

Design Patterns for the Data Structures and Algorithms Course

Bruno R. Preiss
University of Waterloo
Bruno.Preiss@UWaterloo.Ca

<http://www.pads.uwaterloo.ca/Bruno.Preiss/talks/1999/sigcse/slides.ps>

Outline of the Talk

- Introduction
- Object Hierarchies and Design Patterns
- Algorithmic Abstraction
- Summary and Conclusions

Abstract

- design patterns
 - describe/document recurring object-oriented designs
 - framework for teaching design
- Thesis:

Use object-oriented *design patterns* in the teaching of the *second course*—data structures and algorithms.

Introduction

- a good professional practitioner studies the works of others
 - discover recurring patterns
 - incorporate patterns own practice
- benefits of oo methodologies and design patterns are profound
 - patterns make designs flexible and reusable
 - don't save them for advanced courses

Pedagogic Benefits

“Computer science is [the] science of *abstraction*.”

[Aho & Ullman]

- design patterns provide framework for talking about design
- provide framework for thinking about and comparing design decisions/trade-offs
- provide a *vocabulary* for describing software designs

Object Hierarchies and Design Patterns

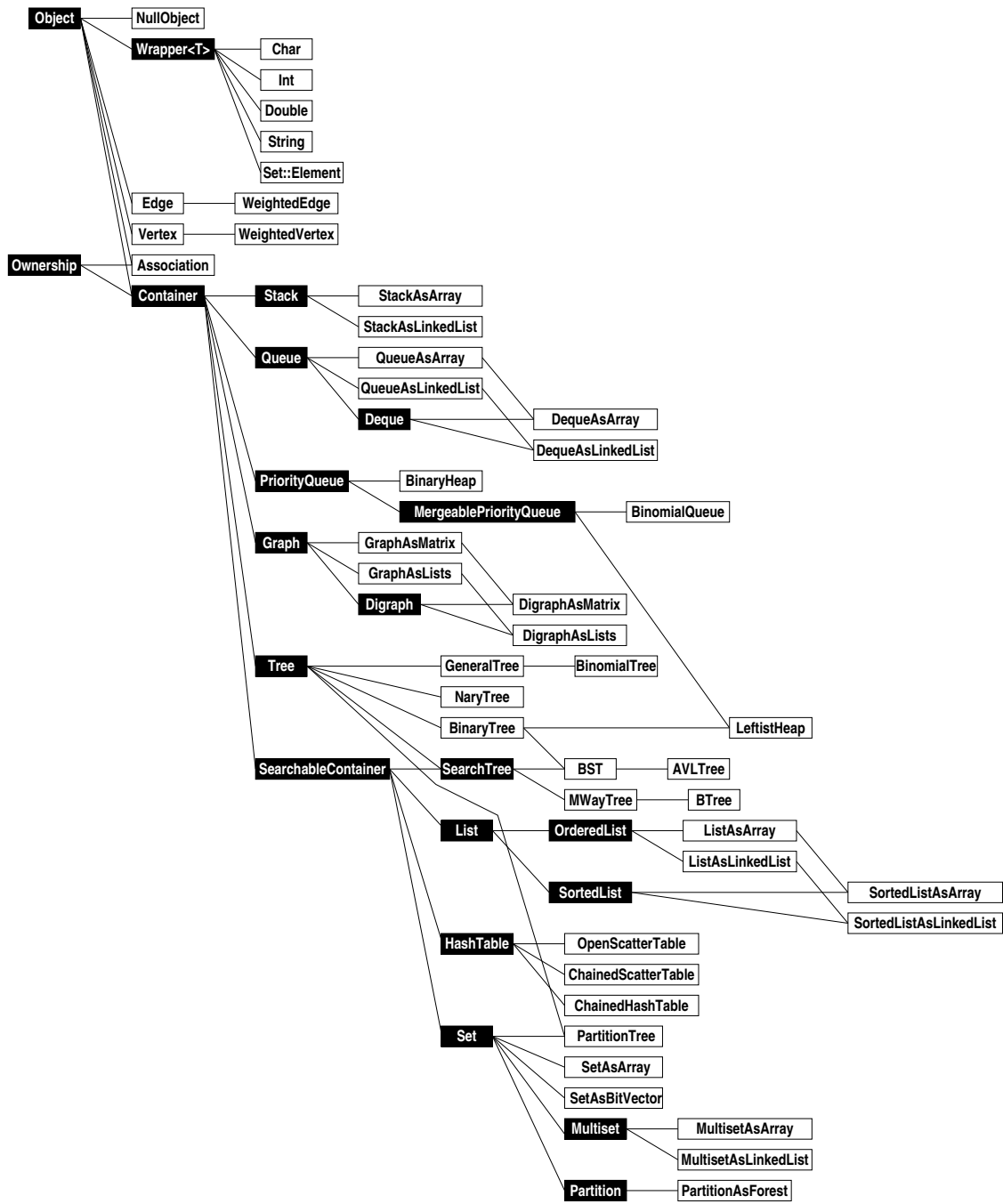
- more to oo programming than fields plus methods
 - *classification* of objects
 - *relationships* between classes
- relationships expressed via *derivation* and *inheritance*
- related classes are derived from a common abstract base class
- *patterns* capture relationships/interactions between unrelated classes

Catalog of Patterns

- containers
- iterators
- cursors
- visitors
- adapters
- singletons

Hierarchical Taxonomy of Data Structures

- illustrates that almost all data structures can be implemented using either an array or a pointer-based implementation
- stacks, queues, priority queues, trees, & graphs are *containers*
- lists, search trees, hash tables, & sets are *searchable containers*
- despite well-documented objects to multiple inheritance, it can be used to show relationships between data structures that cannot be captured otherwise



C++ Language Binding

```
class Object
{
public:
    virtual int Compare (Object const&) const = 0;
    virtual HashValue Hash () const = 0;
    virtual ostream& Put (ostream&) const = 0;
    // ...
};
```

```
class Container : public virtual Object
{
public:
    virtual bool IsEmpty () const;
    virtual bool IsFull () const;
    virtual Iterator& NewIterator () const;
    virtual void Accept (Visitor&) const = 0;
    // ...
};
```

```
class Visitor
{
public:
    virtual void PreVisit (Object&) = 0;
    virtual void Visit (Object&) = 0;
    virtual void PostVisit (Object&) = 0;
    virtual bool IsDone () const = 0;
};
```

Algorithmic Abstraction

By using well-defined, generic interfaces to underlying data structures we can write algorithms that are independent of the underlying implementation.

- pedagogic benefit: capture the *essence* of an algorithm
- rest is implementation detail

Example of Algorithmic Abstraction

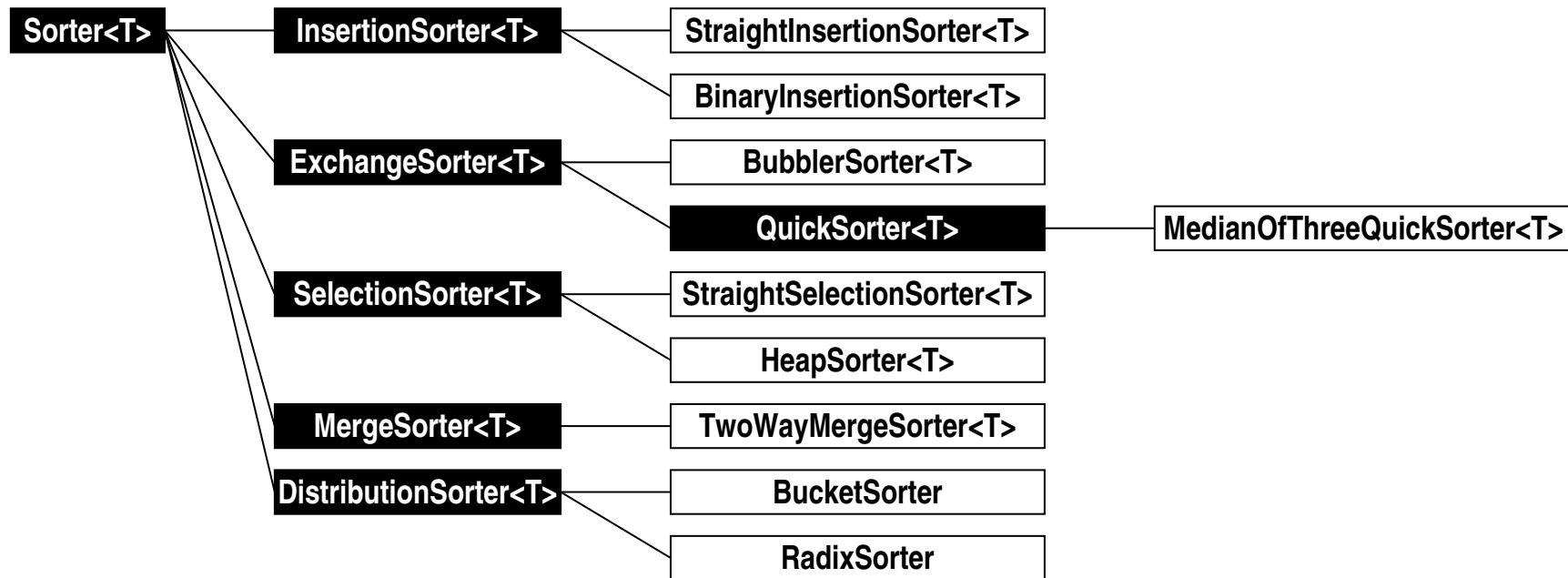
```
class HashingVisitor : public Visitor
{
    HashValue value;
public:
    HashingVisitor ()
        { value = 0; }
    void Visit (Object& object)
        { value += object.Hash (); }
    HashValue Value () const
        { return value; }
};
```

```
HashValue Container::Hash () const
{
    HashingVisitor visitor;
    Accept (visitor);
    return visitor.Value ();
}
```

Algorithmic Abstraction Scenarios

- abstract sorters and sorting functions
- abstract trees and tree traversals
- abstract graphs and graph traversals
- abstract solution spaces and problem solvers

Sorter Class Hierarchy



Abstract Trees and Tree Traversals

- a tree is a specialized container
- many tree algorithms systematically “visit” all nodes
- standard traversal patterns
 - depth-first traversal (preorder, inorder, postorder)
 - breadth-first traversal
- pedagogic benefit: learn traversal without considering either the explicit tree representation or the application-specific activities

```
class Tree : public virtual Container
{
public:
    virtual Object& Key () const = 0;
    virtual Tree& Subtree (unsigned int) const = 0;
    virtual bool IsEmpty () const = 0;
    virtual unsigned int Degree () const = 0;
    virtual void DepthFirstTraversal (Visitor&) const;
    // ...
};
```

```
void Tree::DepthFirstTraversal (Visitor& visitor) const
{
    if (visitor.IsDone ())
        return;
    if (!IsEmpty ())
    {
        visitor.PreVisit (Key ());
        for (unsigned int i = 0; i < Degree (); ++i)
            Subtree (i).DepthFirstTraversal (visitor);
        visitor.PostVisit (Key ());
    }
}
```

Abstract Graphs and Graph Traversals

- view graph as a specialized container that holds *vertices* and *edges*
- many graph algorithms systematically “visit” all nodes
- standard traversal patterns
 - depth-first traversal (preorder, postorder)
 - breadth-first traversal
 - topological-order traversal

- use *iterators* to enumerate
 - vertices in graph
 - edges in graph
 - edges emanating/incident on a vertex
 - vertices adjacent to a vertex

Abstract Solution Spaces and Problem Solvers

- backtracking algs. systematically explore a solution space
- for many problems solution space is a tree
- use abstract solutions and abstract solvers
- pedagogic benefit: learn generic problem solving strategy

```
class Solution : public Object
{
public:
    virtual bool IsFeasible () const = 0;
    virtual bool IsComplete () const = 0;
    virtual int Objective () const = 0;
    virtual int Bound () const = 0;
    virtual Iterator& Successors () const = 0;
};
```

```
class Solver
{
public:
    virtual Solution& Solve (Solution const&);
    // ...
};
```

```
void DepthFirstSolver::Solve (Solution const& solution)
{
    if (solution.IsComplete ())
        UpdateBest (solution);
    else
    {
        Iterator& i = solution.Successors ();
        while (!i.IsDone ()) {
            Solution& successor = (Solution&) (*i);
            DoSolve (successor);
            delete &successor;
            ++i;
        }
        delete &i;
    }
}
```

Summary and Conclusions

Preach design patterns from day one:

- speak the *vocabulary* of objects and design patterns
- use appropriate *abstractions* incl. algorithmic abstraction
- demonstrate common *design patterns*
- explicate *relationships* between classes
- exploit *commonalities* between classes
- illustrate salient *differences* between classes