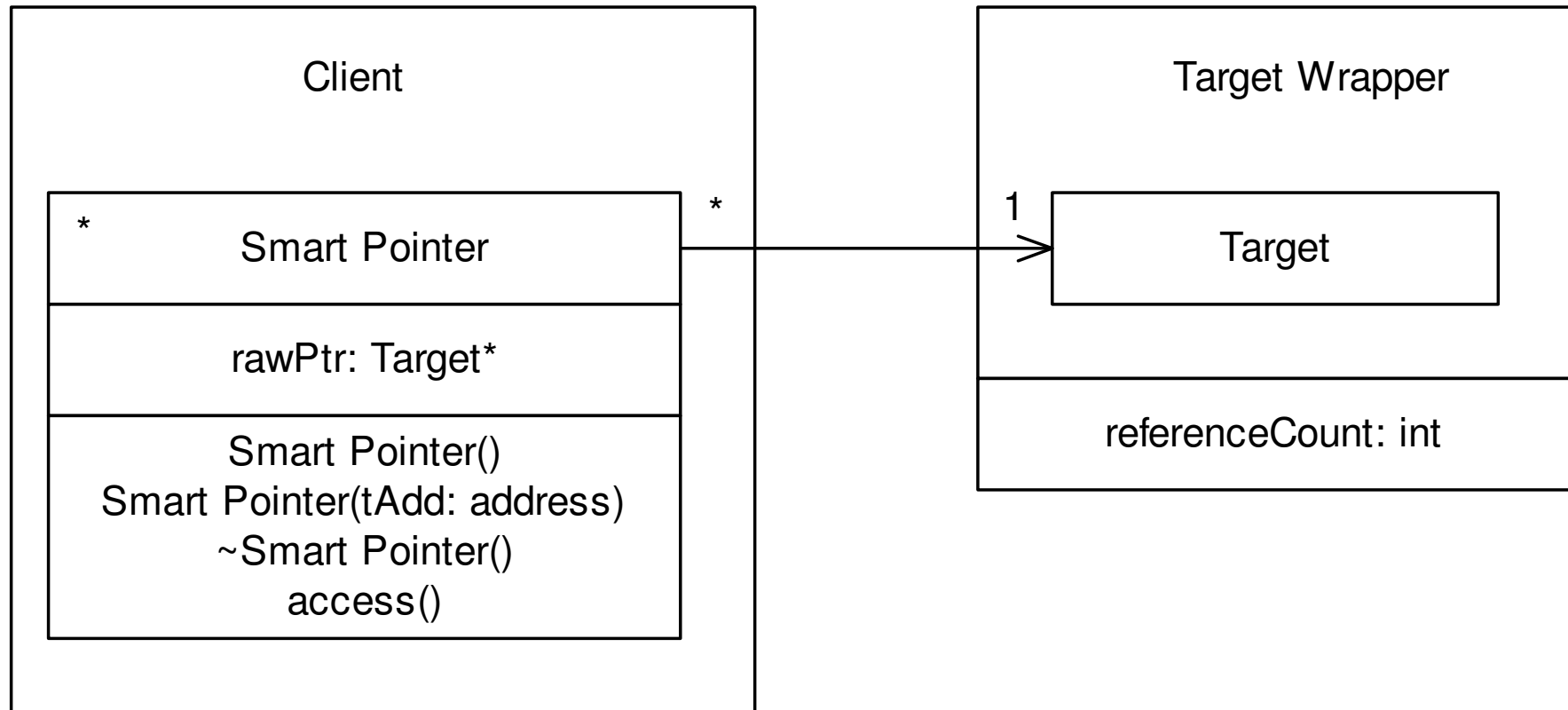
Fixed-Size Buffer Pattern



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
 - Small run-time overhead on every pointer manipulation
 - Doesn't work with cyclic data structures
 - Doesn't work well with mixed raw and smart pointers

Smart Pointer Pattern



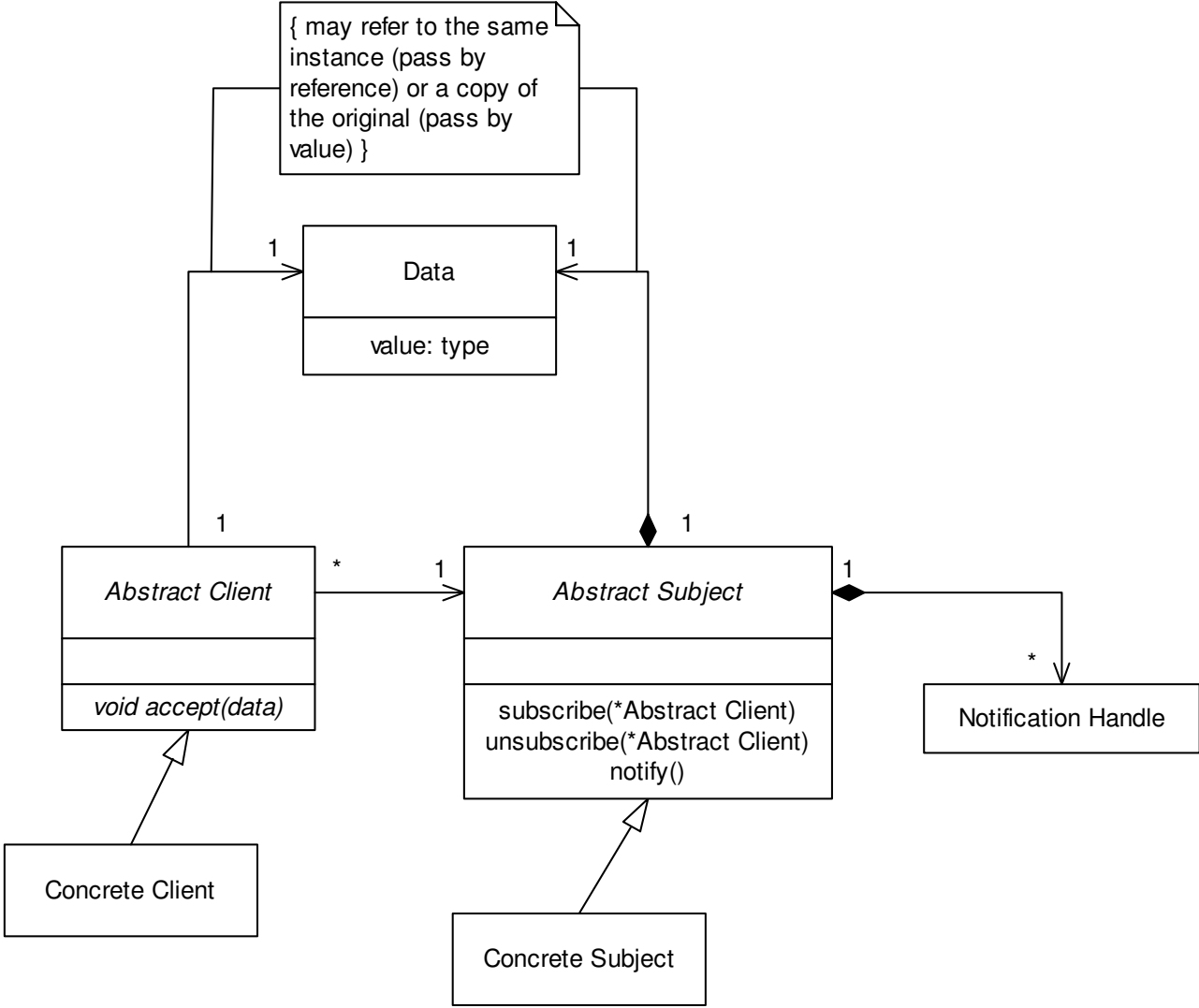
Distribution Patterns

- Observer Pattern
- Proxy Pattern
- Broker Pattern

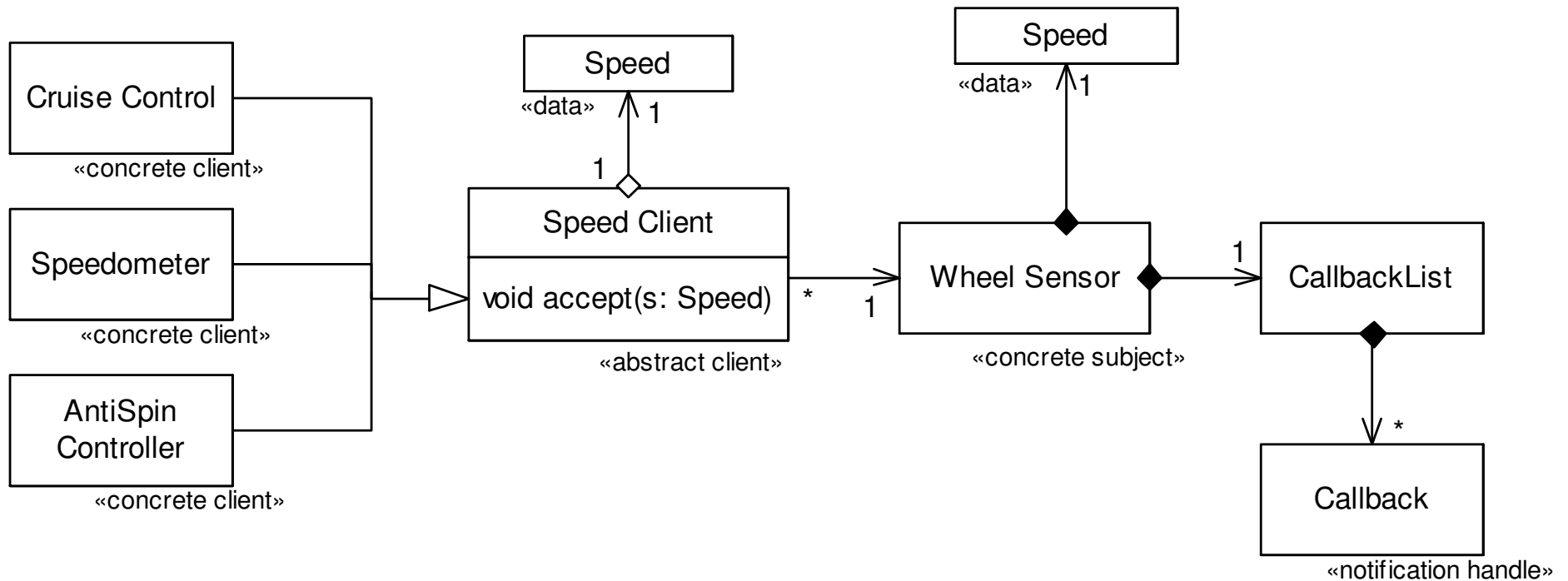
Observer Pattern

- 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
 - Clients are only updated when the update criteria is met not necessarily when the client *wants* the data

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

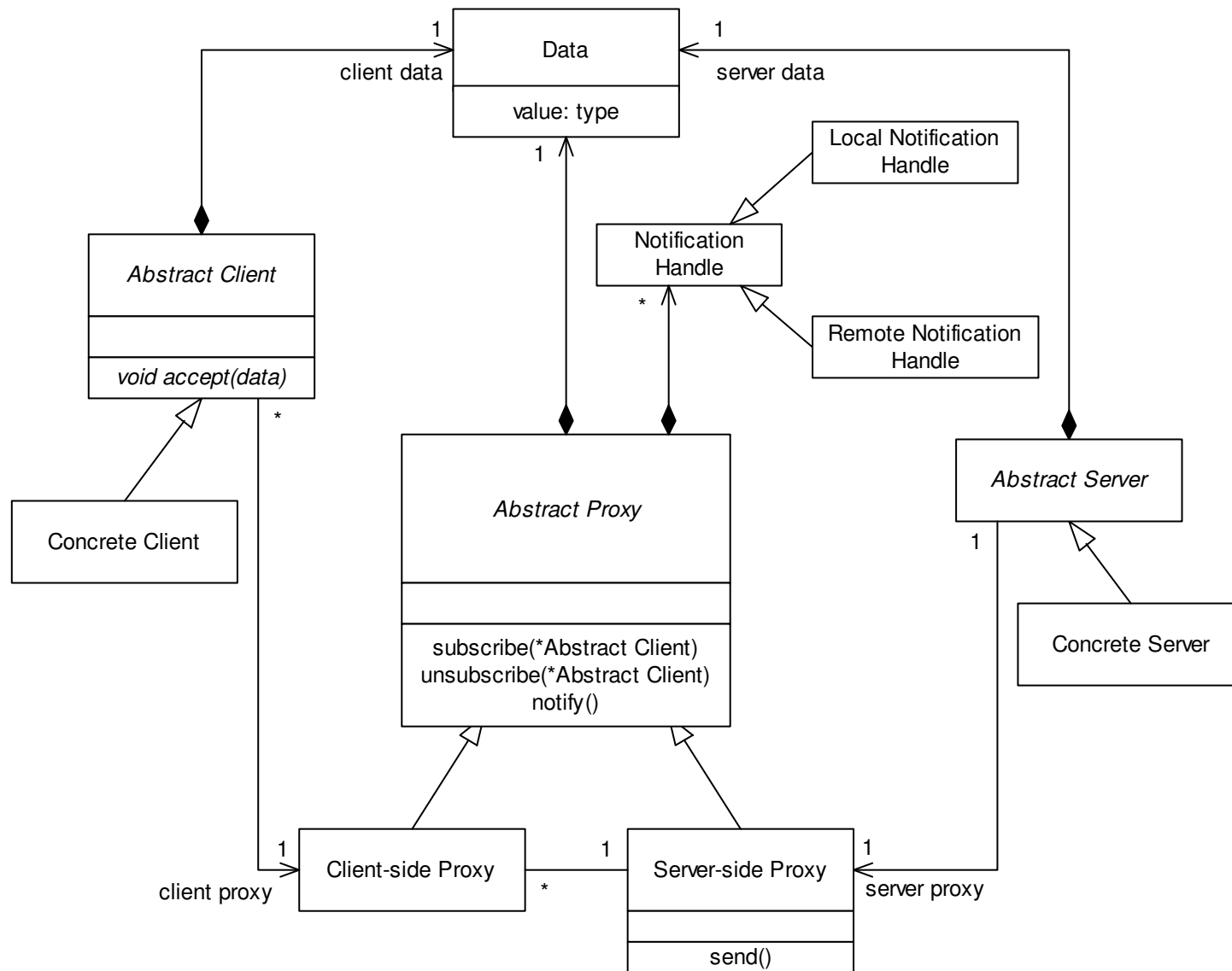
- Solution

- Similar to a observer pattern but with client and server proxies to isolate the required infrastructure

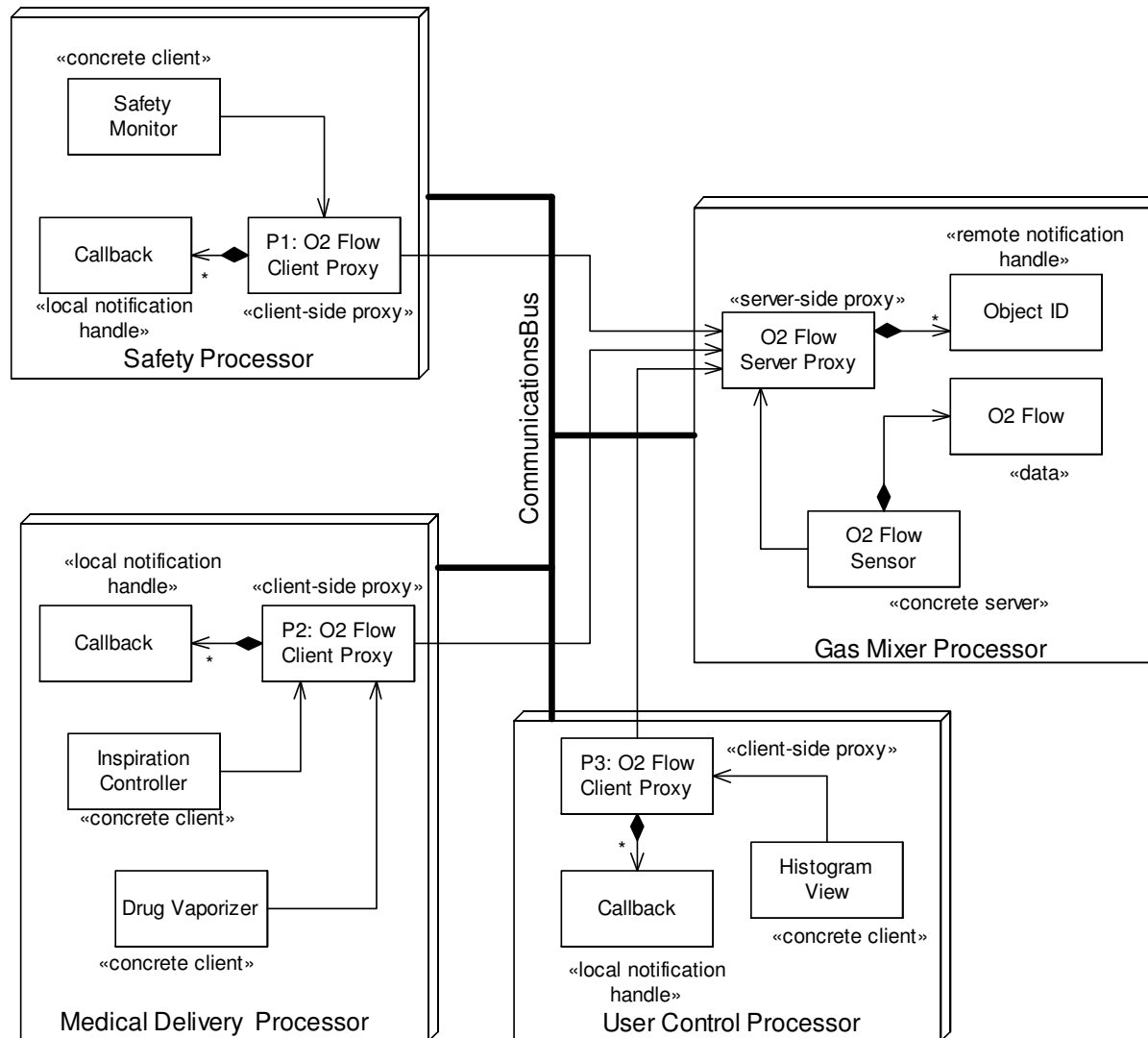
- Consequences

- A simple adaptation of the observer pattern
- Good isolation of subject from communications
- Optimizes bus traffic

Proxy Pattern



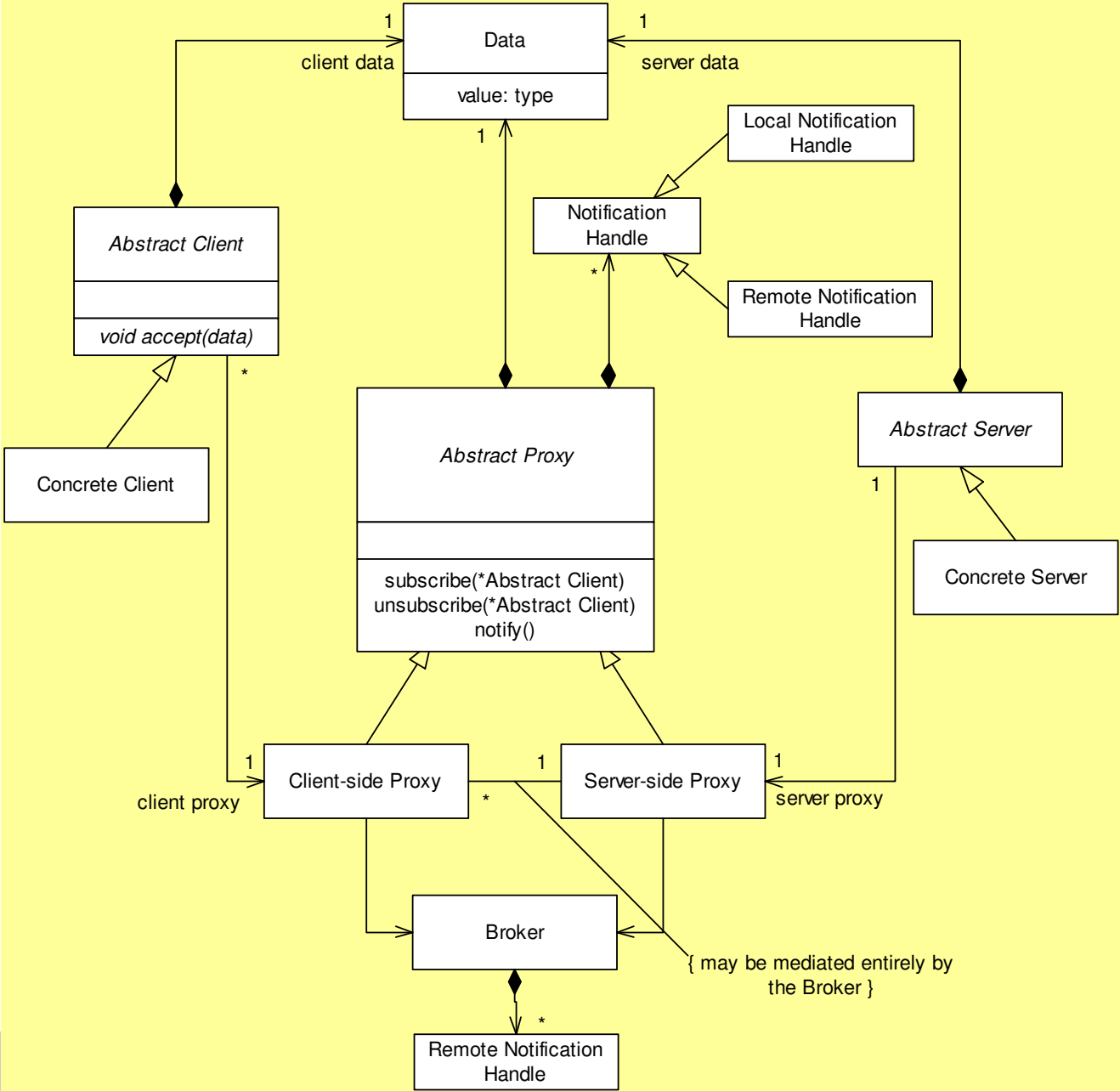
Proxy Pattern Example



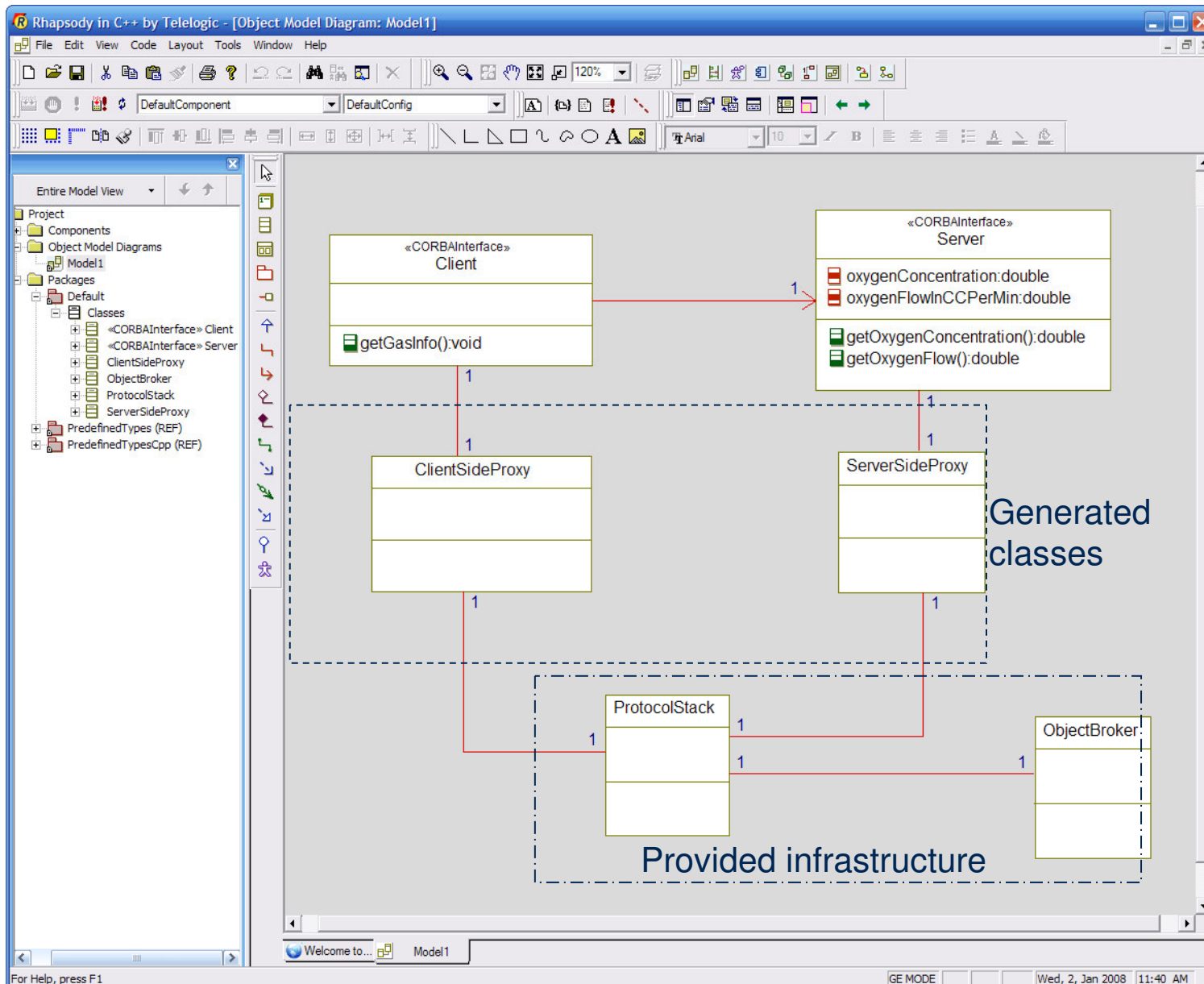
Broker Pattern

- 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

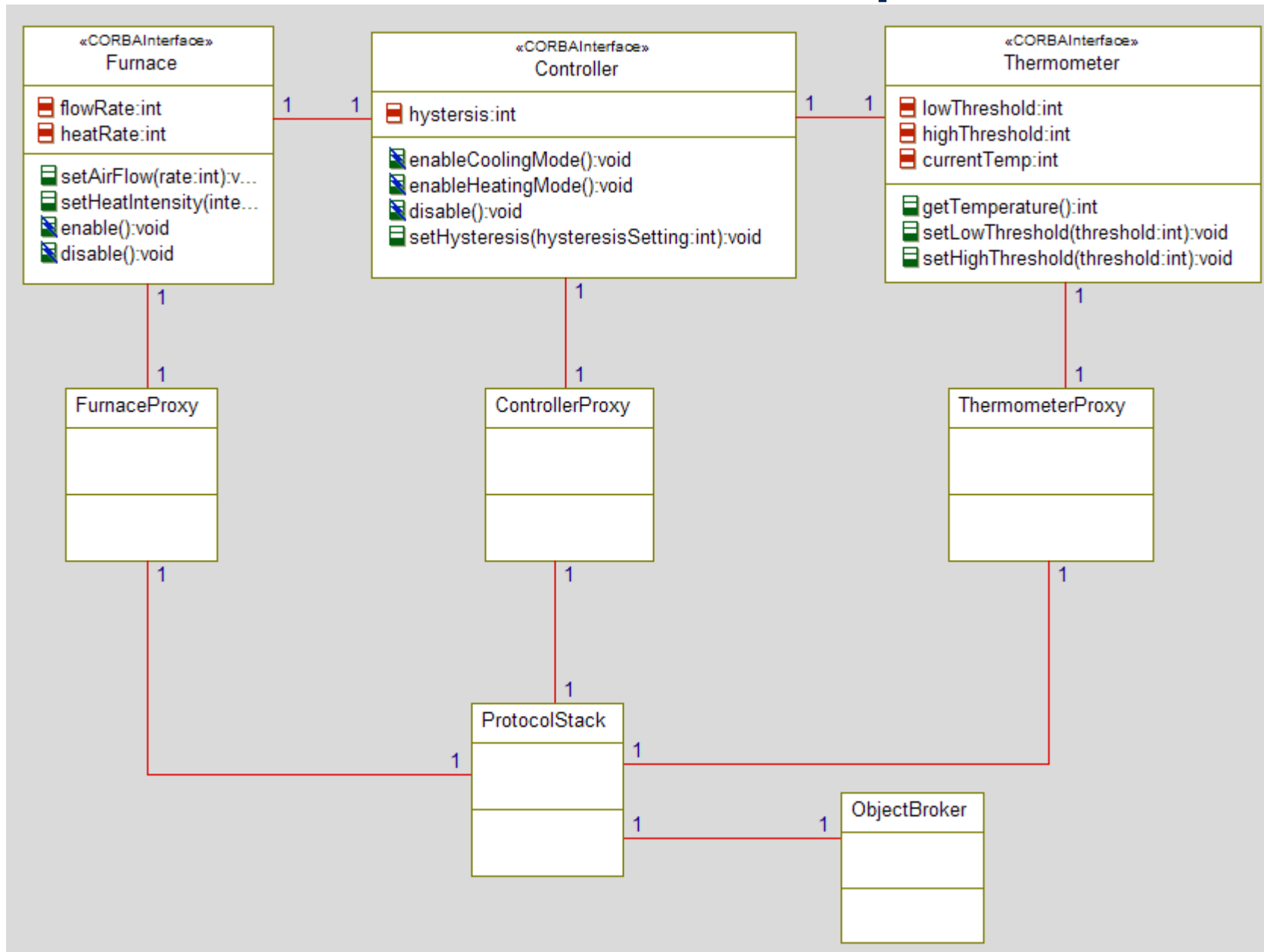
Broker Pattern



Broker Pattern Example



Broker Pattern Example



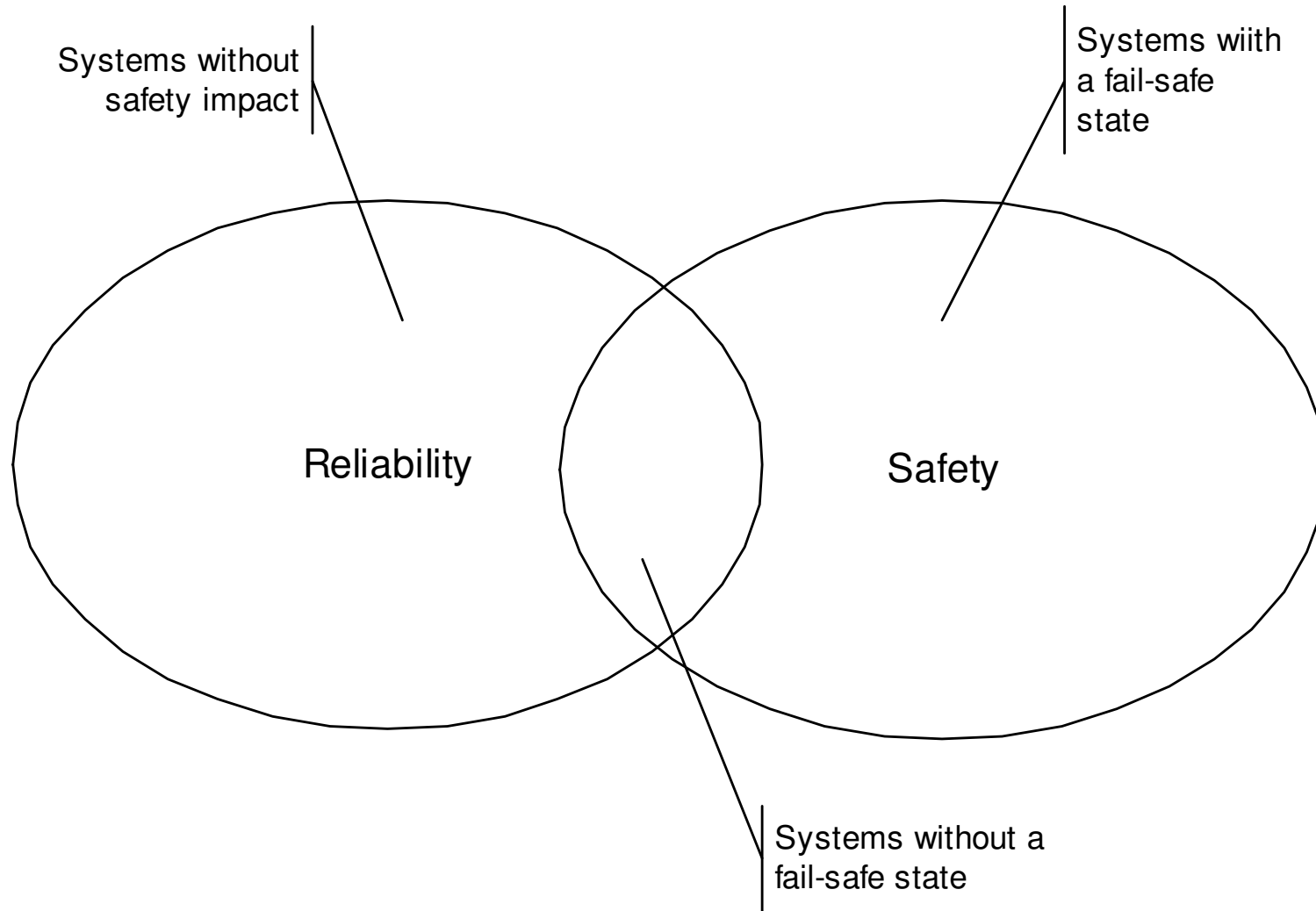
Safety and Reliability Patterns

- Protected Single Channel Pattern
- Homogeneous Redundancy Pattern
- Triple Modular Redundancy Pattern

Safety and Reliability

- Reliability is measured by MTBF
 - MTBF = probability of being able to deliver functionality
 - Reliability is a *stochastic* measure
- Safety is measured by risk
 - risk = severity_{fault} * likelihood_{fault}
- Fault is nonconformant behavior
 - Error: a design or implementation flaw
 - Hardware
 - Software
 - Failure: breakage of something that was previously correct
 - Hardware only

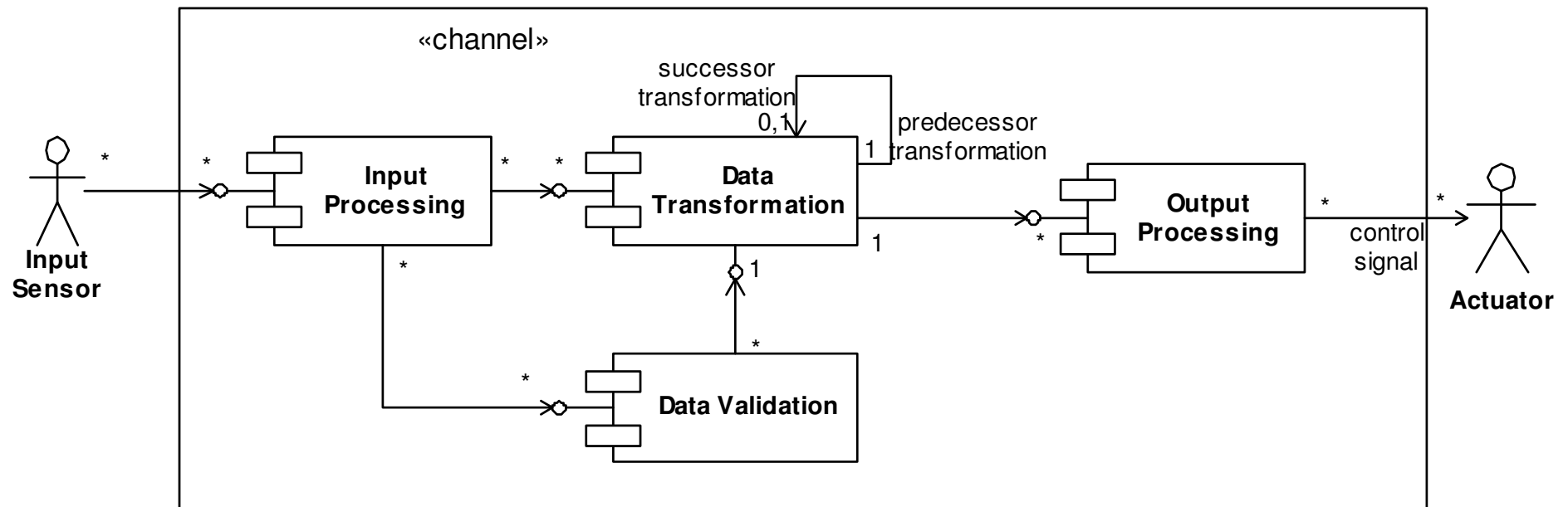
Safety vs Reliability



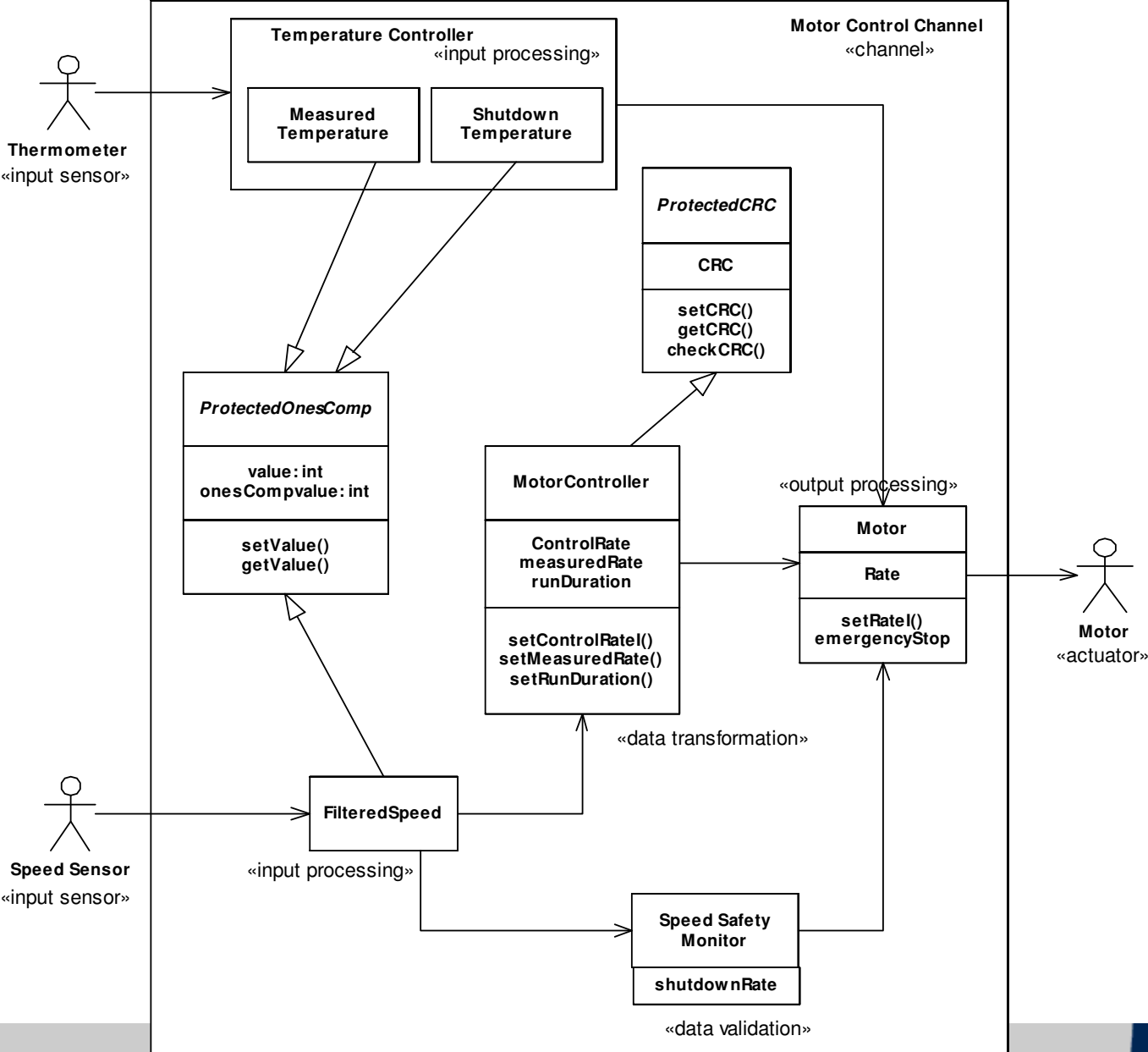
Protected Single Channel Pattern

- 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



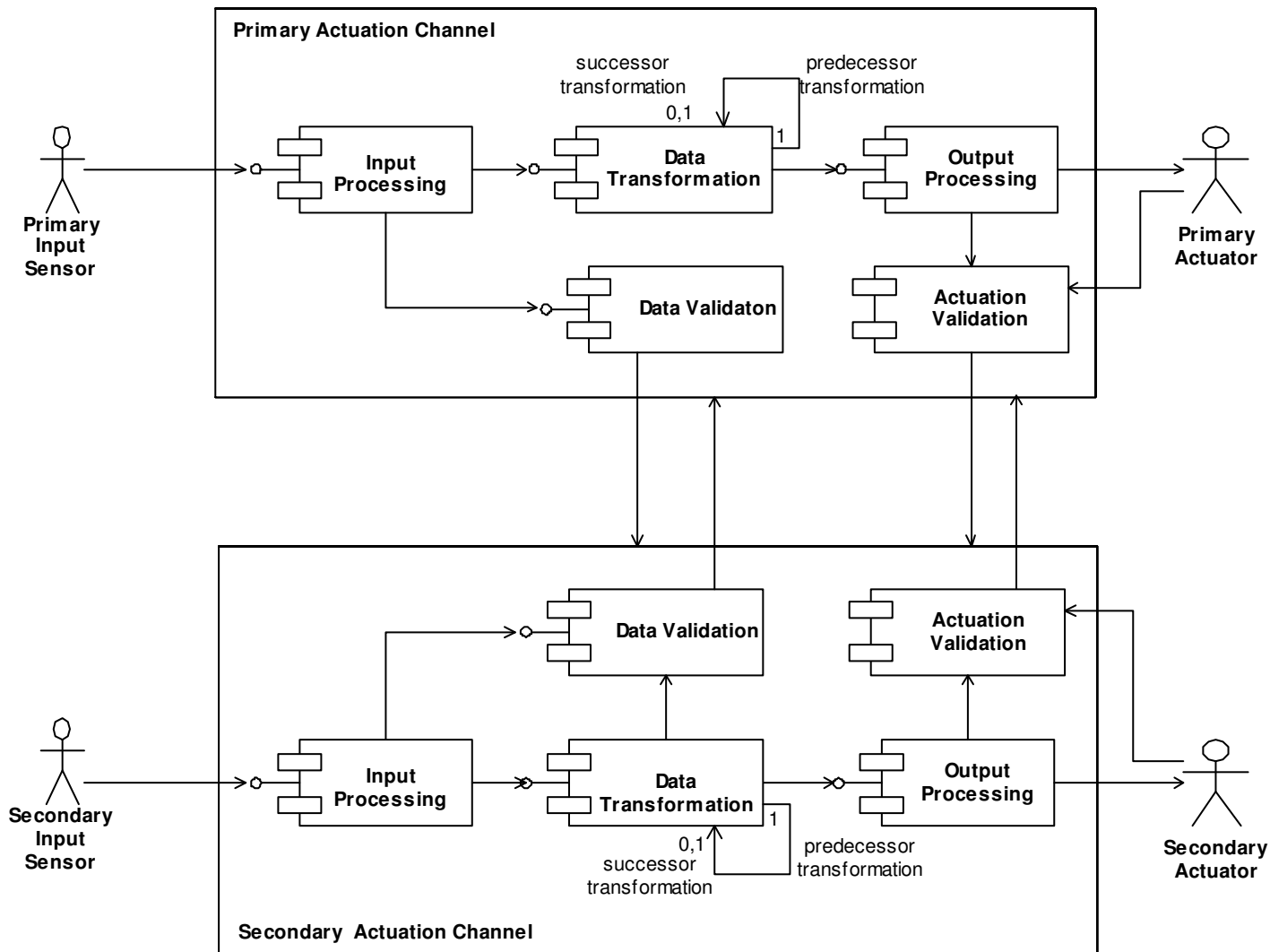
Protected Single Channel Pattern Example



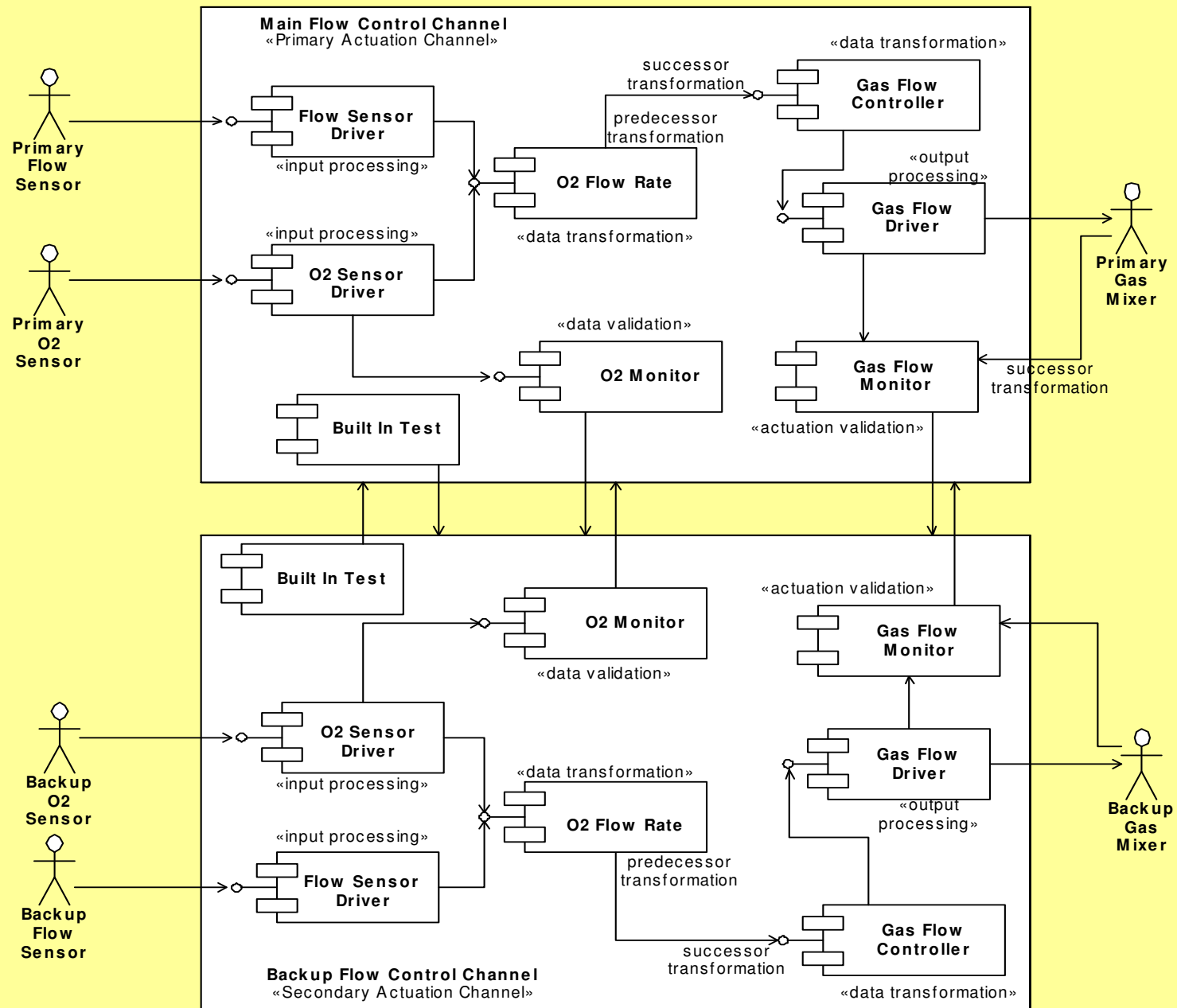
Homogeneous Redundancy Pattern

- 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



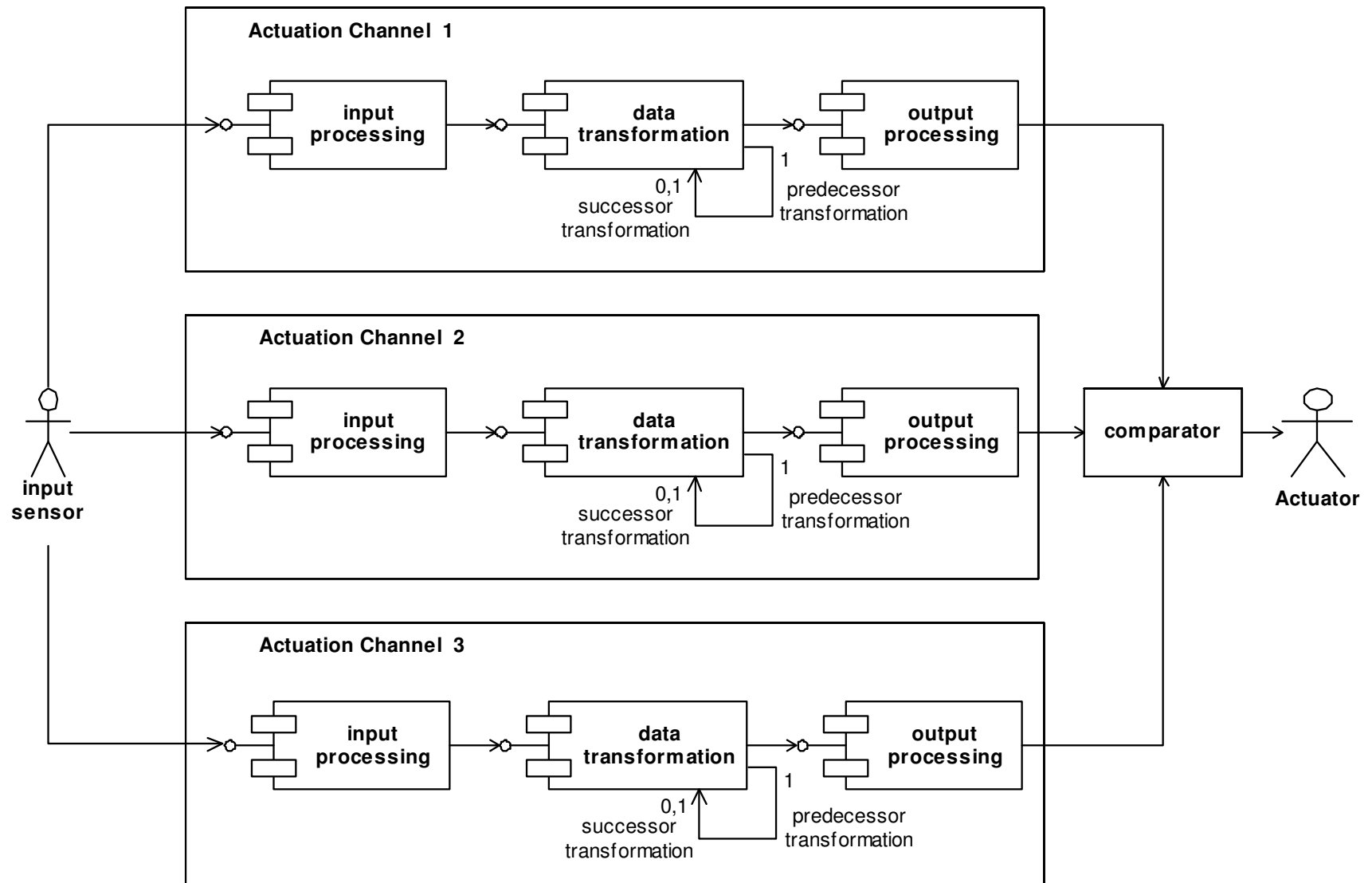
Homogeneous Redundancy Pattern Example



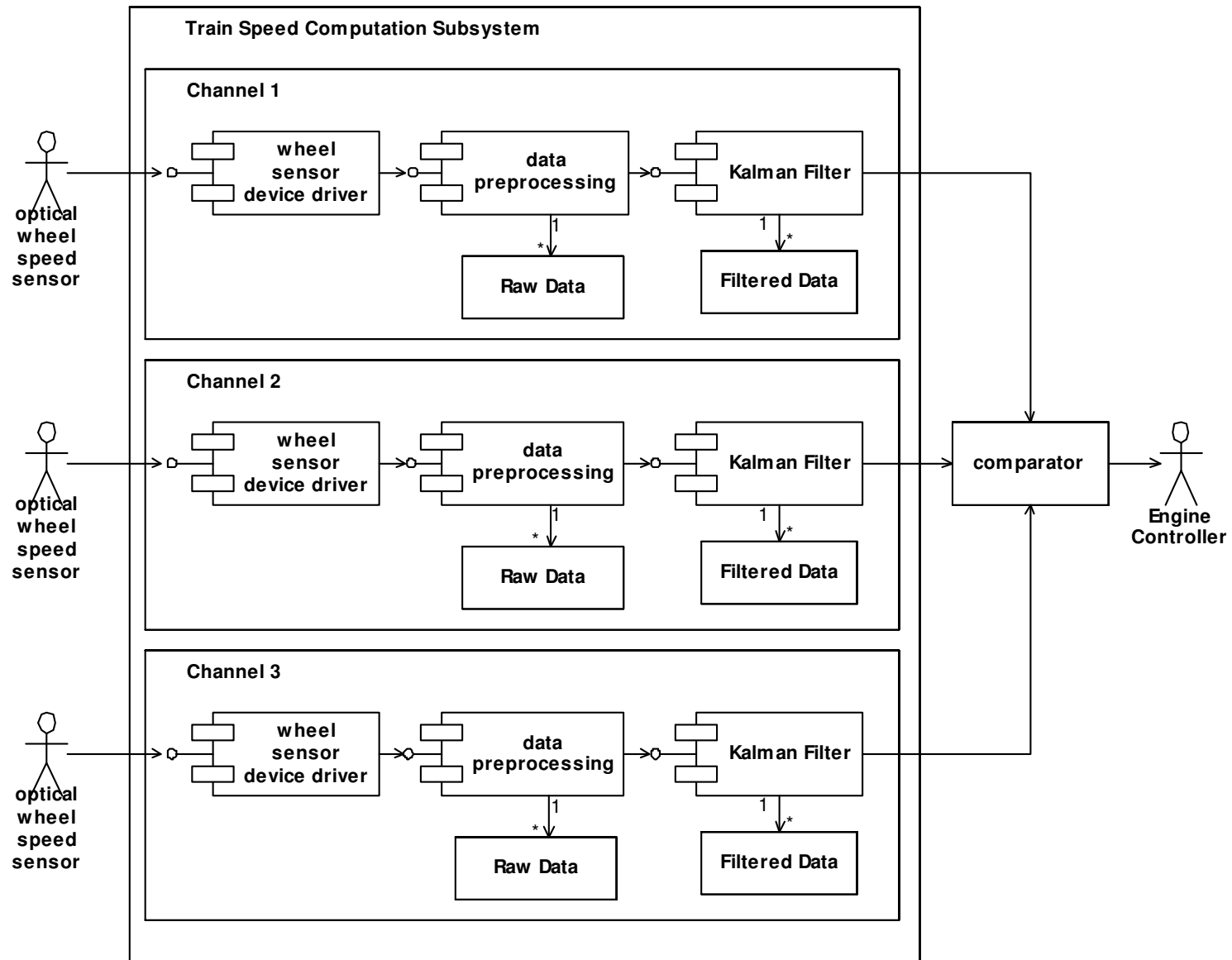
Triple Modular Redundancy Pattern

- 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



Real-Time Pattern References

White papers on RT UML and related topics at www.telelogic.com/modeling

