Lecture Notes on Object Design

Bernd Brügge
Lehrstuhl für Angewandte Softwaretechnik
Technische Universität München

1 February 2002

Object Design

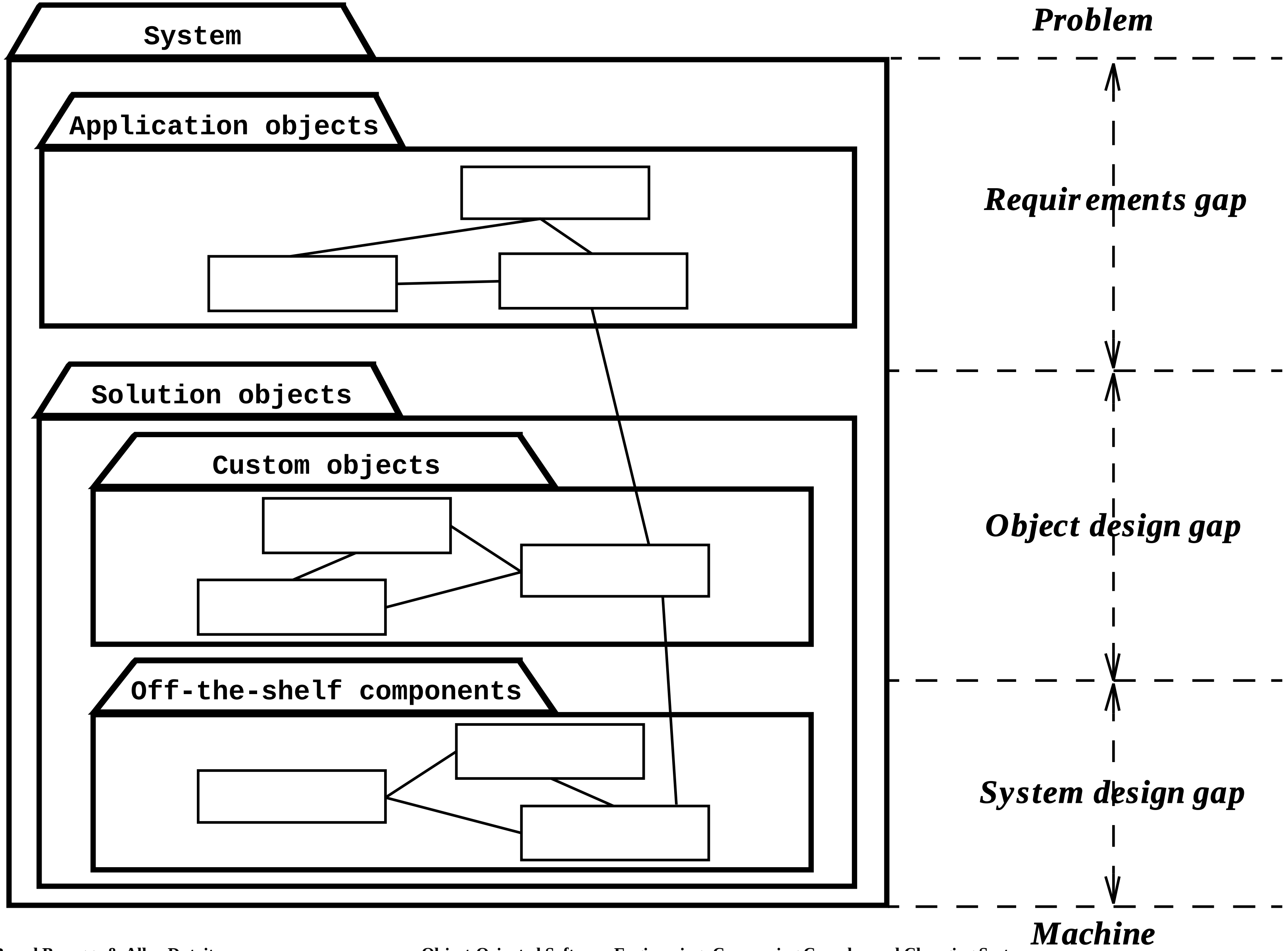
- Object design is the process of adding details to the requirements analysis and making implementation decisions
- The object designer must choose among different ways to implement the analysis model with the goal to minimize execution time, memory and other measures of cost.
  - **Requirements Analysis: The functional model and the dynamic model deliver operations for the object model**
  - **Object Design: We decide on where to put these operations in the object model**
- Object Design serves as the basis of implementation

Odds and Ends

- Next week:
  - **Thursday February 7: Client Acceptance Test**
  - **Friday February 8: Last lecture**
- Exam:
  - **Thursday February 14, 14:30-16:00**
  - **Open Book**

- Interested in more software engineering?
  - **Diploma thesis in the USA**
  - **Advanced Project Management Seminar**
  - **Software Engineering II**
  - **Softwaretechnik Praktikum Augmented Reality (Prof. Gudrun Klinker)**

Object Design: Closing the Gap



Object Design Issues

- Full definition of associations
- Full definition of classes
- Choice of algorithms and data structures (Info I)
- Detection of new application-domain independent classes (example: Cache, Cash)
- Optimization (also called “Refactoring”)
- Increase of inheritance (Generalization, Specialization)
- Decision on control
- Packaging

Terminology of Activities

- Object-Oriented Methodologies
  - ***System Design***
    - **Decomposition into subsystems**
  - ***Object Design***
    - **Choice of implementation language (not earlier!)**
    - **Choice of data structures and algorithms**
- SA/SD (structured analysis/structured design) uses different terminology:
  - ***Preliminary Design***
    - **Decomposition into subsystems**
    - **Choice of data structures**
  - ***Detailed Design***
    - **Choice of algorithms**
    - **Data structures are refined**
    - **Choice of implementation language**
    - **Typically in parallel with preliminary design, not a separate phase**

Object Design Activities

- 1. Service specification
  - **Describes precisely each class interface**
  - **Class builder (implementor), Class user, Class extender**
- 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 and reusability**
- 4. Object model optimization
  - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

Service Specification

- Requirements analysis
  - **Identifies attributes and operations without specifying their types or their parameters.**
- Object design
  - **Adds visibility information**
  - **Adds type signature information**
  - **Adds contracts**

Add Visibility

UML defines three levels of visibility:

- Private (Class implementor):
  - **A private attribute can be accessed only by the class in which it is defined.**
  - **A private operation can be invoked only by the class in which it is defined.**
  - **Private attributes and operations cannot be accessed by subclasses or other classes.**
- Protected (Class extender):
  - **A protected attribute or operation can be accessed by the class in which it is defined and on any descendent of the class.**
- Public (Class user):
  - **A public attribute or operation can be accessed by any class.**

Information Hiding Heuristics

- Carefully define the public interface for classes as well as subsystems (façade)
- Always apply the "Need to know" principle.
  - **Only if somebody needs to access the information, make it publicly possible, but then only through well defined channels, so you always know the access.**
- The fewer an operation knows
  - **the less likely it will be affected by any changes**
  - **the easier the class can be changed**
- Trade-off: Information hiding vs efficiency
  - **Accessing a private attribute might be too slow (for example in real-time systems or games)**

Information Hiding Design Principles

- Only the operations of a class are allowed to manipulate its attributes
  - **Access attributes only via operations.**
- Hide external objects at subsystem boundary
  - **Define abstract class interfaces which mediate between system and external world as well as between subsystems**
- Do not apply an operation to the result of another operation.
  - **Write a new operation that combines the two operations.**

Add Type Signature Information

```
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
```

Design by Contract

- Contracts on a class enable caller and callee to share the same assumptions about the class. Contracts include three types of constraints:
- Invariant:
  - **A predicate that is always true for all instances of a class. Invariants are constraints associated with classes or interfaces.**
- Precondition:
  - **Preconditions are predicates associated with a specific operation. A predicate that must be true before the operation is invoked. Preconditions are used to specify constraints that a caller must meet before calling an operation.**
- Postcondition:
  - **Postconditions are predicates associated with a specific operation. A predicate must be true after an operation is invoked. Postconditions are used to specify constraints that the object must ensure after the invocation of the operation.**

Expressing constraints in UML

- OCL (Object Constraint Language)
  - **OCL allows constraints to be formally specified on single model elements or groups of model elements**
  - **A constraint is expressed as an OCL expression returning the value true or false. OCL is not a procedural language (cannot constrain control flow).**
- OCL expressions for Hashtable operation put():
  - **Invariant:**
    - **context Hashtable inv: numElements >= 0**

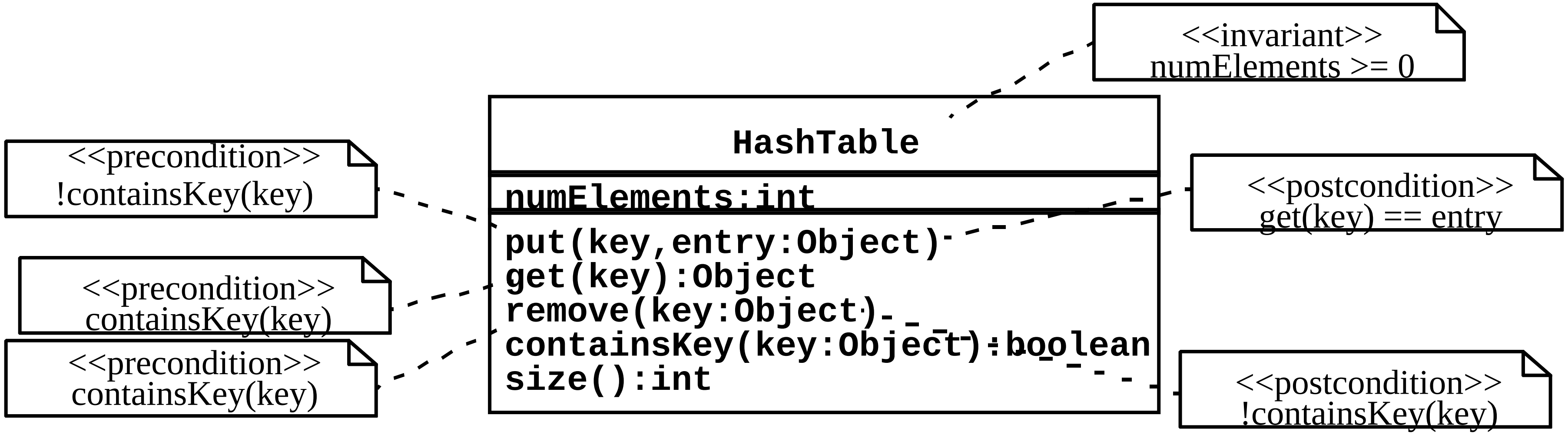
OCL expression

Context is a class operation put

  - **Precondition:**
    - **context Hashtable::put(key, entry) pre:!containsKey(key)**
  - **Post-condition:**
    - **context Hashtable::put(key, entry) post: containsKey(key) and get(key) = entry**

Expressing Constraints in UML

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



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.**

Component Selection

- Search for existing
  - **off-the-shelf class libraries,**
  - **frameworks or**
  - **components**
- Adjust the class libraries, framework or components
  - **Change the API if you have access to the source code.**
  - **Use the adapter or bridge pattern if you don't have access**
- Architecture-driven Design

Reuse...2/1/02

- **Look for existing classes in class libraries**
  - **JSAPI, JTAPI, ...**
- Select data structures appropriate to the algorithms
  - **Container classes**
  - **Arrays, lists, queues, stacks, sets, trees, ...**
- It might be necessary to define new internal classes and operations
  - **Complex operations defined in terms of lower-level operations might need new classes and operations**

Odds and Ends

- February 14, 14:30-16:00, exam for
  - **Bachelors, Masters in CE, Exchange Students, ...**
- Scope:
  - **Chapters 1-9 in the book**
  - **Material presented in the lecture**
  - **Tools (e.g., REQuest, CVS, Notes) are *NOT* part of the exam**
- Open book:
  - **Bring anything you want**
  - **However, your notes, the book, and the slides are all what you need.**

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 (refactoring)
  - **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.**

Restructuring Activities

- Revisiting inheritance to increase reuse
- Revising inheritance to remove implementation dependencies
- Realizing associations

Increase Inheritance

- Rearrange and adjust classes and operations to prepare for inheritance
  - **Generalization: Finding the base class first, then the sub classes.**
  - **Specialization: Finding the the sub classes first, then the base class**
- Generalization is a common modeling activity. It allows to abstract common behavior out of a group of classes
  - **If a set of operations or attributes are repeated in 2 classes the classes might be special instances of a more general class.**
- Always check if it is possible to change a subsystem (collection of classes) into a superclass in an inheritance hierarchy.

Generalization: Building a super class from several classes

- You need to prepare or modify your classes for generalization.
  - **All operations must have the same signature**
- Often the signatures do not match:
  - **Some operations have fewer arguments than others: Use overloading (Possible in Java)**
  - **Similar attributes in the classes have different names: Rename attribute and change all the operations.**
  - **Operations defined in one class but no in the other: Use abstract methods and method overriding.**
- Superclasses are desirable. They
  - **increase modularity, extensibility and reusability**
  - **improve configuration management**
- Many design patterns use superclasses
  - **Try to retrofit an existing model to use a design pattern**

Implement Associations

- Two strategies for implementing associations:
  1. **Be as uniform as possible**
  2. **Make an individual decision for each association**
- Example of a uniform implementation (often used by CASE tools)
  - **1-to-1 association:**
    - **Role names are treated like attributes in the classes and translate to references**
  - **1-to-many association:**
    - **Always Translate into a Vector**
  - **Qualified association:**
    - **Always translate into to Hash table**

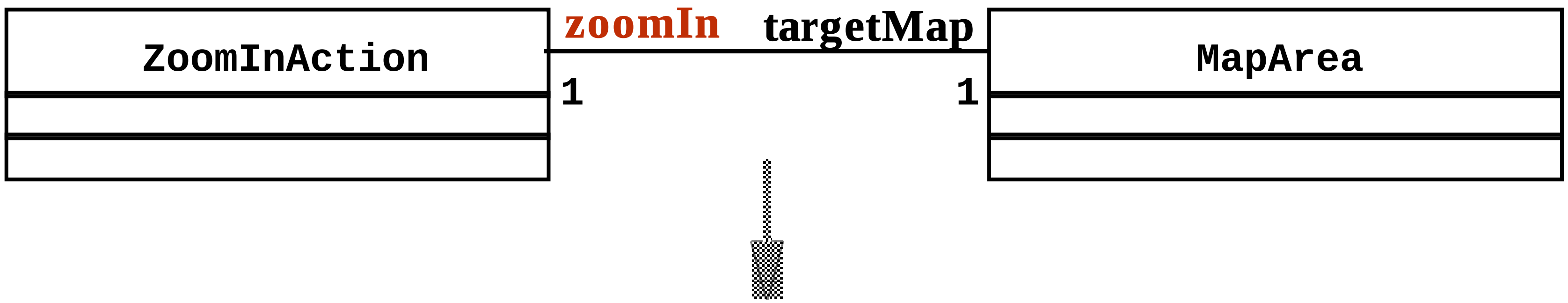
Unidirectional 1-to-1 Association

Object design model befor e transformation

Object design model after transformation

Bidirectional 1-to-1 Association

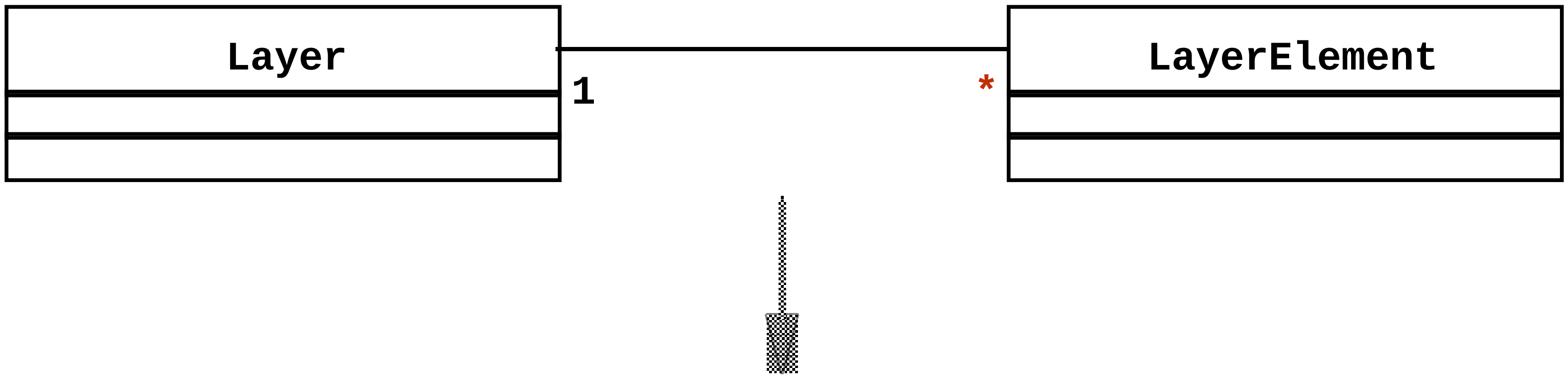
Object design model before transformation



Object design model after transformation

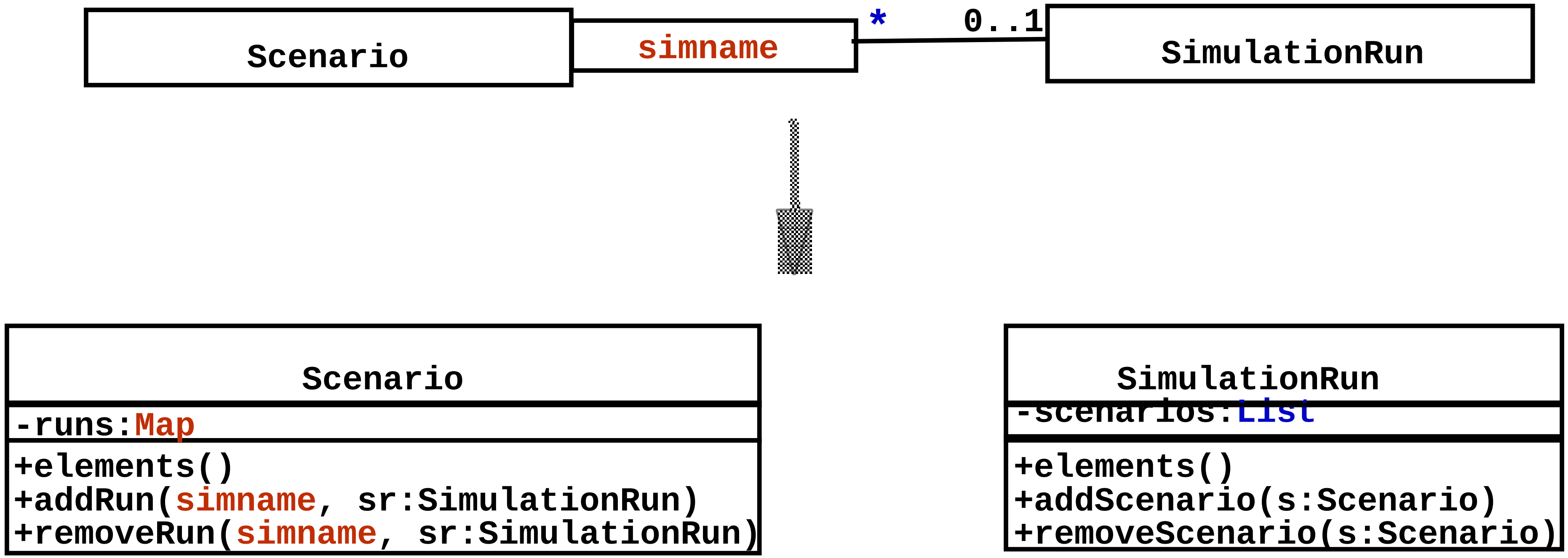
1-to-Many Association

Object design model before transformation



Object design model after transformation

Qualification



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

- Design optimizations are an important part of the object design phase:
  - **The requirements analysis model is semantically correct but often too inefficient if implemented directly.**
- 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**
- A note of caution:
  - ***Only* optimize when needed.**
  - **Perform optimizations based on an *actual* profile.**

Design Optimization Activities

1. Add redundant associations *(data flow optimization)*
   - **What are the most frequent operations? (Sensor data lookup?)**
   - **How often is the operation called? (once times a month, every 50 msec)**

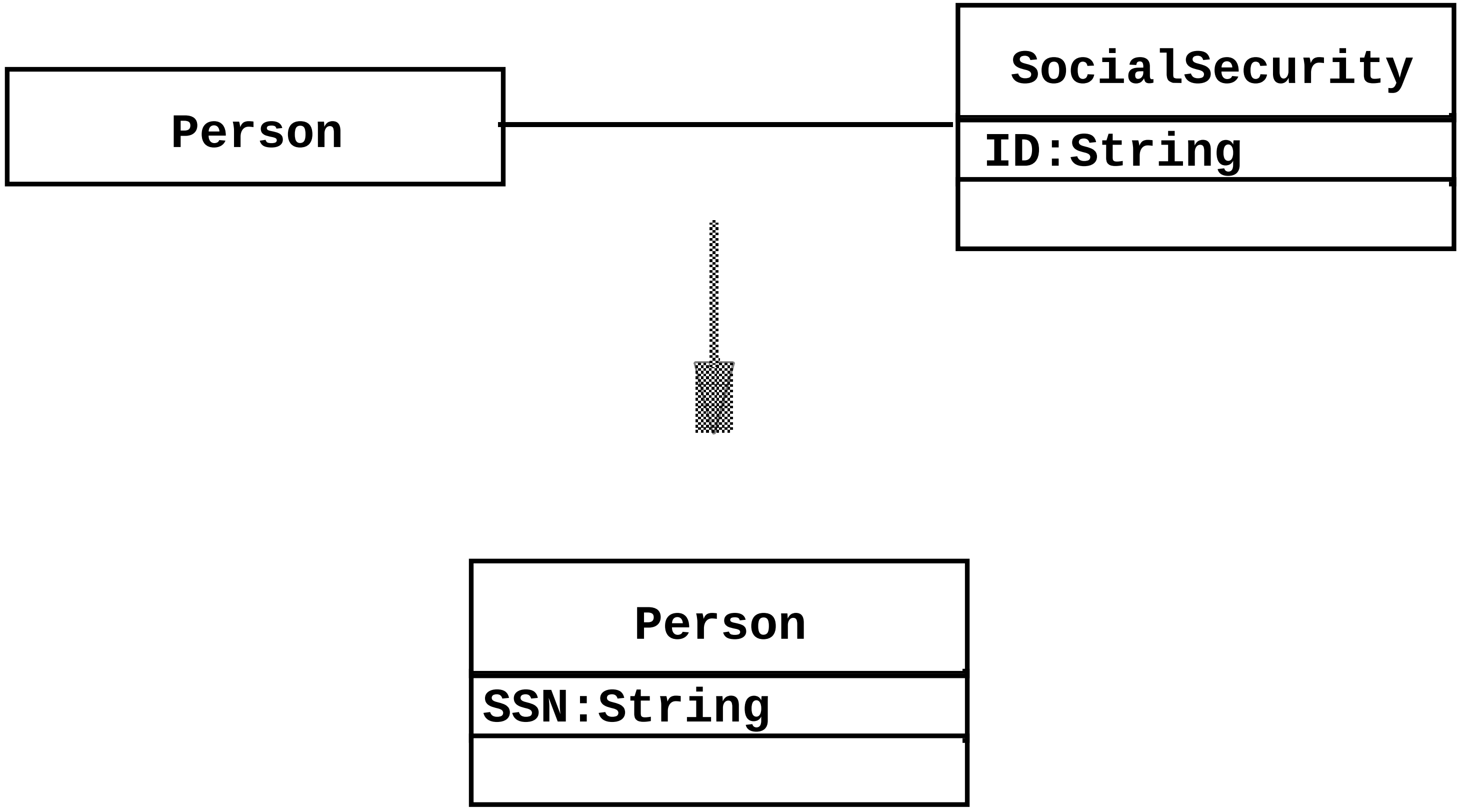
2. Rearrange execution order *(control flow optimization)*
   - **Eliminate dead paths as early as possible (Use knowledge of distributions, frequency of path traversals)**
   - **Narrow search as soon as possible**
   - **Check if execution order of loop should be reversed**

- 3. Turn classes into attributes *(object optimization)*

Implement Application domain classes

- To collapse or not collapse: Attribute or association?
- Object design choices:
  - **Implement entity as embedded attribute**
  - **Implement entity as separate class with associations to other classes**
- Associations are more flexible than attributes but often introduce unnecessary indirection.

Optimization Activities: Collapsing Objects



To Collapse or not to Collapse?

- Collapse a class into an attribute if the only operations defined on the attributes are Set() and Get().

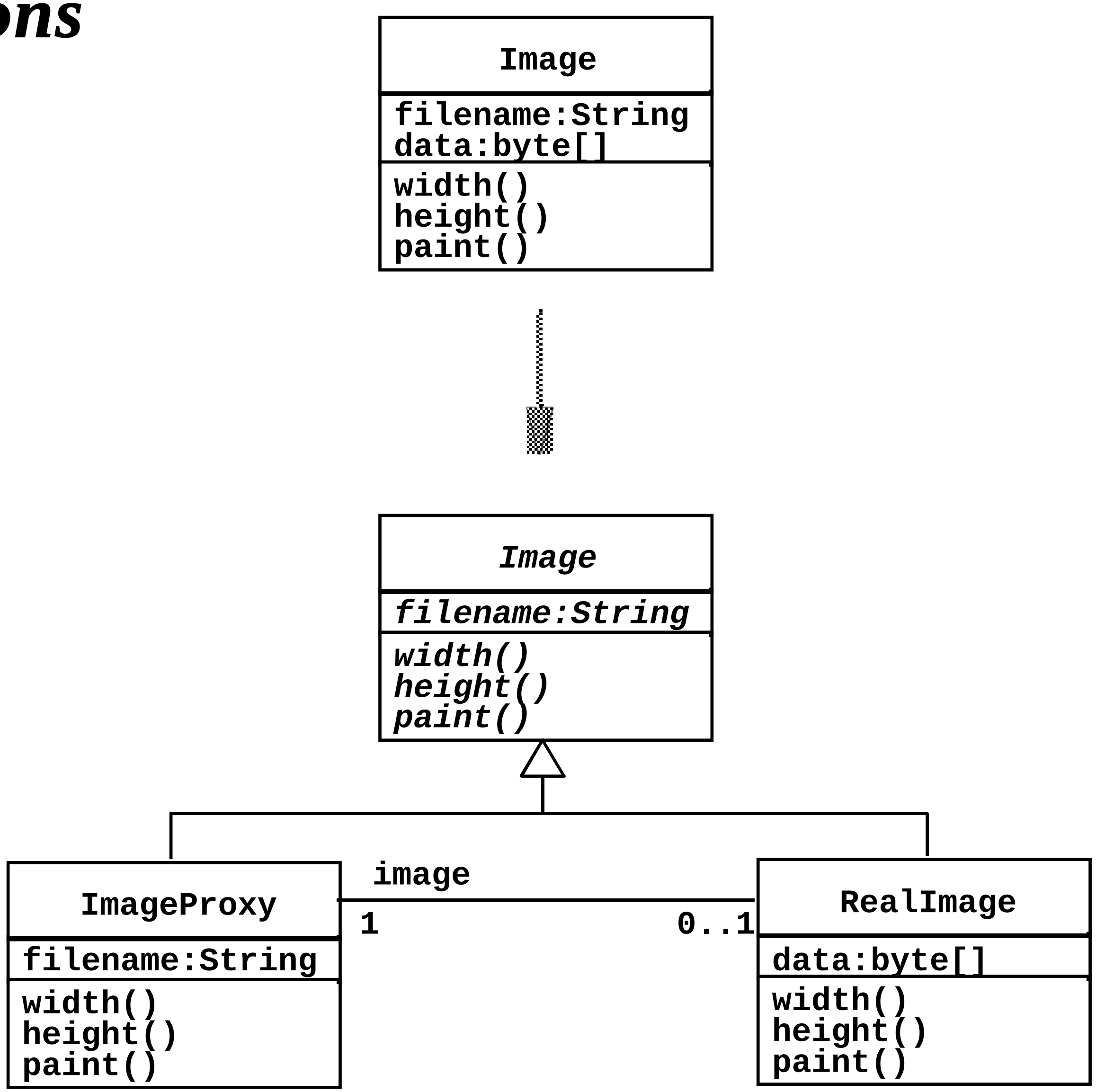
Design Optimizations ctd

Store derived attributes

- **Example: Define new classes to store information locally (database cache)**

- Problem with derived attributes:
  - **Derived attributes must be updated when base values change.**
  - **There are 3 ways to deal with the update problem:**
    - **Explicit code: Implementor determines affected derived attributes (push)**
    - **Periodic computation: Recompute derived attribute occasionally (pull)**
    - **Active value: An attribute can designate set of dependent values which are automatically updated when active value is changed (notification, data trigger)**

Optimization Activities: Delaying Complex Computations



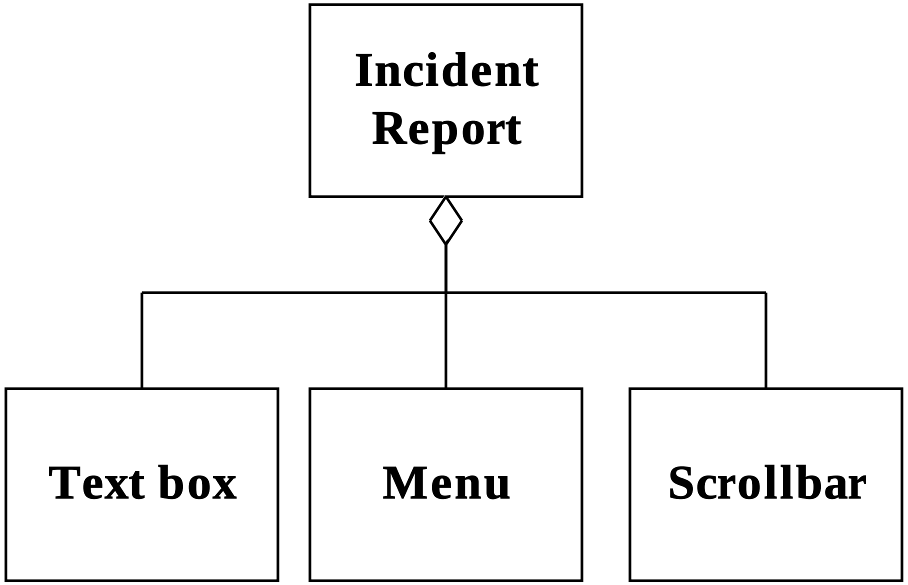
Realization of Application Domain Classes

- New objects are often needed during object design:
  - **Use of Design patterns lead to new classes**
  - **The implementation of algorithms may necessitate objects to hold values**
  - **New low-level operations may be needed during the decomposition of high-level operations**
- Example: The EraseArea() operation offered by a drawing program.
  - **Conceptually very simple**
  - **Implementation**
    - **Area represented by pixels**
    - **Repair () cleans up objects partially covered by the erased area**
    - **Redraw() draws objects uncovered by the erasure**
    - **Draw() erases pixels in background color not covered by other objects**

Application Domain vs Solution Domain Objects

Requirements Analysis
(Language of Application Domain)

Object Design
(Language of Solution Domain)



Package it all up

- Pack up design into discrete physical units that can be edited, compiled, linked, reused
- Construct physical modules
  - **Ideally use one package for each subsystem**
  - **System decomposition might not be good for implementation.**
- Two design principles for packaging
  - **Minimize coupling:**
    - **Classes in client-supplier relationships are usually loosely coupled**
    - **Large number of parameters in some methods mean strong coupling (> 4-5)**
    - **Avoid global data**
  - **Maximize cohesiveness:**
    - **Classes closely connected by associations => same package**

Packaging Heuristics

- Each subsystem service is made available by one or more boundary objects within the package
- Start with one boundary object for each subsystem service
  - **Try to limit the number of interface operations (7+-2)**
- If the subsystem service has too many operations, reconsider the number of boundary objects
- If you have too many boundary objects, reconsider the number of subsystems
- Difference between boundary objects and Java interfaces
  - ***Boundary object :*** **Used during requirements analysis, system design and object design. Basis for service**
  - ***Java interface:*** **Used during implementation in Java (A Java interface may or may not implement an boundary object)**

Further Readings

- For more on refactoring:
  - **Fowler. *Refactoring*. Addison-Wesley. 1999.**
- For more on OCL:
  - **Warmer & Klappe. *The object constraint language*. Addison-Wesley, 1998.**