Lecture Notes on

Object Design

Bill Scherlis / Bernd Brügge

15-413 ***Software Engineering*** Fall 1999

21 October 1999

2 November 1999

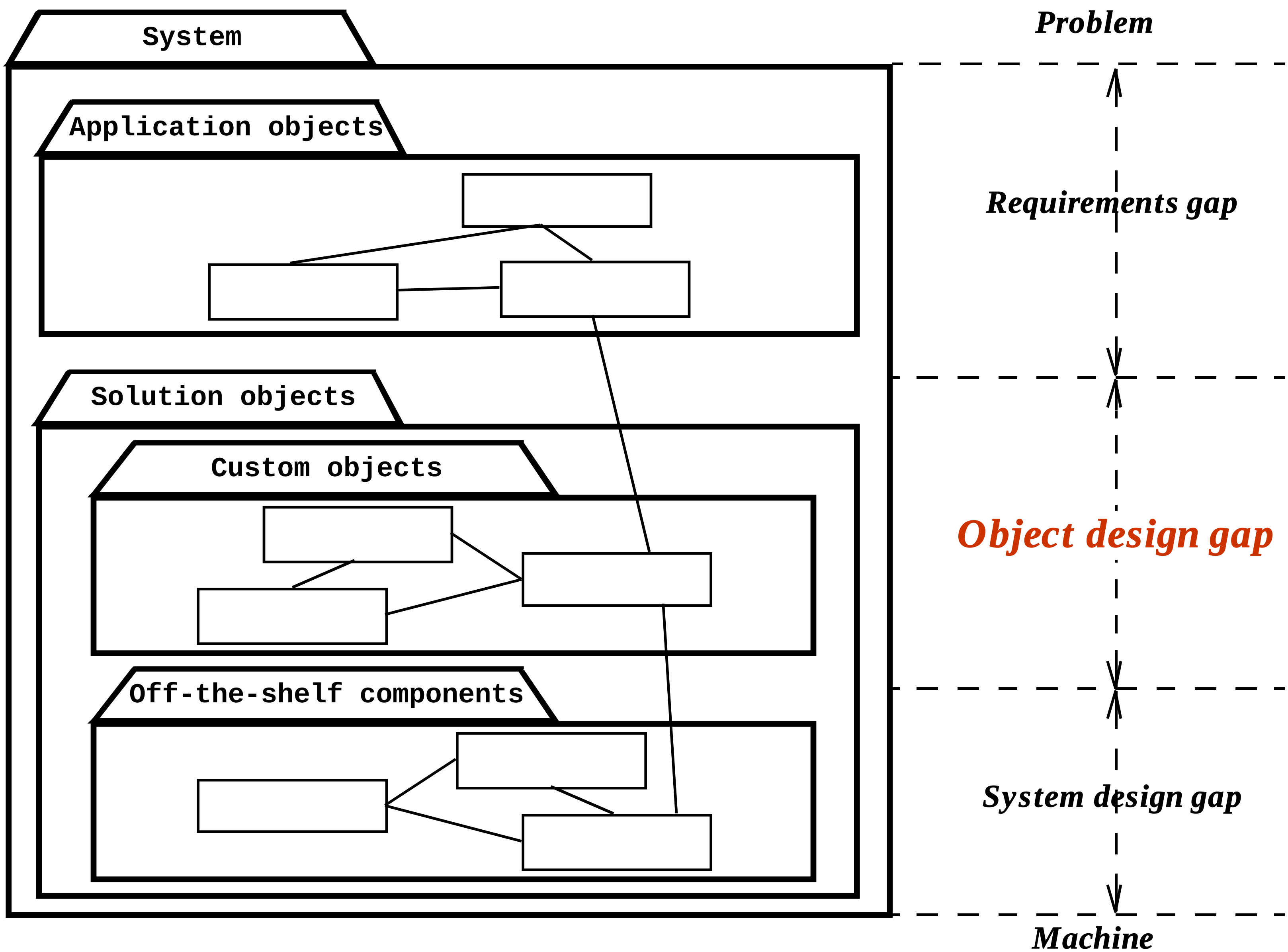
Object Design – The Basis for Implementation

- Object Design
  - **Add details to requrements analysis**
  - **Make implementation decisions: Build / Buy**
  - **Make implementation decisions: Solution objects**
- Design drivers
  - **Time. Cost. Quality / assurance.**
  - **Features / capability. Performance.**
  - **Future. Product line context.**
- Operations in the object model
  - **Requirements Analysis: Use cases, functional and dynamic models deliver operations for object model**
  - **Object Design: Iterate on where to put these operations in the object model**

Build / Buy

- Buy
  - Cost
    - Project resources
    - Development time
  - Market leverage
    - Ride growth curves
    - Quality
  - Adoption cost
- Build
  - Availability
  - Risk
    - Control over process
    - Future trajectory
    - Support: Small vendor
    - Support: Large vendor
    - Architectural
    - Certification

Object Design: Closing the Gap



Example Object Design Issues

- Full definition of associations
- Full definition of classes
- Encapsulation of algorithms and data structures
- Detection of new application-domain independent classes (example: Cache)
- Optimization
- Increase of inheritance
- Decision on control
- Packaging

Example Object Design Criteria

- Rate of change
  - **Of requirements**
  - **Of OTS environment**
- Relatedness within product line
- Performance sensitivity
- Division of labor, division of expertise
- Tolerance for interdependency

Object Design Activities

- 1. Service specification
  - **Describes precisely each class interface**
- 2. Component selection
  - **Identify off-the-shelf components and additional solution objects**
- 3. Object model restructuring
  - **Transforms the object design model to improve its understandability and extensibility**
- 4. Object model optimization
  - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

Object Design Concepts

- Solution Objects
  - **Objects in the implementation domain**
- Signatures
  - **Classes, Interfaces**
  - **Methods, Fields**
  - **Packages**
- Contracts
  - **Invariants, Preconditions, Postconditions**
  - **Mechanical attributes**
- Specification
  - **UML Object Constraint Language (OCL)**

Specification – 1

UML Object Constraint Language (OCL)

```
Context Hashtable inv:
  numElements >= 0

context Hashtable::put(key, entry) pre:
  !containsKey(key)

context Hashtable::put(key, entry) post:
  containsKey(key)                  and
  get(key) = entry                  and
  numElements = numElements@pre + 1
```

Specification – 2

- Examples
  - **Textual description of invariants, preconditions, postconditions.**
    - **Typical: Data invariant for complex types**
    - **Often used in code walkthroughs**
  - **State machine for a class**
    - **Lifetime of an object**
    - **How an object responds to events**
    - **Constraints on class clients: order of method calls, etc**
- In general
  - **Range from informal to formal**
- When is formal useful?
  - **Complex invariants and conditions**
  - **High risk: consequences of error**

1. Service Specification

- Requirements analysis
  - **Identifies attributes and operations**
    - **without specifying their types or their parameters.**
- Object design
  - **Identify missing attributes, operations**
  - **Specify details**

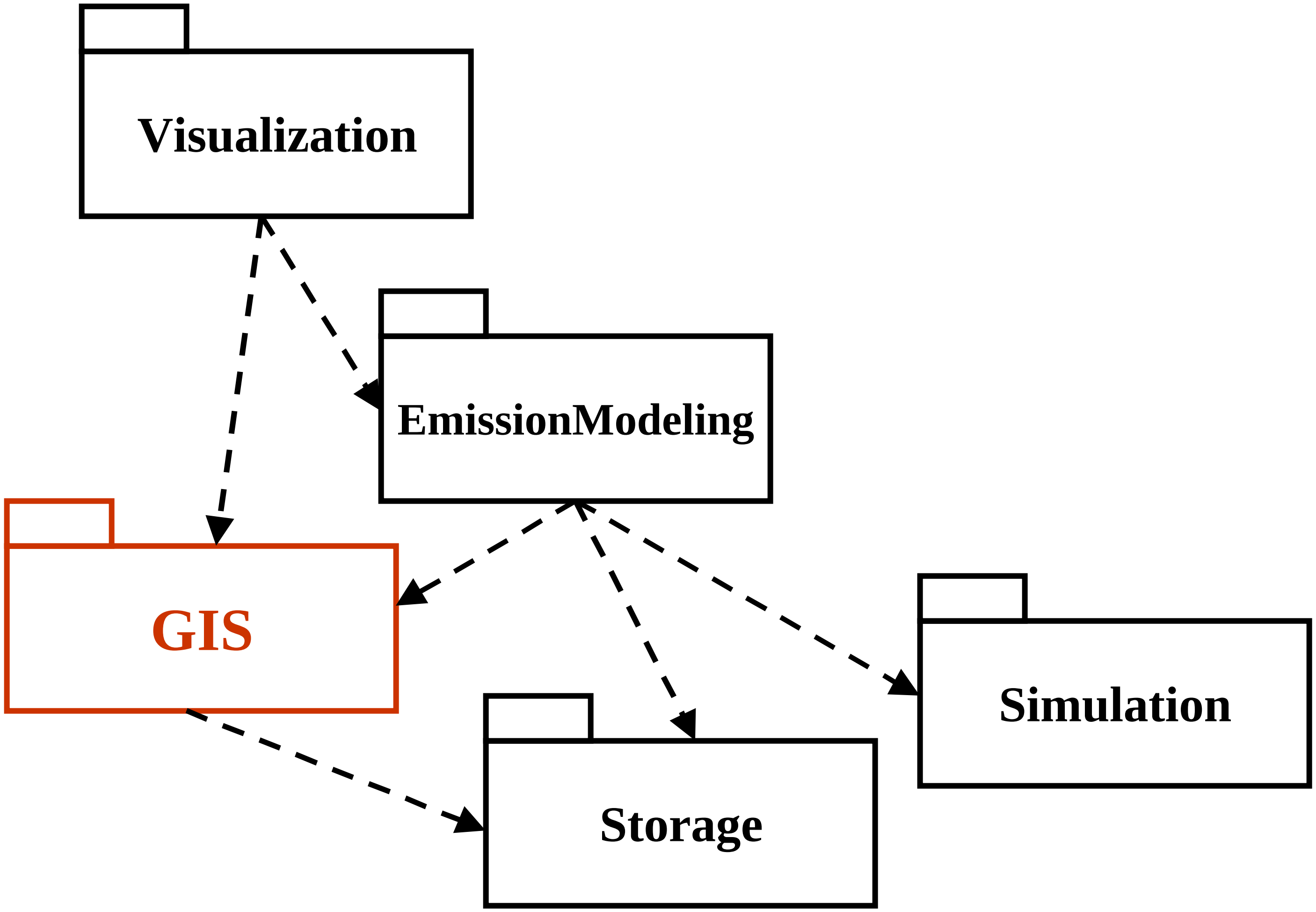
    1a. Type signature information

    1b. Visibility information

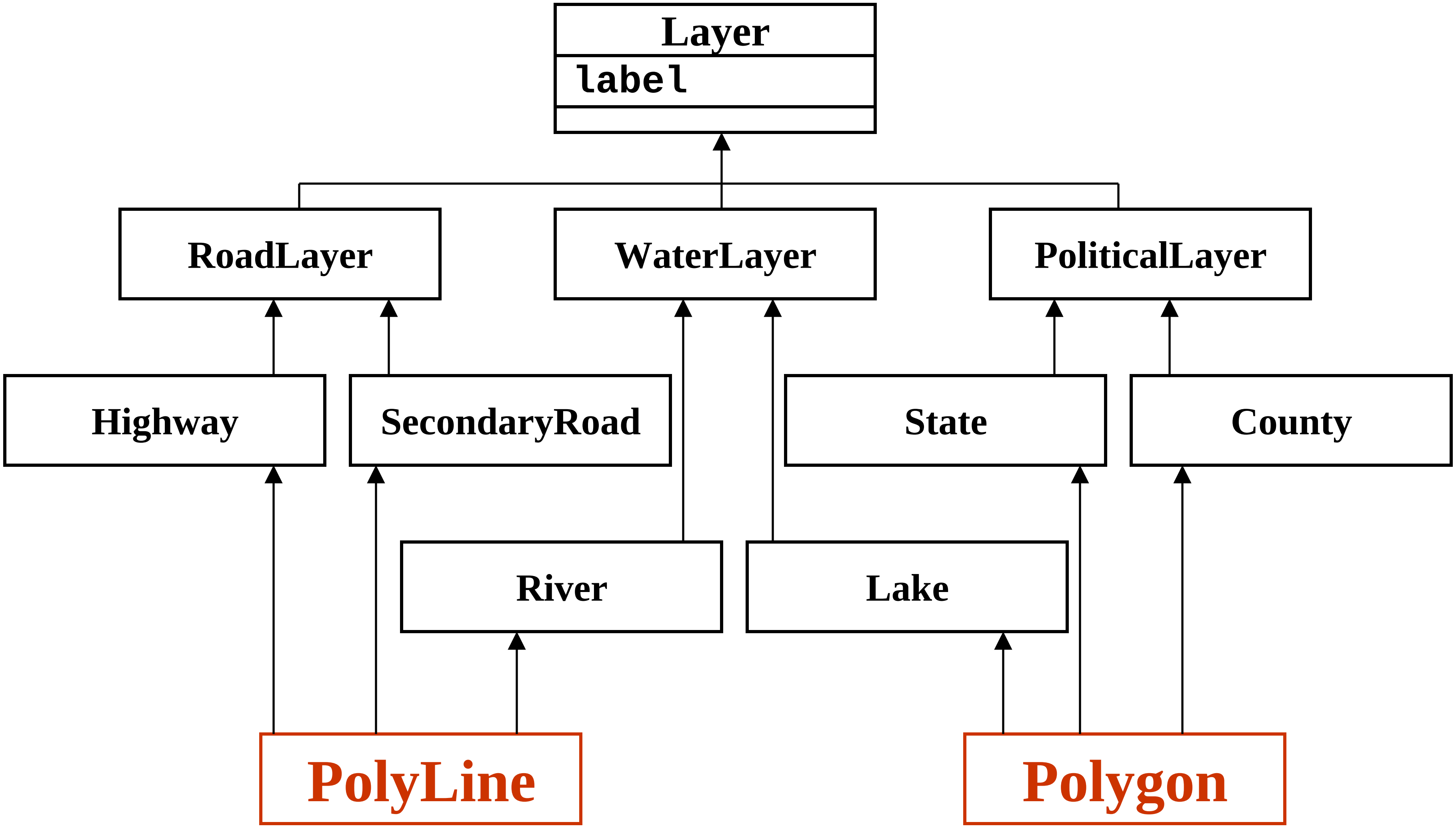
    1c. Contracts

    1d. Exceptions

GIS subsystems



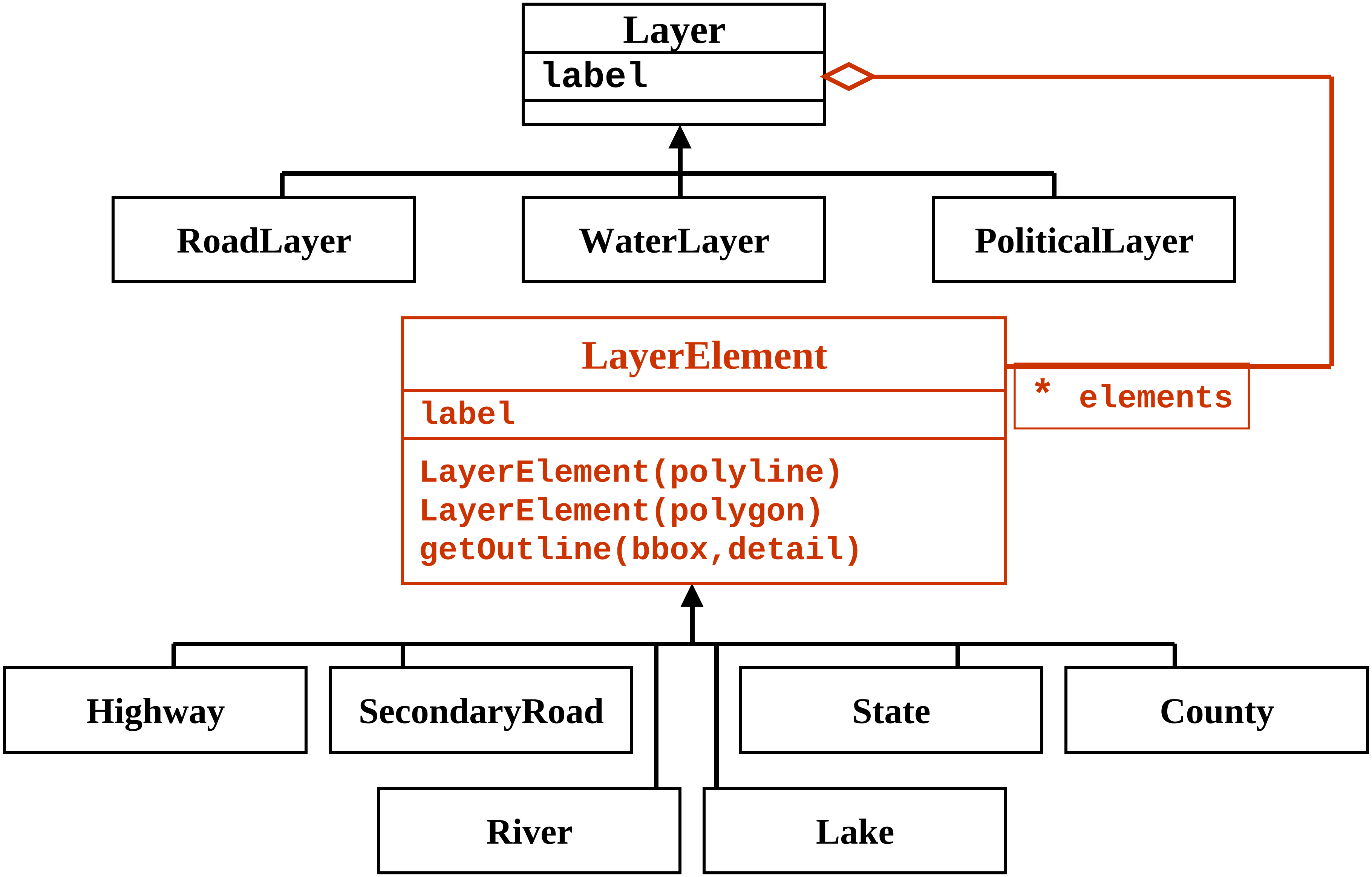
GIS Object Model: Before



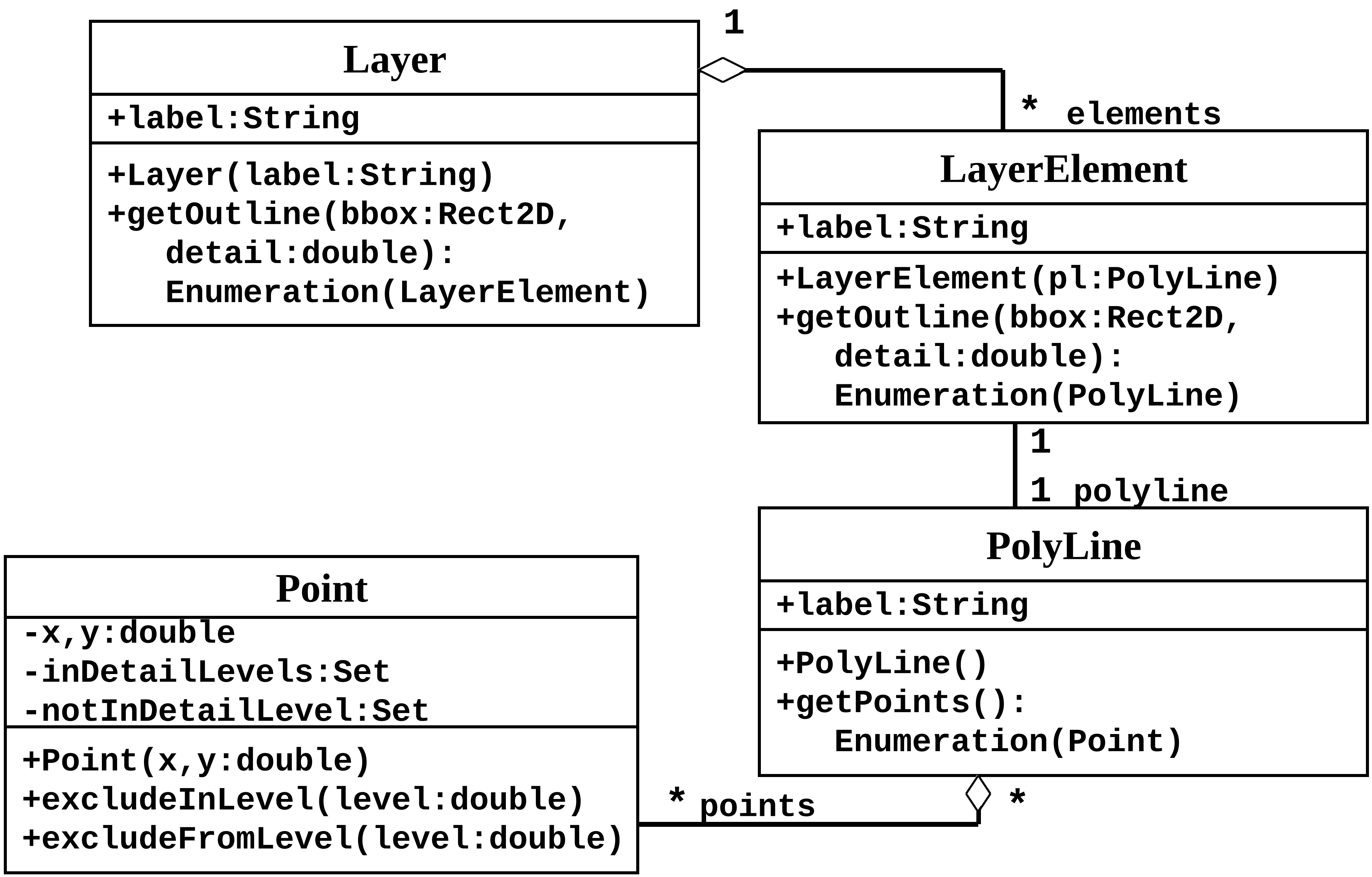
GIS: From System Design to Object Design

- Task specification
- Subsystem model
- Use case: *Zoom-the-Map*

GIS Object Model: After



GIS Object Model: After



1a. Service Specification: Add Type Signatures

```
Hashtable
-----------------
-numElements:int
-----------------
+put()
+get()
+remove()
+containsKey()
+size()
```

```
Hashtable
-----------------
-numElements:int
-----------------
+put(key:Object,entry:Object)
+get(key:Object):Object
+remove(key:Object)
+containsKey(key:Object):boolean
+size():int
```

1b. Service Specification: Add Visibility

UML defines three levels of visibility:

- Private (= Java `private`):
  - **Private attributes/operations can be accessed/invoked only within the class in which they are defined.**
  - **Private attributes/operations cannot be accessed by subclasses or other classes.**
- Protected (*nothing comparable* in Java):
  - **Protected attributes/operations can be accessed by the class in which they are defined and in any descendent class.**
- Public (= Java `public`):
  - **Public attributes/operations can be accessed by any class.**

Information Hiding : Why Encapsulate?

- Encapsulate
  - **Apply "need to know" principle.**
  - **The less an operation knows . . .**
    . . . the less likely it will be affected by any changes
    . . . the easier the class can be changed
- Build firewalls around classes
  - **Define public interfaces for classes as well as subsystems**
- The classic trade-off
  - **Information hiding vs. efficiency**
    - **Modularity vs. performance**

Information Hiding Design Principles

- Encapsulate attributes (fields)
  - **Only the operations of a class are allowed to manipulate its attributes**
  - **Access attributes only via operations.**
  - **Example: Java Beans**
  - **(Exception: static finals)**
- Encapsulate external objects at the subsystem boundary
  - **Define abstract class interfaces which mediate between system and external world as well as between subsystems**
- Use combiners
  - **Do not apply an operation to the result of another operation.**
  - **Write a new operation that combines the two operations.**

1c. Service Specification: Contracts

- Contracts on a class/method enable caller and callee to share assumptions about the class/method.
- Contracts include three types of constraints:
  - **Invariant: A predicate that is "always" true for instances of a class. Invariants are used to specify consistency constraints among class attributes.**
  - **Precondition: A predicate that must be true before a specific operation is invoked. A constraint on a caller.**
  - **Postcondition: A predicate that will be true after a specific operation has been invoked. A constraint on the operation.**

Specification

UML Object Constraint Language (OCL)

Truth-valued expressions. Not procedural.

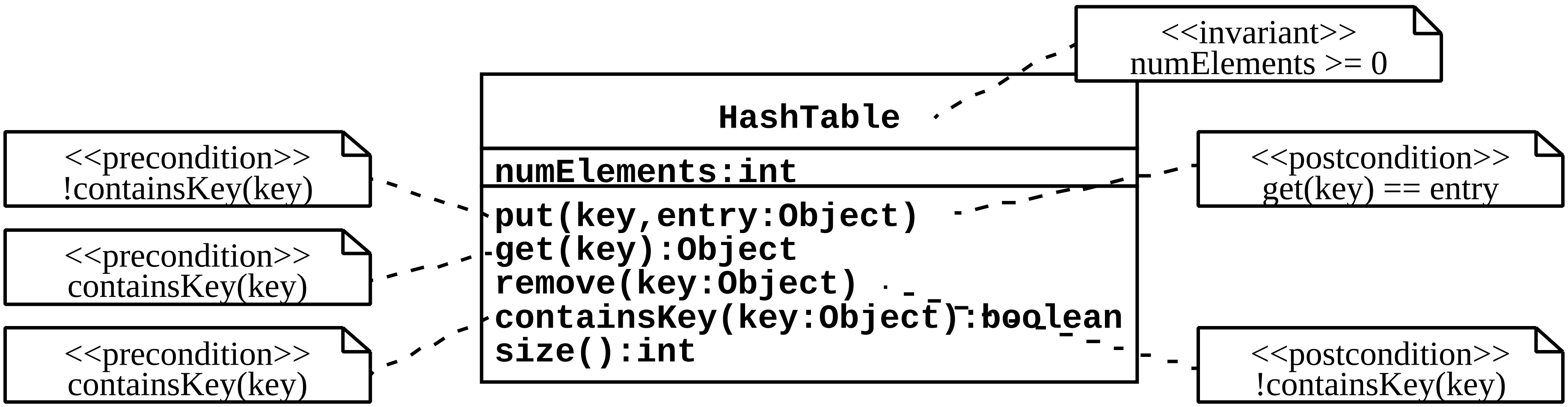
```
Context Hashtable inv:
  numElements >= 0

context Hashtable::put(key, entry) pre:
  !containsKey(key)

context Hashtable::put(key, entry) post:
  containsKey(key)                 and
  get(key) = entry                 and
  numElements = numElements@pre + 1
```

Specification and OCL

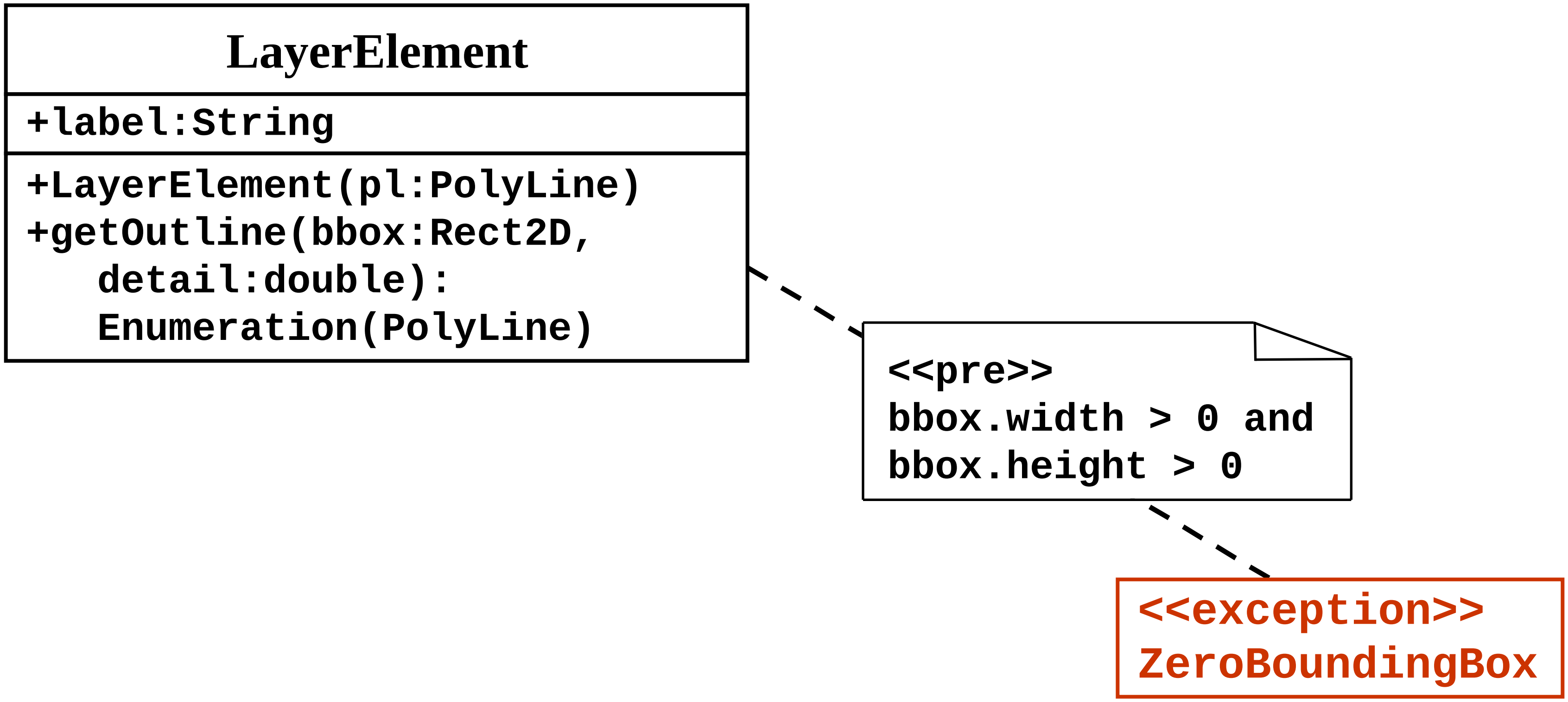
- A constraint can also be depicted as a note attached to the constrained UML element by a dependency relationship.



GIS Specification Example

```
context Point inv:
  Point.allInstances->forAll(p1,p2:Point |
  (p1.x=p2.x and p1.y=p2.y) implies p1=p2)
```

1d. Exceptions



Object Design Areas

- 1. Service specification
  - **Describes precisely each class interface**
- 2. Component selection
  - **Identify off-the-shelf components and additional solution objects**
- 3. Object model restructuring
  - **Transforms the object design model to improve its understandability and extensibility**
- 4. Object model optimization
  - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

2. Component Selection

The most important reuse decisions – OTS components

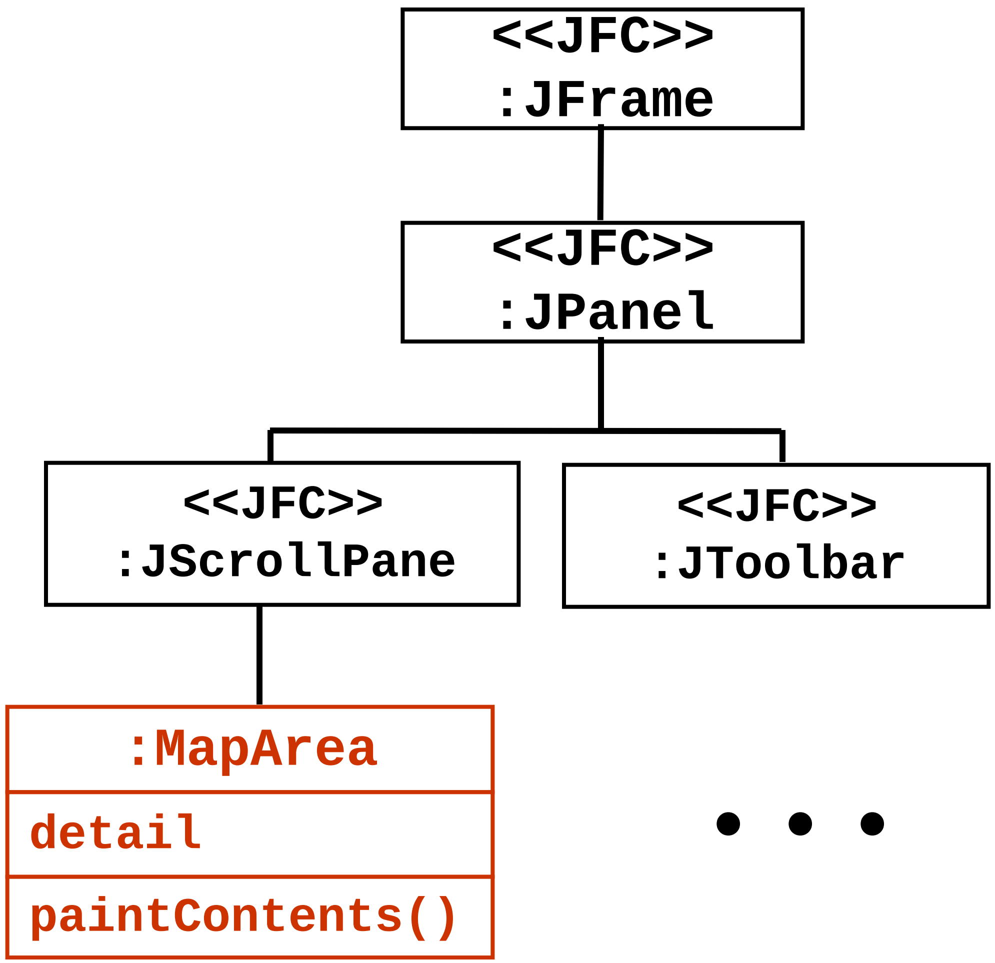
2a. Select and adjust **class libraries**

- **Examples: AWT, Swing/JFC, MFC, etc.**

2b. Select and adjust **application frameworks**

- **Examples: COM, Beans.**

Example: JFC adjustment



Issue: **Line representation**

JFC: `int[], int[]`

GIS: `Enum(Point)`

***Approaches*:**

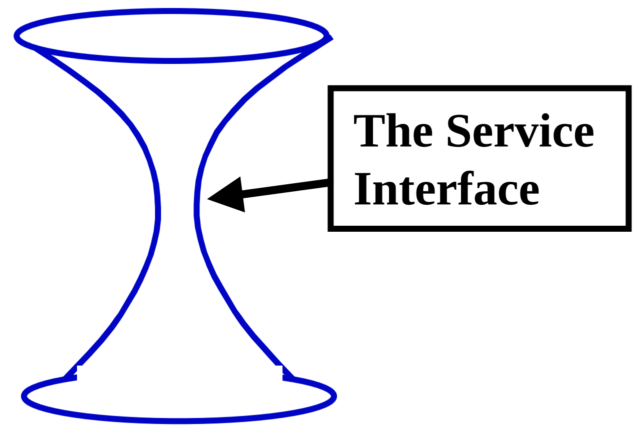
1. **Code translate method**
2. **Place the method:**
   - `MapArea`
   - `Layer`
   - **Adapter**

Application Frameworks

- *Example:* Java Beans
  - **Event model**
    - **Serialization**
    - **Locks**
  - **Fields**
    - **Private**
    - **Nomenclature**
  - **Introspection**
  - **Persistence**
    - **Customization**
  - ***Etc.***

- Frameworks
- Patterns
- Libraries
- Components

The API Hourglass



Object Design Areas

- 1. Service specification
  - **Describes precisely each class interface**
- 2. Component selection
  - **Identify off-the-shelf components and additional solution objects**
- 3. Object model restructuring
  - **Transforms the object design model to improve its understandability and extensibility**
- 4. Object model optimization
  - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

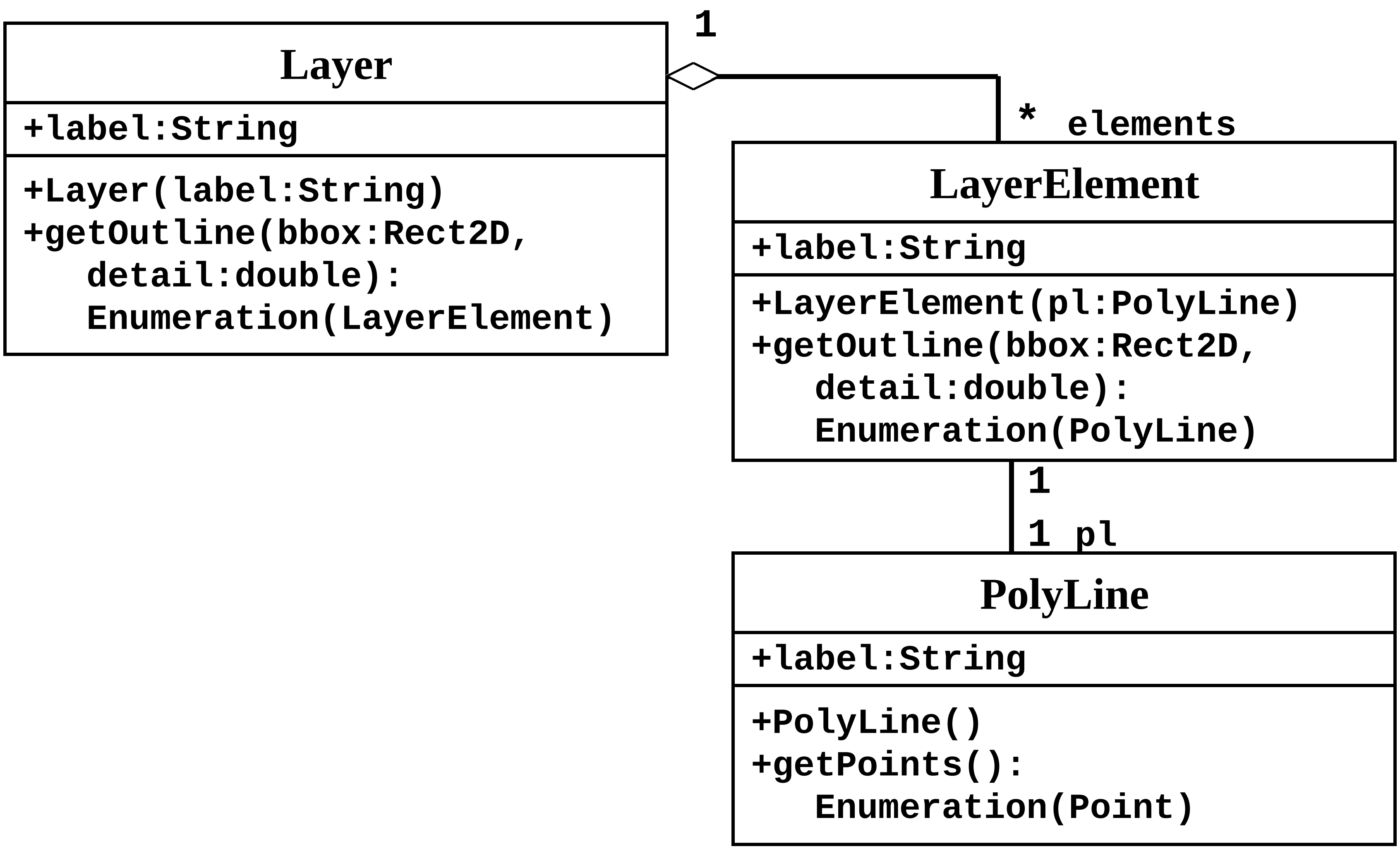
3. Restructuring Activities

3a. Realize **associations**

3b. Increase **reuse**

3c. Remove implementation **dependencies**

3a. Realize Associations

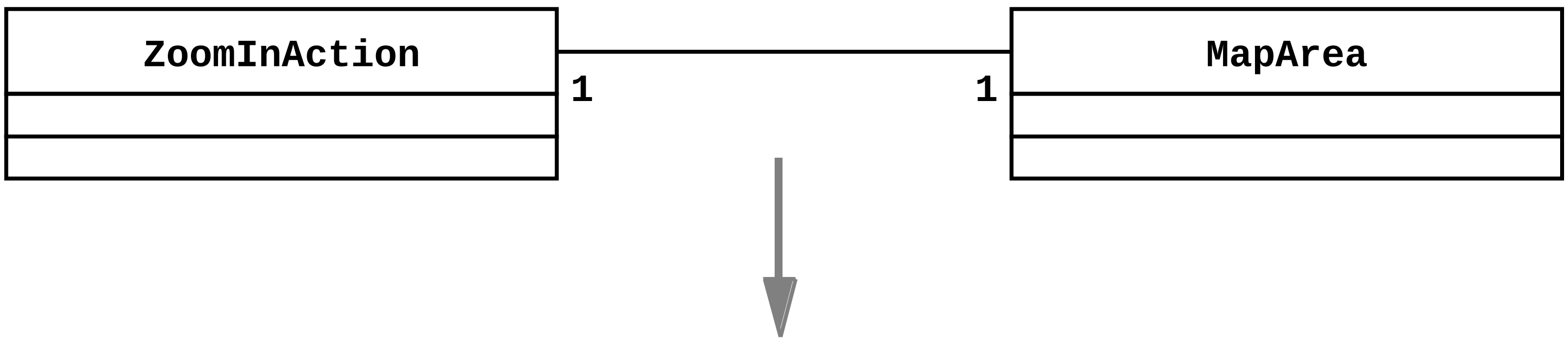


3a. Realize Associations

- Kinds / Dimensions
  - **1-1, 1-many, many-many**
  - **0..1**
  - **Uni-/bi-directional**
  - **Qualified**
  - **Visibility**
- **Considerations**
  - **Operations needed**
  - **Performance**
  - **References**
- **Implementation decisions**
  - **Collections**
  - **Use of separate objects**

Unidirectional 1-to-1 Association

Object design model before transformation



Object design model after transformation

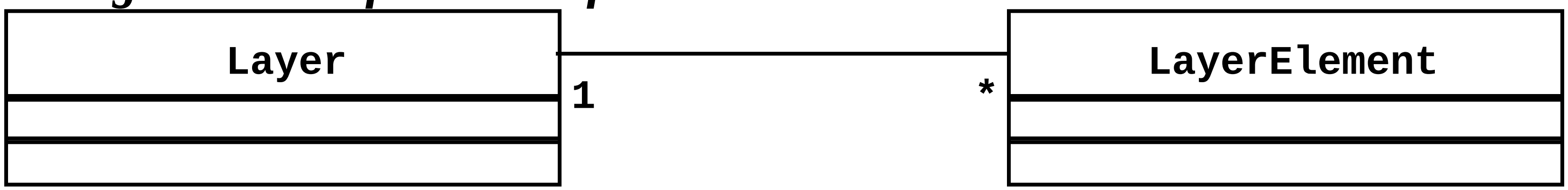
Bidirectional 1-to-1 Association

Object design model before transformation

Object design model after transformation

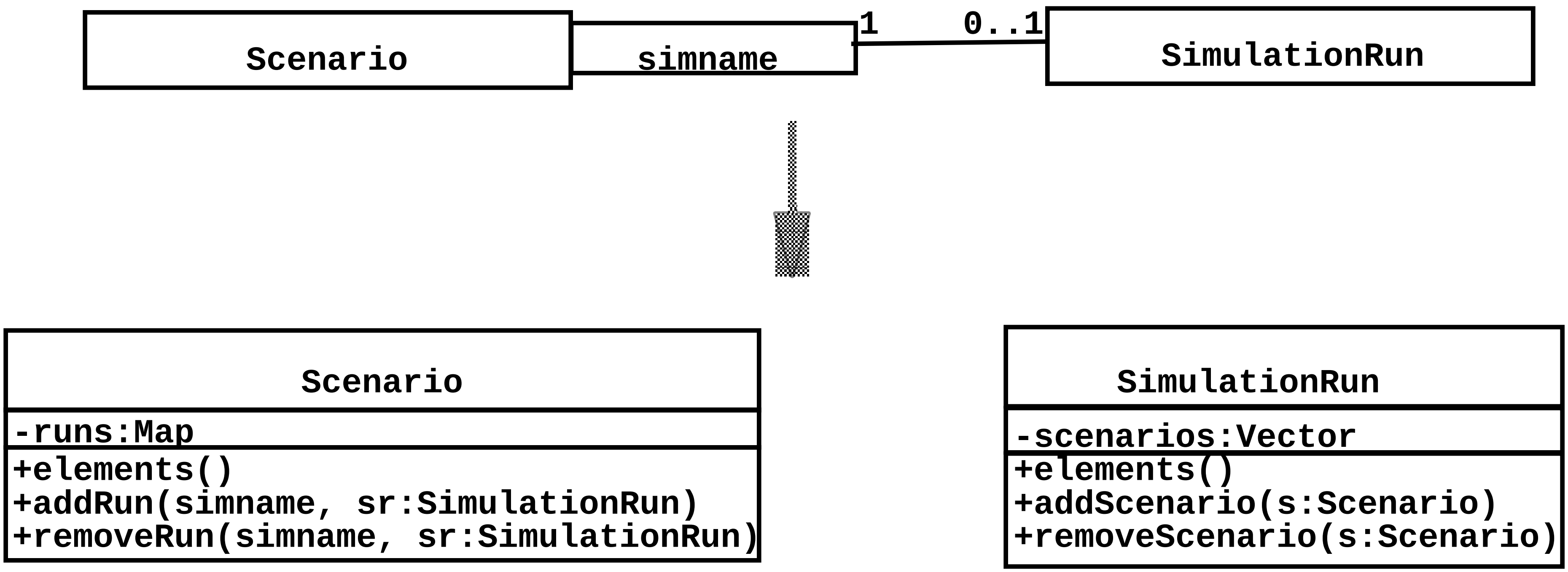
1-to-Many Association

Object design model before transformation



Object design model after transformation

Qualification



3b. Increase Reuse

Increase Inheritance

- In general:
  - **Taxonomy reflects understanding of application domain**
- Rearrange and adjust classes/operations for inheritance
- Abstract common behavior from class groups
  - **Detect opportunities to “hoist” behaviors**
- A subsystem could become a superclass.

The cost of inheritance

- Performance: dynamic dispatch
- Recompilation

Building a super class from several classes

- Prepare for inheritance. Refactor operations to have similar signatures:
  - **Fewer/mismatched arguments: Overload names**
  - **Mismatched attribute names: Rename attribute, change operations.**
- Abstract out the common behavior into supers
  - **Supers are desirable.**
    - **Better modularity, extensibility and reusability**
    - **Improved configuration management**

Inheritance: Good and Bad

- **Kinds of Inheritance**
  - **Interfaces**
  - **Implementations**
- **The cost of implementation inheritance**
  - ***Code rot:*** **Persistence of abstractions beyond their time**
  - **Dependency on small local implementation decisions**
    - **Data representations and invariants**
  - ***Example*: Jframe and JInternalFrame**
- **Approaches**
  - **Delegate, don't inherit**
    - **"Subclass" delegates to the "super"**
  - **Use abstract supers**
  - **Proliferate interfaces**

Refactoring

- ***Reorganizing hierarchies, signatures, and collaborations***
  - **Without changing overall system function (generally)**
- Examples
  - **Moving from one taxonomy to another taxonomy**
  - **Multiple inheritance to single inheritance**
  - **Relocating methods within a hierarchy**
  - **Renaming to exploit overloading**
    - **I.e., hierarchy transparency at code level**

Object Design Areas

- 1. Service specification
  - **Describes precisely each class interface**
- 2. Component selection
  - **Identify off-the-shelf components and additional solution objects**
- 3. Object model restructuring
  - **Transforms the object design model to improve its understandability and extensibility**
- 4. Object model optimization
  - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

Design Optimizations

- Important part of the object design phase:
  - **Requirements analysis model is semantically correct but often too inefficient if directly implemented.**
- Optimization activities during object design:
  1. Add redundant associations to minimize access cost
  2. Rearrange computations for greater efficiency
  3. Store derived attributes to save recomputation time
- As an object designer you must strike a balance between efficiency and clarity.
  - **Optimizations will make your models more obscure**

Design Optimizations

- Necessary part of the object design phase:
  - **Requirements analysis model –**
    - **Semantically correct**
    - **Well structured (modular)**
    - **Probably too inefficient if directly implemented.**
  - **Object designer: balance between efficiency and clarity.**
    - **Efficiency: At *run time*, *compile time*, *design time*.**
    - **Optimizations will make *models* more obscure**
    - **Optimizations will make *programs* harder to evolve**

4. Design Optimization Activities

4a. Add redundant associations to reduce access cost

- What are the most frequent operations?
  - Sensor data lookup?
- How often is the operation called?
  - 30 times/month?   20 times/second?

4b. Turn classes into attributes to avoid recomputation

- Eliminate unnecessary abstraction structure

4c. Cache expensive results

- But can cache coherency be maintained?

4d. Compute lazily

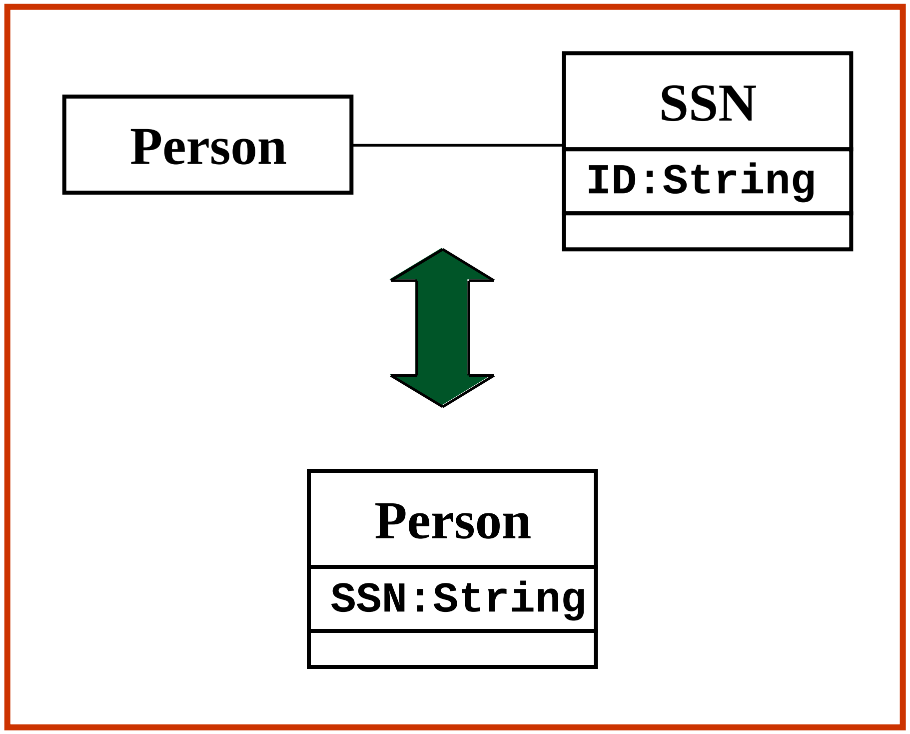
- Delay expensive operations

4a. Add Redundant Associations

- More generally
  - **Replace bidirectional by unidirectional**
  - **Replace many-many by 1-many**
  - **Replace 1-many by 1-1**
  - **Add additional associations**
- ***Cache***
- When to do this?
  - **What is the frequency of traversal?**
  - **What is the relative cost of traversal and the operation performed?**
  - **Can search be replaced by indexing?**
    - **E.g., order or hash objects**
  - **Can indexing be replaced by direct reference?**
    - **E.g., cache references**

4b. Collapse Objects

- Can this association be an attribute?
- Can this object be an attribute of another object?
  - **Object design choices:**
    - **Implement entity as embedded attribute**
    - **Implement entity as separate class with associations to other classes**
- Associations
  - **More flexible than attributes**
  - **Can introduce unnecessary indirection**



4c. Cache Expensive Results

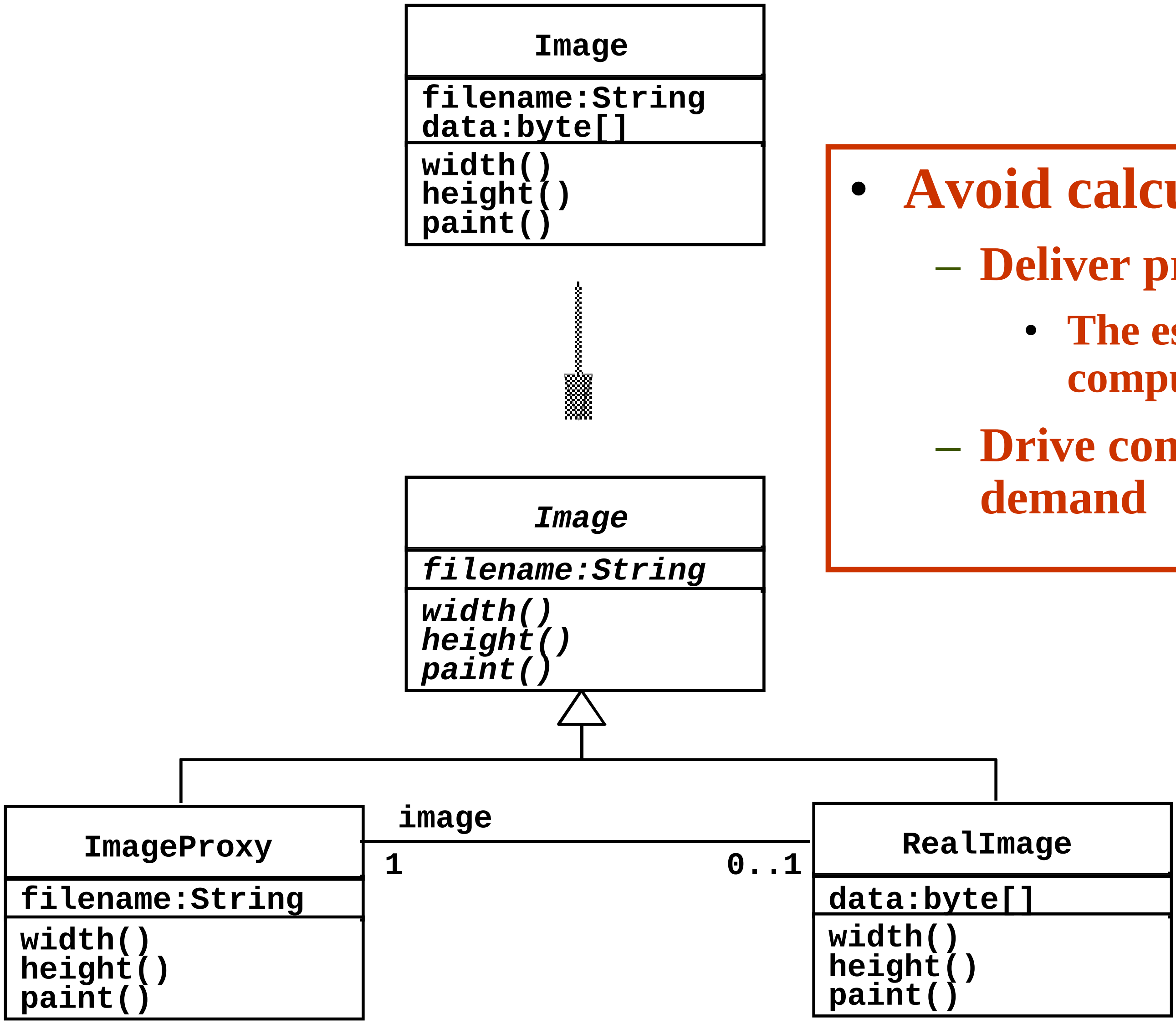
Example: How to deal with disconnection in distributed systems?

- Store derived attributes
  - **Example: Define new classes to store information locally**
    - **Database cache**

Issues:

- Derived attributes must be updated when base values change.
  - Cache coherency
- Storage costs can increase
- Approaches to the update problem:
  - ***Periodic computation* –** Deliberately recompute at intervals (inexact)
  - ***Explicit linking* –** Modified MVC. Active value. Consistency check.

4d. Delay Complex Computations



- Avoid calculating results
  - Deliver proxies instead
    - The essence of lazy computation
  - Drive computation by demand

The Object Design Document (ODD), 1

- Object design document
  - **Similar to RAD + ...**

    … + Additions to object, functional and dynamic models
    (from solution domain)

    … + Navigational map for object model

    … + Javadoc documentation for all classes

ODD Conventions

- Each subsystem in a system provides a service
  - **Describe the set of operations provided by the subsystem**
- Specifying a service operation as
  - **Signature: Name of operation, fully typed parameter list and return type**
  - **Abstract: Describes the operation**
  - **Pre: Precondition for calling the operation (where appropriate)**
  - **Post: Postcondition describing important state after the execution of the operation (where appropriate)**
- Use JavaDoc for the specification of service operations.

The Object Design Document (ODD), 2

- **ODD Management issues**
  - **Update the RAD models in the RAD?**
  - **Should the ODD be a separate document?**
  - **Target audience for these documents**
    - **Customer?**
    - **Developer?**
    - **Remote team?**
  - **If time is short:**
    - **Focus on the Navigational Map and Javadoc documentation**

- **Example of acceptable ODD:**
  - ***[** to be provided **]***

Packaging

- Package design into discrete **physical units** that can be edited, compiled, linked, reused:
  - **Ideally one package per subsystem**
  - **But: system decomposition might not favor implementation.**

- Design principles
  - ***Minimize coupling:***
    - **Client-supplier relationships**
    - **Limit number of parameters**
    - **Avoid global data**
  - ***Maximize cohesiveness:***
    - **Tights associations *imply* same package**
  - ***Consider the number of interface objects offered***
    - ***Interface object* :**
      - **Denotes a service or API**
      - **For requirements analysis, system/object design.**
    - ***Java interface*:**
      - **Implements an interface object, or not.**