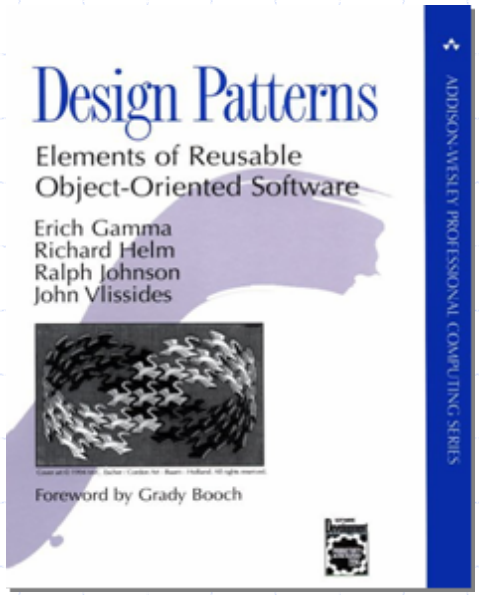
An Introduction to Design Patterns

Douglas C. Schmidt
Vanderbilt University



www.dre.vanderbilt.edu/~schmidt/gof.ppt
schmidt@dre.vanderbilt.edu
Based on material produced by John Vlissides

Overview

Part I: Motivation & Concept

- the issue
- what design patterns are
- what they're good for
- how we develop & categorize them

Overview (cont'd)

Part II: Application

- use patterns to design a document editor
- demonstrate usage & benefits

Part III: Wrap-Up

- observations, caveats, & conclusion

Part I: Motivation & Concept

OOD methods emphasize design notations

- Fine for specification, documentation

But OOD is more than just drawing diagrams

- Good draftsmen  good designers

Good OO designers rely on lots of experience

- At least as important as syntax

Most powerful reuse is *design* reuse

- Match problem to design experience

Recurring Design Structures

OO systems exhibit recurring structures that promote

- abstraction
- flexibility
- modularity
- elegance

Therein lies valuable design knowledge

Problem:

capturing, communicating, & applying this knowledge

A *Design Pattern*...

- abstracts a recurring design structure
- comprises class and/or object
  - dependencies
  - structures
  - interactions
  - conventions
- names & specifies the design structure explicitly
- distills design experience

Four Basic Parts

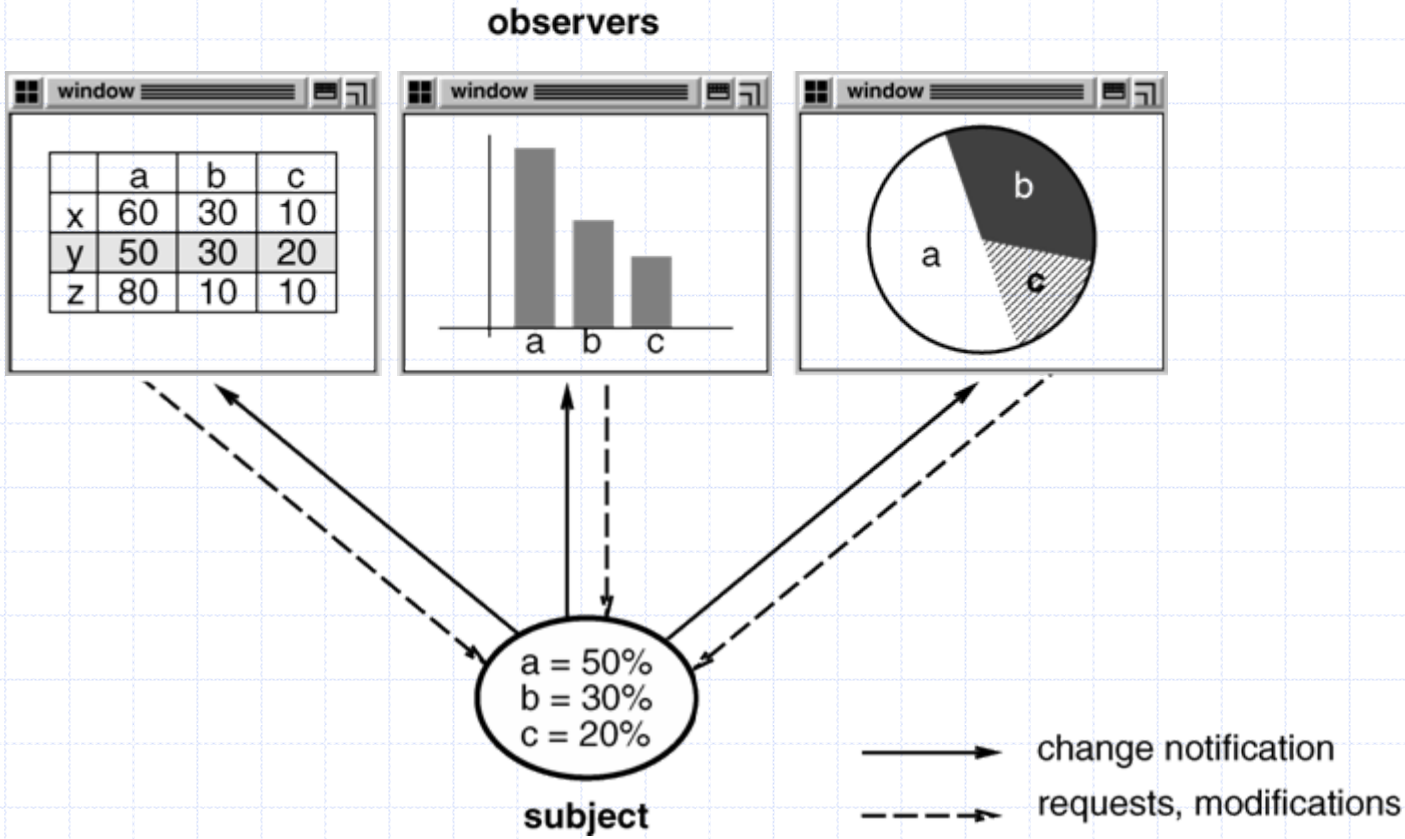
1. Name
2. Problem (including "forces")
3. Solution
4. Consequences & trade-offs of application

Language- & implementation-independent

A "micro-architecture"

Adjunct to existing methodologies (RUP, Fusion, SCRUM, etc.)

Example: OBSERVER



Goals

Codify good design

- distill & generalize experience
- aid to novices & experts alike

Give design structures explicit names

- common vocabulary
- reduced complexity
- greater expressiveness

Capture & preserve design information

- articulate design decisions succinctly
- improve documentation

Facilitate restructuring/refactoring

- patterns are interrelated
- additional flexibility

Design Space for GoF Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Scope: domain over which a pattern applies

Purpose: reflects what a pattern does

Part I: Motivation & Concept (cont'd)

Design Pattern Template (1st half)

NAME scope purpose

Intent

short description of the pattern & its purpose

Also Known As

Any aliases this pattern is known by

Motivation

motivating scenario demonstrating pattern's use

Applicability

circumstances in which pattern applies

Structure

graphical representation of the pattern using modified UML notation

Participants

participating classes and/or objects & their responsibilities

Design Pattern Template (2nd half)

...

Collaborations
- how participants cooperate to carry out their responsibilities

Consequences
- the results of application, benefits, liabilities

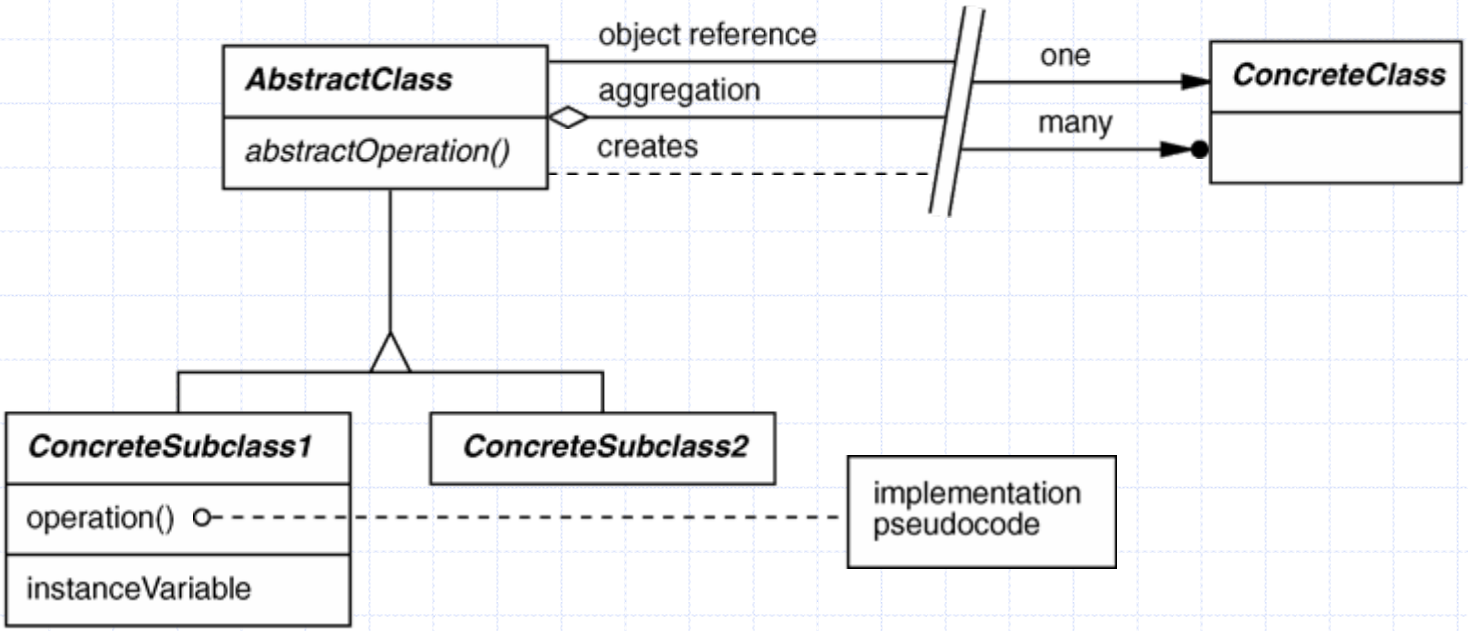
Implementation
- pitfalls, hints, techniques, plus language-dependent issues

Sample Code
- sample implementations in C++, Java, C#, Smalltalk, C, etc.

Known Uses
- examples drawn from existing systems

Related Patterns
- discussion of other patterns that relate to this one

Modified UML/OMT Notation



OBSERVER — object behavioral

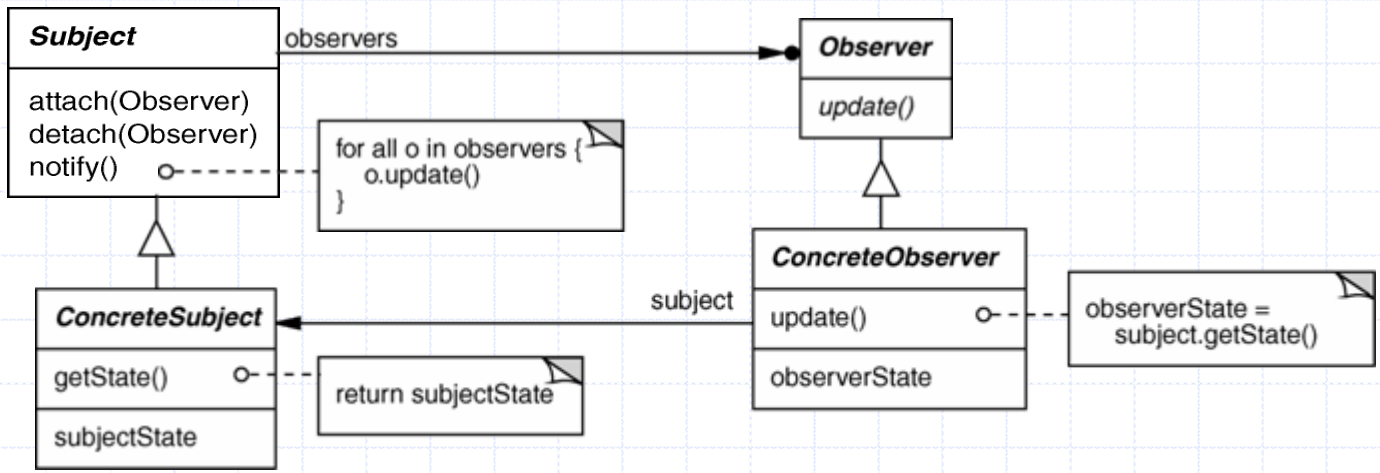
Intent

define a one-to-many dependency between objects so that when one object changes state, all dependents are notified & updated

Applicability

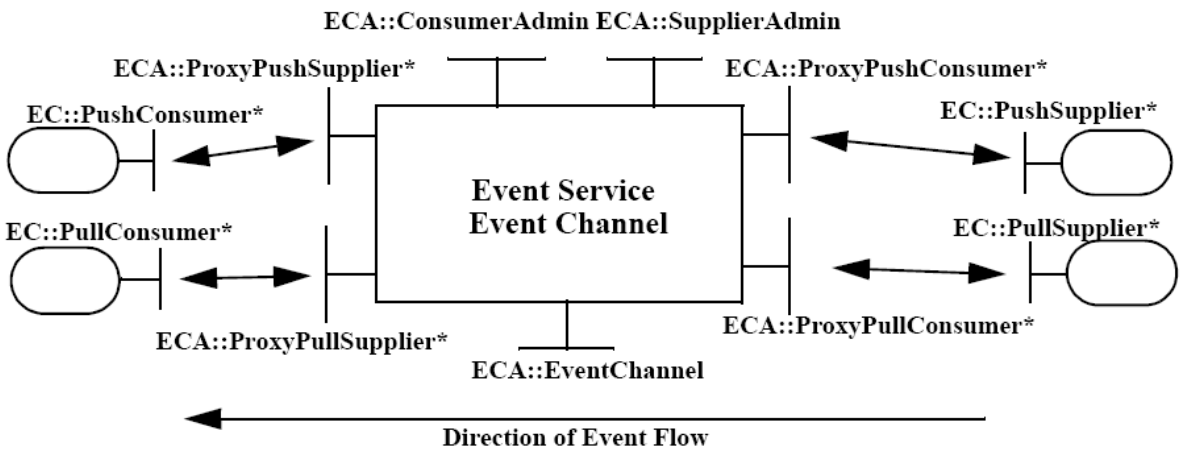
- an abstraction has two aspects, one dependent on the other
- a change to one object requires changing untold others
- an object should notify unknown other objects

Structure



OBSERVER object behavioral

```
class ProxyPushConsumer : public // …
  virtual void push (const CORBA::Any &event) {
    for (std::vector<PushConsumer>::iterator i
          (consumers.begin ()); i != consumers.end (); i++)
      (*i).push (event);
  }

class MyPushConsumer : public // ….
  virtual void push
   (const CORBA::Any &event) { /* consume the event. */ }
```



CORBA Notification Service example using C++ Standard Template Library (STL) iterators (which is an example of the Iterator pattern from GoF)

OBSERVER (cont'd) object behavioral

Consequences

+ modularity: subject & observers may vary independently
+ extensibility: can define & add any number of observers
+ customizability: different observers offer different views of subject
– unexpected updates: observers don't know about each other
– update overhead: might need hints or filtering

Implementation

- subject-observer mapping
- dangling references
- update protocols: the push & pull models
- registering modifications of interest explicitly

Known Uses

- Smalltalk Model-View-Controller (MVC)
- InterViews (Subjects & Views, Observer/Observable)
- Andrew (Data Objects & Views)
- Pub/sub middleware (e.g., CORBA Notification Service, Java Message Service)
- Mailing lists

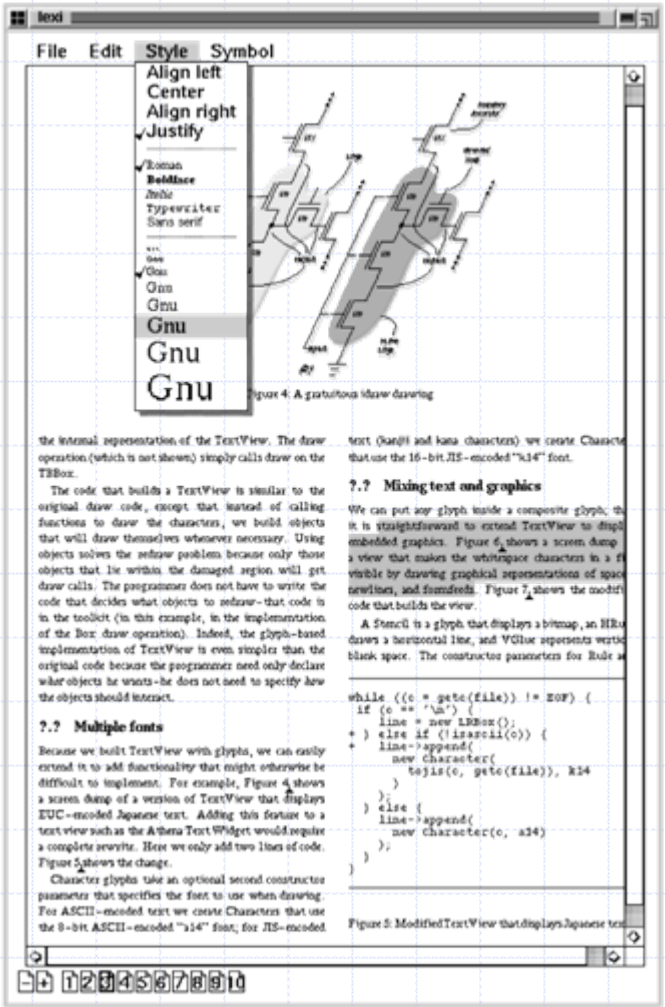
Benefits of Patterns

- *Design* reuse
- Uniform design vocabulary
- Enhance understanding, restructuring, & team communication
- Basis for automation
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

Liabilities of Patterns

- Require significant tedious & error-prone human effort to handcraft pattern implementations *design* reuse
- Can be deceptively simple uniform design vocabulary
- May limit design options
- Leaves some important details unresolved

Part II: Application: Document Editor (Lexi)



7 Design Problems

1. Document structure
2. Formatting
3. Embellishment
4. Multiple look & feels
5. Multiple window systems
6. User operations
7. Spelling checking & hyphenation

Note that none of these patterns are restricted to document editors...

Document Structure

Goals:

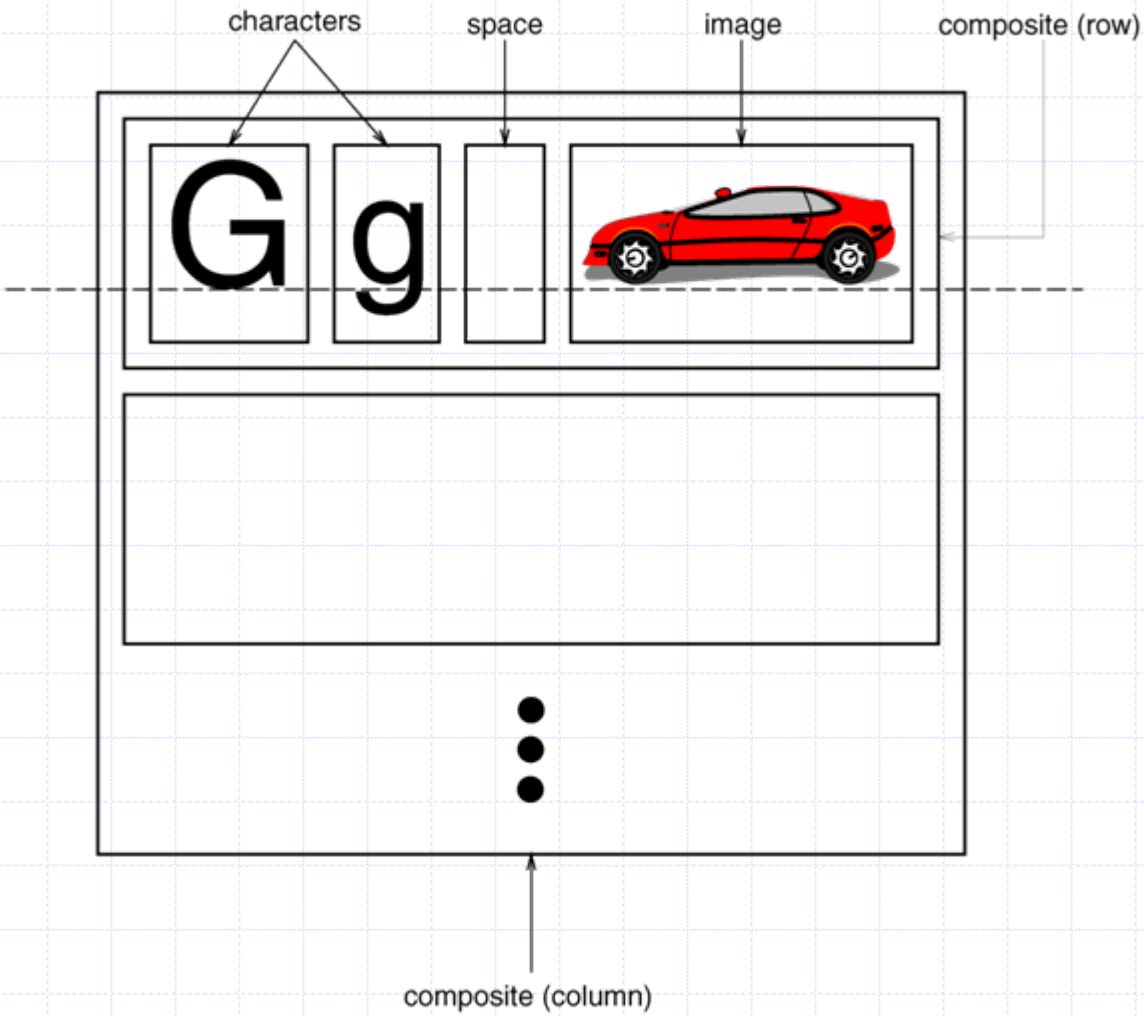
- present document's visual aspects
- drawing, hit detection, alignment
- support physical structure (e.g., lines, columns)

Constraints/forces:

- treat text & graphics uniformly
- no distinction between one & many

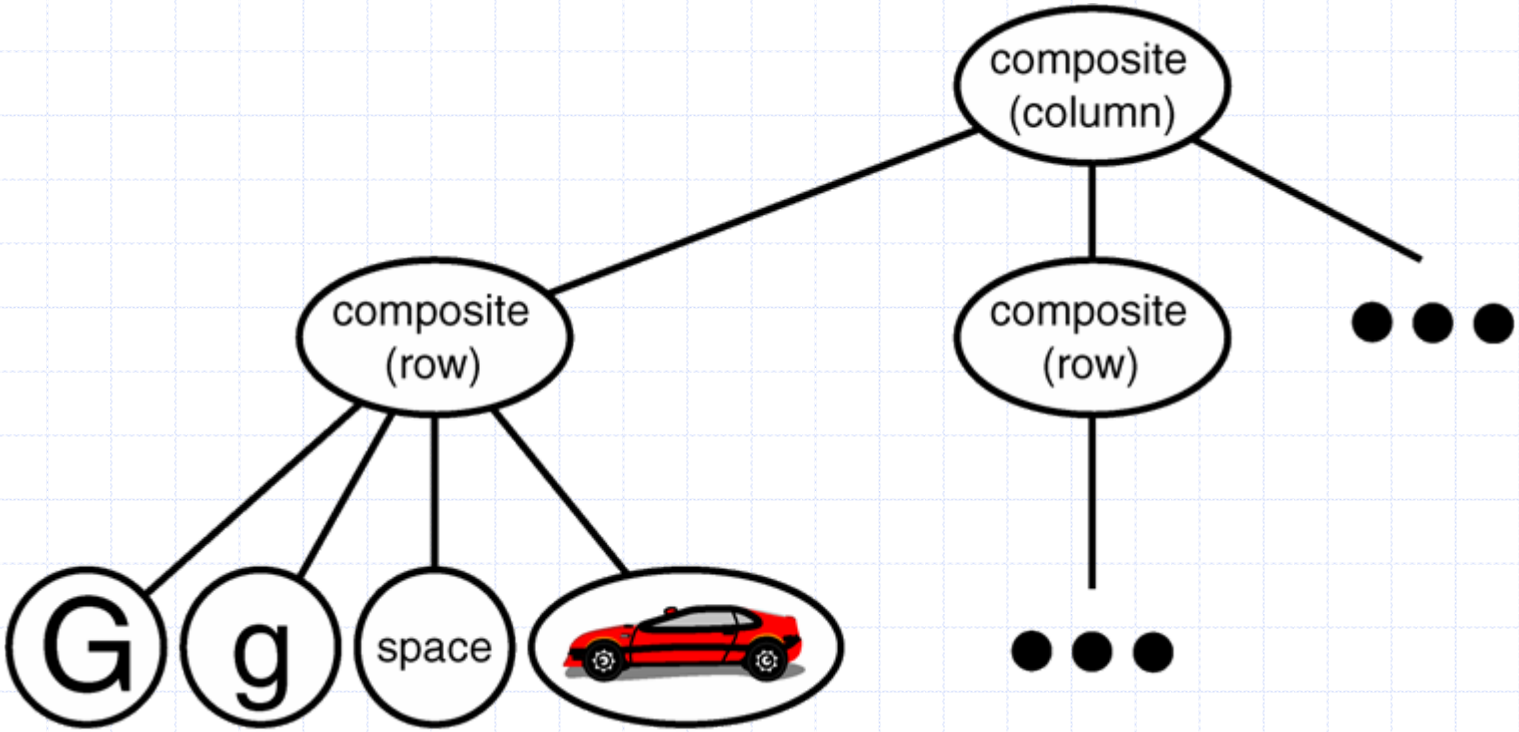
Document Structure (cont'd)

Solution: Recursive Composition



Document Structure (cont'd)

Object Structure



Document Structure (cont'd)

Glyph

Base class for composable graphical objects

Basic interface:

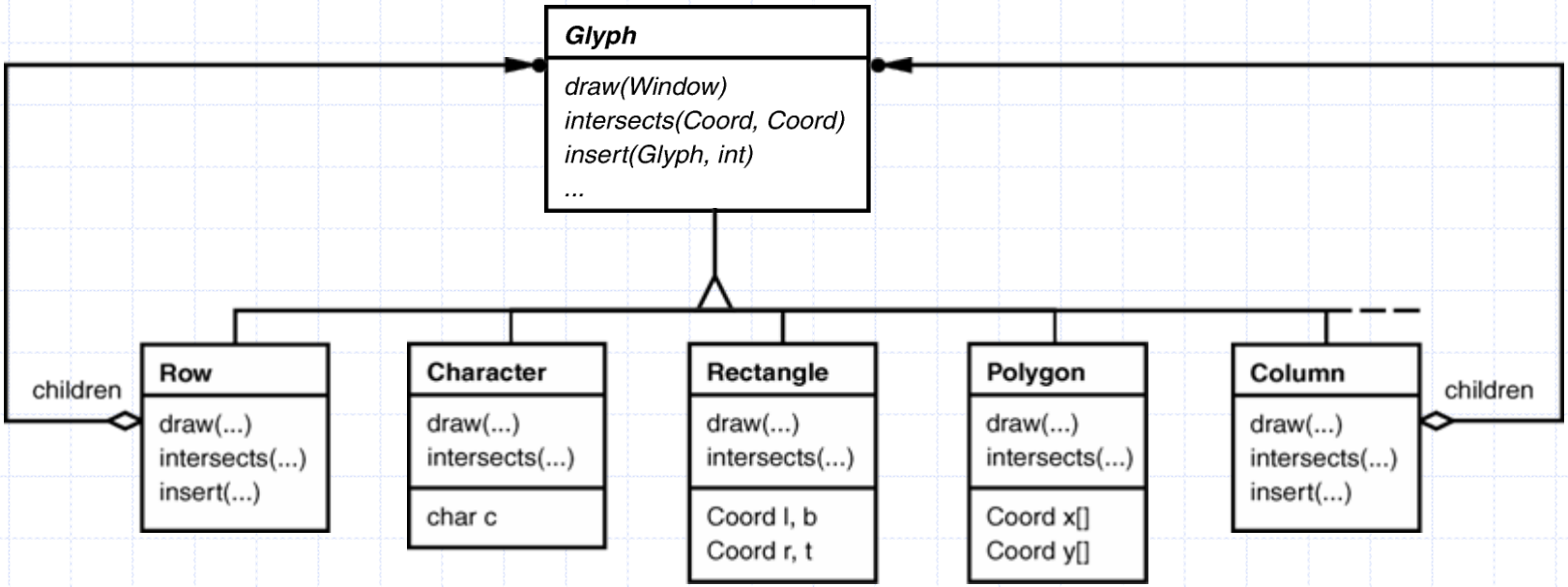
Task	Operations
appearance	`void draw(Window)`
hit detection	`boolean intersects(Coord, Coord)`

structure	`void insert(Glyph)`
	`void remove(Glyph)`
	`Glyph child(int n)`
	`Glyph parent()`

Subclasses: Character, Image, Space, Row, Column

Document Structure (cont'd)

Glyph Hierarchy



Note the inherent recursion in this hierarchy

- i.e., a Row *is a* Glyph & a Row also *has* Glyphs!

COMPOSITE — object structural

Intent

treat individual objects & multiple, recursively-composed objects uniformly

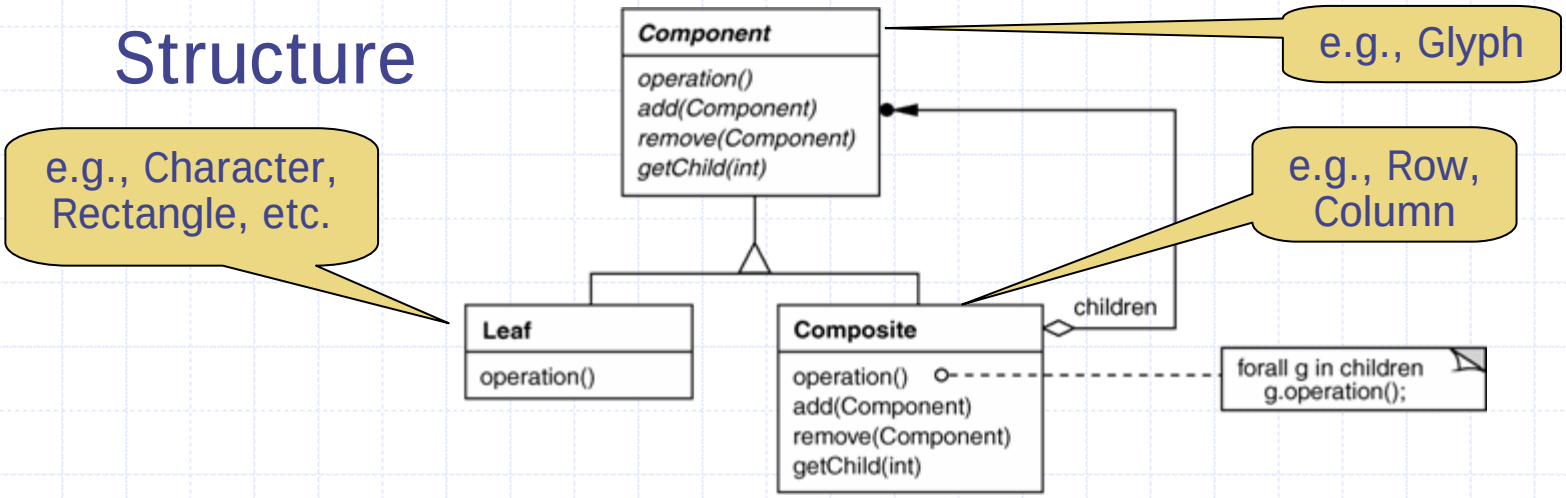
Applicability

objects must be composed recursively,

and no distinction between individual & composed elements,

and objects in structure can be treated uniformly

Structure



COMPOSITE object structural

Component

```
class Glyph {
public:
  virtual void
    draw (const Drawing_Region &) = 0;
  // ...
  virtual void add_child (Glyph *) {}
protected:
  int x_, y_;
  // Coordinate position.
};
```

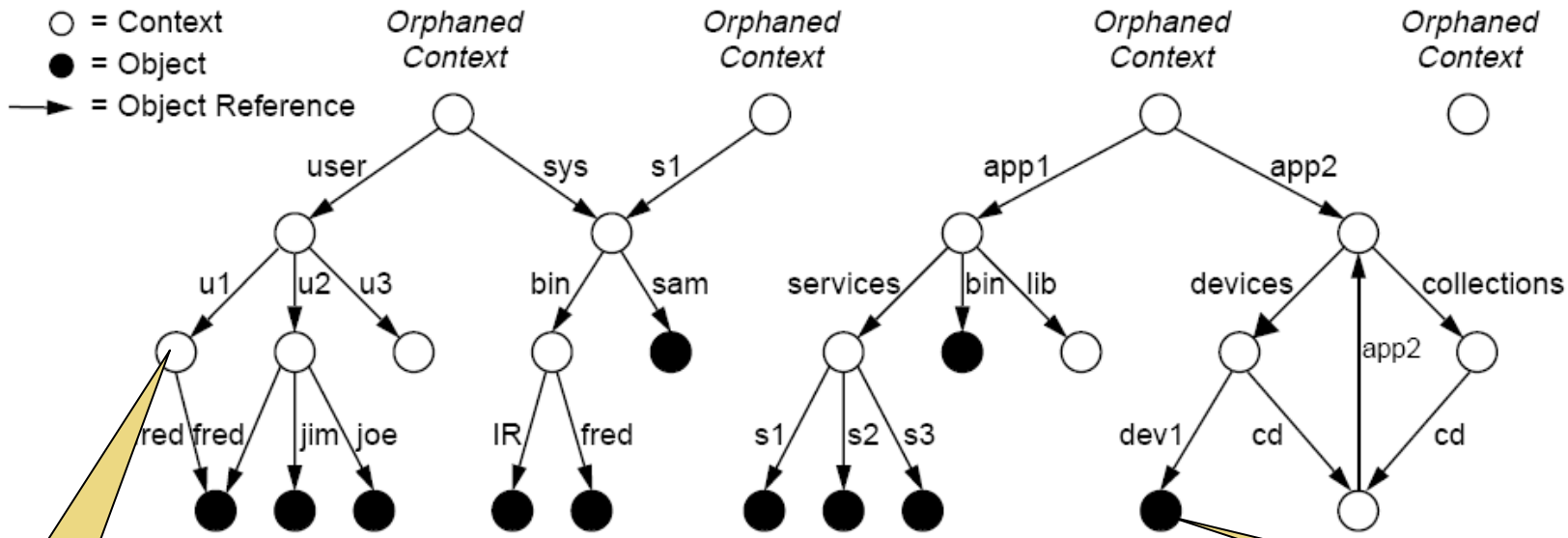
Leaf

```
class Character : public Glyph {
public:
  Character
    (const std::string &name);
  // ...
  virtual void
    draw (const Drawing_Region &c)
  { c.draw_text (x_, y_, name_); }
private:
  std::string name_;
};
```

Composite

```
class Row : public Glyph {
public:
  Row (std::vector<Glyph*> children);
  // ...
  virtual void
    draw (const Drawing_Region &c){
      for (std::vector<Glyph*>::iterator
             i (children_);
           i != children_.end ();
           i++)
        (*i)->draw (c);
  }
  // ...
  virtual void add_child (Glyph *g) {
    children_.push_back (g);
  }
private:
  std::vector<Glyph*> children_;
  // ...
};
```

COMPOSITE object structural



CORBA Naming Service example using CosNaming::BindingIterator (which is an example of the "Batch Iterator" pattern compound from POSA5)

COMPOSITE object structural

```
void list_context (CosNaming::NamingContext_ptr nc) {
  CosNaming::BindingIterator_var it; // Iterator reference
  CosNaming::BindingList_var bl;     // Binding list
  const CORBA::ULong CHUNK = 100;    // Chunk size

  nc->list (CHUNK, bl, it);          // Get first chunk
  show_chunk (bl, nc);               // Print first chunk
  if (!CORBA::is_nil(it)) {          // More bindings?
    while (it->next_n(CHUNK, bl))    // Get next chunk
      show_chunk (bl, nc);           // Print chunk
    it->destroy();                   // Clean up
  }
}

void show_chunk (const CosNaming::BindingList_ptr &bl, // Helper function
                 CosNaming::NamingContext_ptr nc) {
  for (CORBA::ULong i = 0; i < bl.length (); ++i) {
    cout << bl[i].binding_name[0].id << "." << bl[i].binding_name[0].kind;

    if (bl[i].binding_type == CosNaming::ncontext) {
      cout << ": context" << endl;
      CORBA::Object_var obj = nc->resolve (bl[i].binding_name);
      list_context (CosNaming::NamingContext::_narrow (obj));
    }
    else cout << ": reference" << endl;
  }
}
```

Handle Composite Node

Handle Leaf Node

COMPOSITE (cont'd) object structural

Consequences

+ uniformity: treat components the same regardless of complexity
+ extensibility: new Component subclasses work wherever old ones do
– overhead: might need prohibitive numbers of objects
– Awkward designs: may need to treat leaves as lobotomized composites

Implementation

- do Components know their parents?
- uniform interface for both leaves & composites?
- don't allocate storage for children in Component base class
- responsibility for deleting children

Known Uses

- ET++ Vobjects
- InterViews Glyphs, Styles
- Unidraw Components, MacroCommands
- Directory structures on UNIX & Windows
- Naming Contexts in CORBA
- MIME types in SOAP

Formatting

Goals:

- automatic linebreaking, justification

Constraints/forces:

- support multiple linebreaking algorithms
- don’t tightly couple these algorithms with the document structure

Solution: Encapsulate Linebreaking Strategy

Compositor

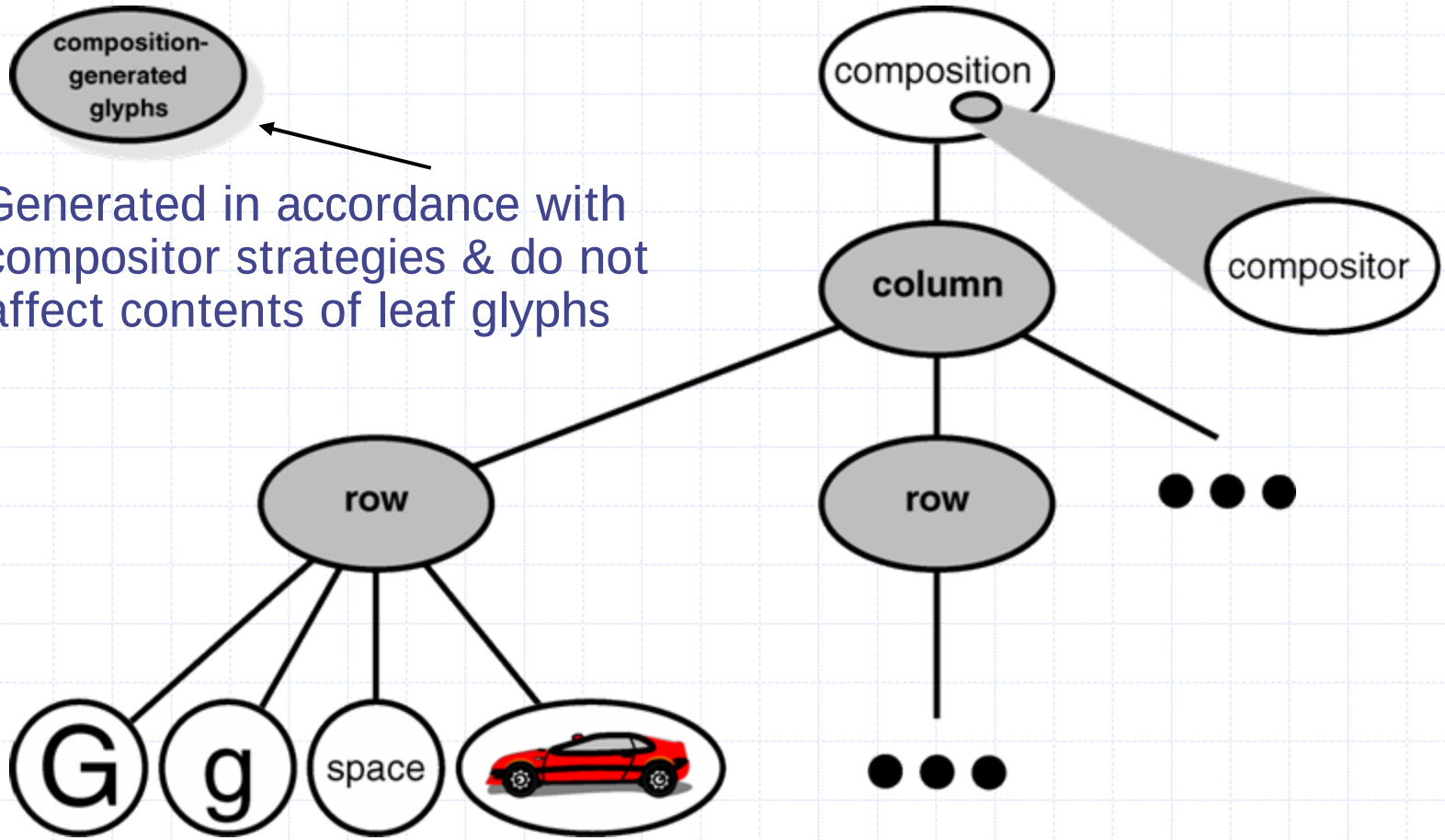
- base class abstracts linebreaking algorithm
- subclasses for specialized algorithms, e.g., **SimpleCompositor**, **TeXCompositor**

Composition

- composite glyph (typically representing a column)
- supplied a compositor & leaf glyphs
- creates row-column structure as directed by compositor

Formatting (cont'd)

New Object Structure



Generated in accordance with compositor strategies & do not affect contents of leaf glyphs

STRATEGY — object behavioral

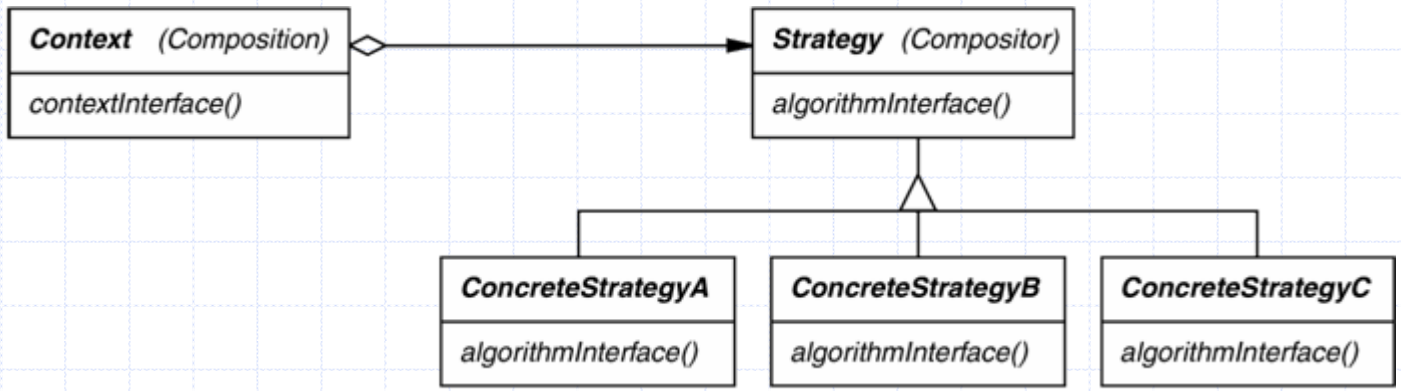
Intent

define a family of algorithms, encapsulate each one, & make them interchangeable to let clients & algorithms vary independently

Applicability

when an object should be configurable with one of many algorithms,
and all algorithms can be encapsulated,
and one interface covers all encapsulations

Structure



STRATEGY — object behavioral

```
class Composition : public Glyph {
public:
  void perform_composition
       (const Compositor &compositor,
        const std::vector<Glyphs*> &leaf_glyphs)
  {
    compositor.set_context (*this);
    for (std::vector<Glyph*>::iterator
           i (leaf_glyphs);
         i != leaf_glyphs.end ();
         i++) {
        this->insert (*i);
        compositor.compose ();
    }
  }
private:
  // Data structures for composition.
};
```

Creates row-column structure as directed by compositor

```
class Compositor {
public:
  void set_context
    (Composition &context);
  virtual void compose () = 0;
  // ...
};
```

Simple algorithm

```
class SimpleCompositor
  : public Compositor {
public:
  virtual void compose ()
  { /* ... */}
};
```

Complex algorithm

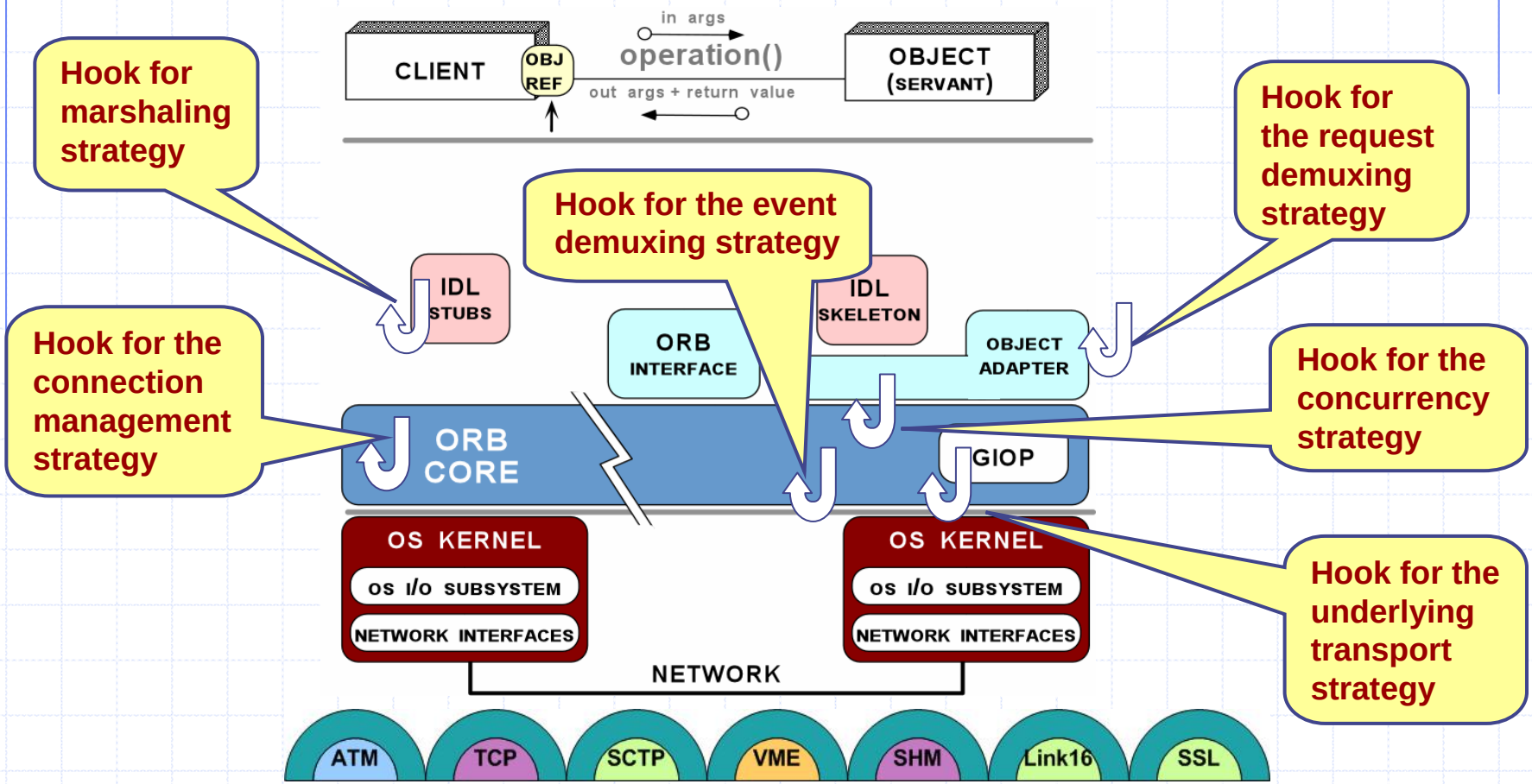
```
class TexCompositor
  : public Compositor {
public:
  virtual void compose ()
  { /* ... */}
};
```

Strategy can be changed dynamically!

```
Composition comp;
TexCompositor tc;
comp.perform_composition (tc, leaf_glyphs);

SimpleCompositor sc;
comp.perform_composition (sc, leaf_glyphs);
```

Formatting (cont'd)

STRATEGY object behavioral



Strategy can also be applied in distributed systems (e.g., middleware)

STRATEGY (cont'd) object behavioral

Consequences

+ greater flexibility, reuse
+ can change algorithms dynamically
– strategy creation & communication overhead
– inflexible Strategy interface
– semantic incompatibility of multiple strategies used together

Implementation

- exchanging information between a Strategy & its context
- static strategy selection via parameterized types

Known Uses

- InterViews text formatting
- RTL register allocation & scheduling strategies
- ET++SwapsManager calculation engines
- The ACE ORB (TAO) Real-time CORBA middleware

See Also

- Bridge pattern (object structural)

Template Method (cont'd) — class behavioral

Intent

- Provide a skeleton of an algorithm in a method, deferring some steps to subclasses

```
class Composition : public Glyph {
public:
  // Template Method.
  void perform_composition (const std::vector<Glyphs*> &leaf_glyphs) {
    set_context (*this);
    for (std::vector<Glyph*>::iterator i (leaf_glyphs);
         i != leaf_glyphs.end (); i++) {
        insert (*i);
        compose ();
     }
  }
  virtual void compose () = 0; // Hook Method
protected:
  // Data structures for composition.
};

class Simple_Composition : public Composition {
  virtual void compose () { /* ... */ }
};

class Tex_Composition : public Composition {
  virtual void compose () { /* ... */ }
};
```

Template Method (cont'd) class behavioral

Intent

- Provide a skeleton of an algorithm in a method, deferring some steps to subclasses

```
class Base_Class {
public:
  // Template Method.
  void template_method (void) {
     hook_method_1 ();
     hook_method_2 ();
     // ...
  }
protected:
  virtual void hook_method_1 () = 0;
  virtual void hook_method_2 () = 0;
};

class Derived_Class_1 : public Base_Class {
  virtual void hook_method_2 () { /* ... */ }
};

class Derived_Class_2 : public Base_Class {
  virtual void hook_method_1 () { /* ... */ }
  virtual void hook_method_2 () { /* ... */ }
};
```

Embellishment

Goals:

- add a frame around text composition
- add scrolling capability

Constraints/forces:

- embellishments should be reusable without subclassing, i.e., so they can be added dynamically at runtime
- should go unnoticed by clients

Embellishment (cont'd)

Solution: "Transparent" Enclosure

Monoglyph

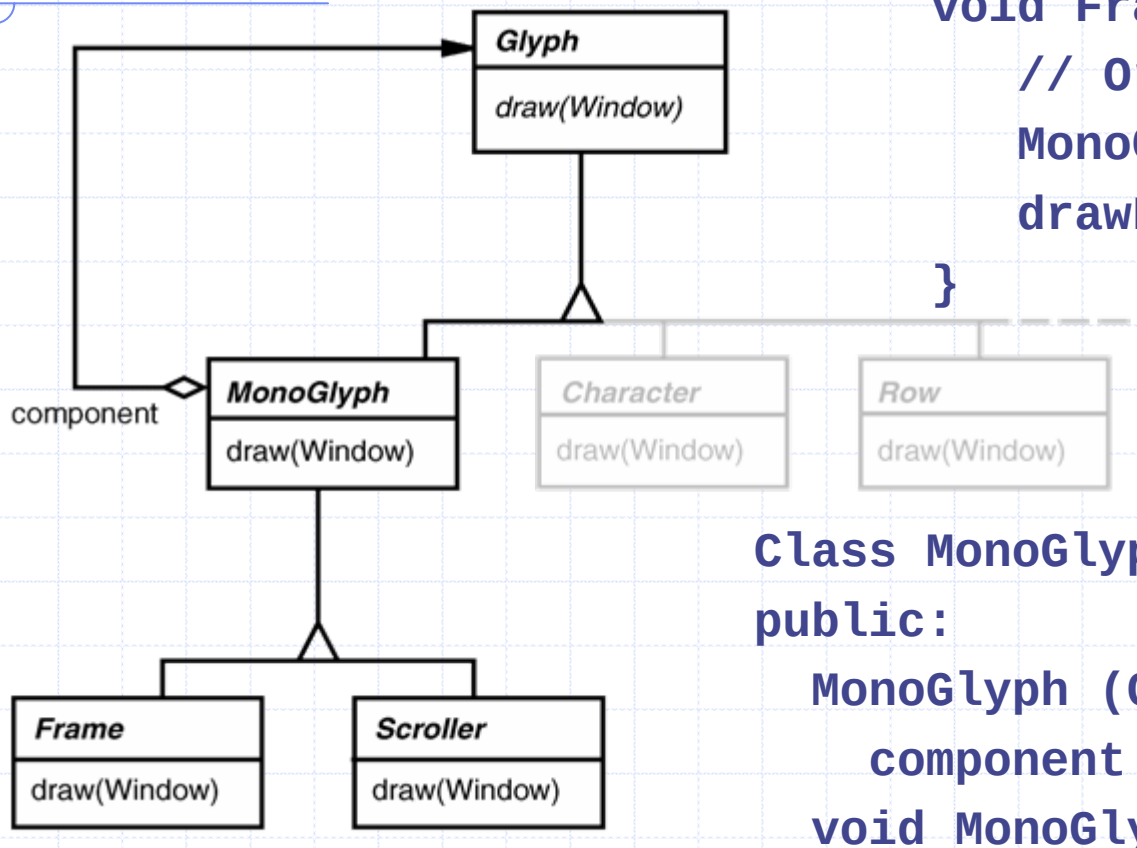
- base class for glyphs having *one* child
- operations on MonoGlyph (ultimately) pass through to child

MonoGlyph subclasses:

- **Frame**: adds a border of specified width
- **Scroller**: scrolls/clips child, adds scrollbars

Embellishment (cont'd)

MonoGlyph Hierarchy

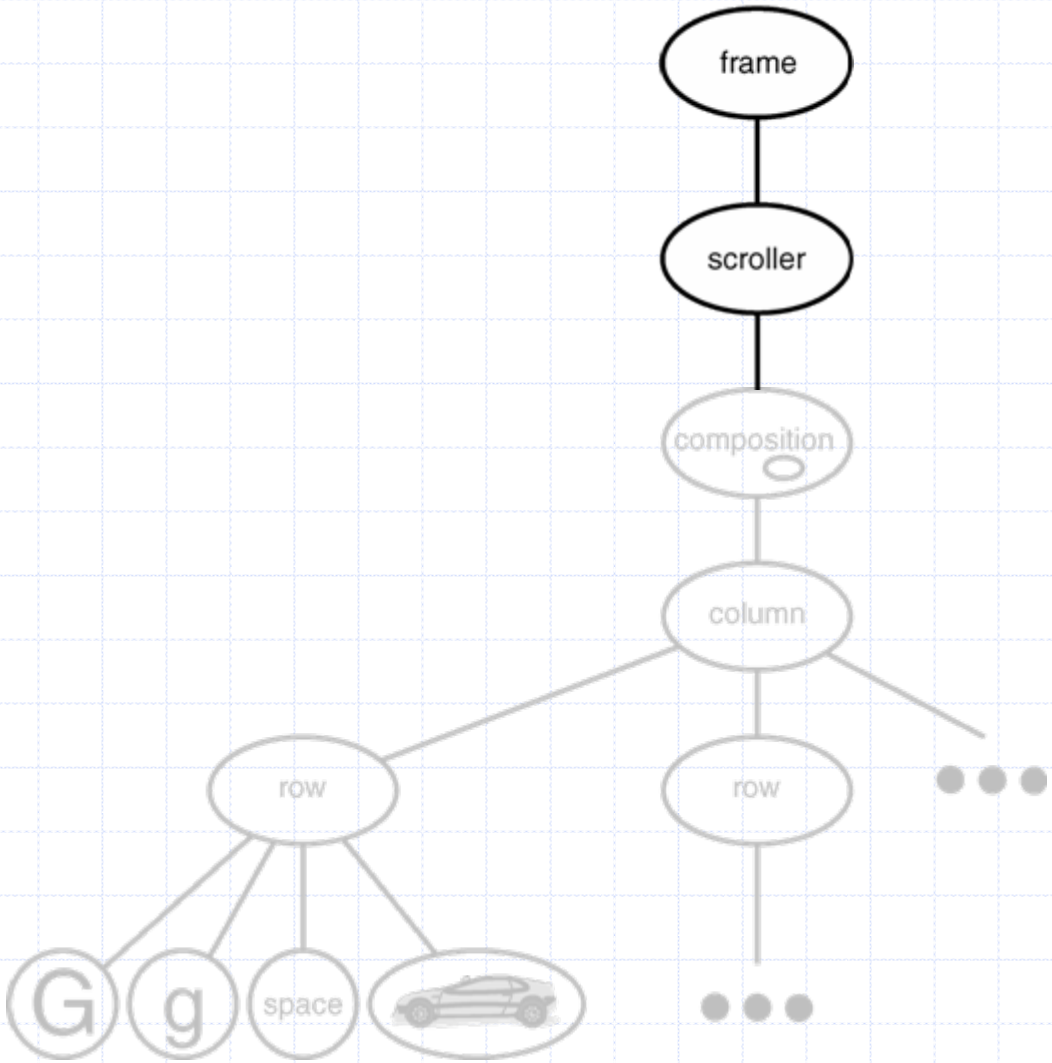



```
void Frame::draw (Window &w) {
   // Order may be important!
   MonoGlyph::draw (w);
   drawFrame (w);
}
```

```
Class MonoGlyph : public Glyph {
public:
  MonoGlyph (Glyph *g):
    component (g) {}
  void MonoGlyph::draw (Window &w)
   { component->draw (w); }
private:
  Glyph *component;
};
```

Embellishment (cont'd)

New Object Structure



Embellishment (cont'd)

DECORATOR — object structural

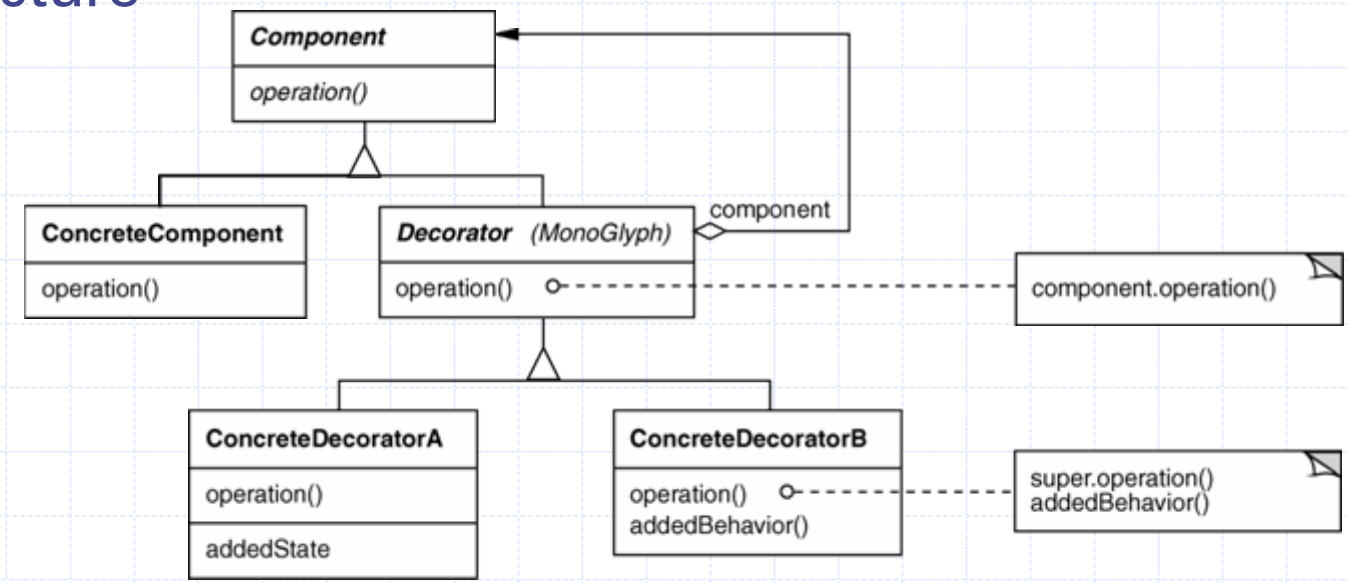
Intent

Transparently augment objects with new responsibilities dynamically

Applicability

- when extension by subclassing is impractical
- for responsibilities that can be added & withdrawn dynamically

Structure



Embellishment (cont'd)

DECORATOR — object structural

```
size_t request_count;

void *worker_task (void *) {
  request_count++;

  // ... process the request
}
```

```
ACE_Thread_Mutex m;
size_t request_count;

void *worker_task (void *) {
  m.acquire ();
  request_count++;
  m.release ();

  // ... process the request
}
```

```
ACE_Thread_Mutex m;
size_t request_count;

void *worker_task (void *) {
  {
    ACE_Guard <ACE_Thread_Mutex> g (m);
    request_count++;
  }

  // ... process the request
}
```

Embellishment (cont'd)

DECORATOR — object structural

```
Atomic_Op<size_t, ACE_Thread_Mutex> request_count;

void *worker_task (void *) {
  request_count++;

  // ... process the request
}
```

```
template <typename T, typename LOCK>
class Atomic_Op {
public:
  void operator++ () {
    ACE_Guard <LOCK> g (m_);
    count_++;
  // ...
  }
private:
  T count_;
  LOCK m_;
};
```

DECORATOR (cont'd) object structural

Consequences

+ responsibilities can be added/removed at run-time
+ avoids subclass explosion
+ recursive nesting allows multiple responsibilities
– interface occlusion
– identity crisis
– composition of decorators is hard if there are side-effects

Implementation

- interface conformance
- use a lightweight, abstract base class for Decorator
- heavyweight base classes make Strategy more attractive

Known Uses

- embellishment objects from most OO-GUI toolkits
- ParcPlace PassivityWrapper
- InterViews DebuggingGlyph
- Java I/O classes
- ACE_Atomic_Op

Multiple Look & Feels

Goals:

- support multiple look & feel standards
- generic, Motif, Swing, PM, Macintosh, Windows, ...
- extensible for future standards

Constraints/forces:

- don’t recode existing widgets or clients
- switch look & feel without recompiling

Multiple Look & Feels (cont'd)

Solution: Abstract Object Creation

Instead of

```
MotifScrollbar *sb = new MotifScrollbar();
```

use

```
Scrollbar *sb = factory->createScrollbar();
```

where **`factory`** is an instance of **MotifFactory** or anything else that makes sense wrt our look & feel requirements

- BTW, this begs the question of who created the **`factory`**!

Multiple Look & Feels (cont'd)

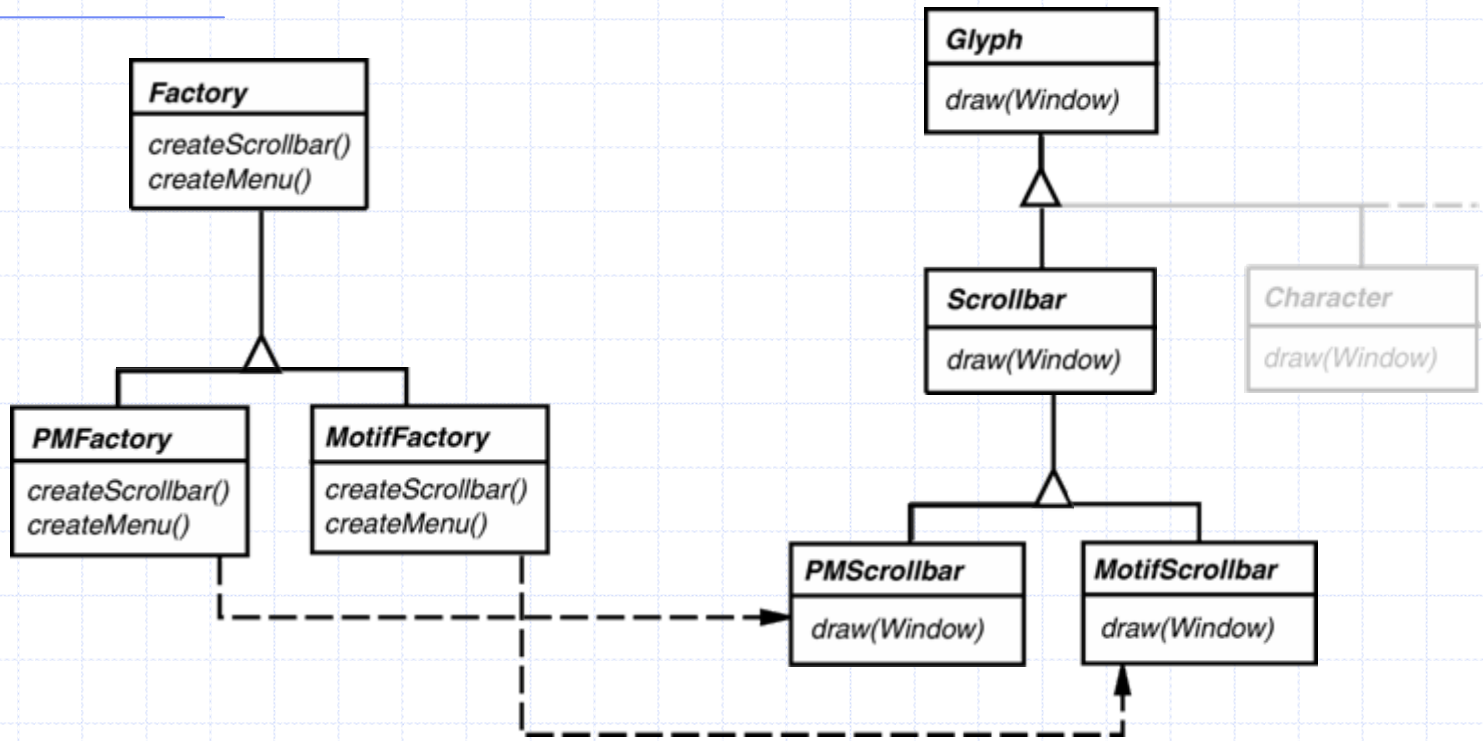
`Factory` Interface

- defines "manufacturing interface"
- subclasses produce specific products
- subclass instance chosen at run-time

```
// This class is essentially a Java interface
class GUIFactory {
public:
    virtual Scrollbar *createScrollbar() = 0;
    virtual Menu *createMenu() = 0;
    ...
};
```

Multiple Look & Feels (cont'd)

Factory Structure




```
Scrollbar *MotifFactory::createScrollBar () {
    return new MotifScrollbar();
}
Scrollbar *PMFactory::createScrollBar () {
    return new PMScrollbar();
}
```

Multiple Look & Feels (cont'd)

ABSTRACT FACTORY — object creational

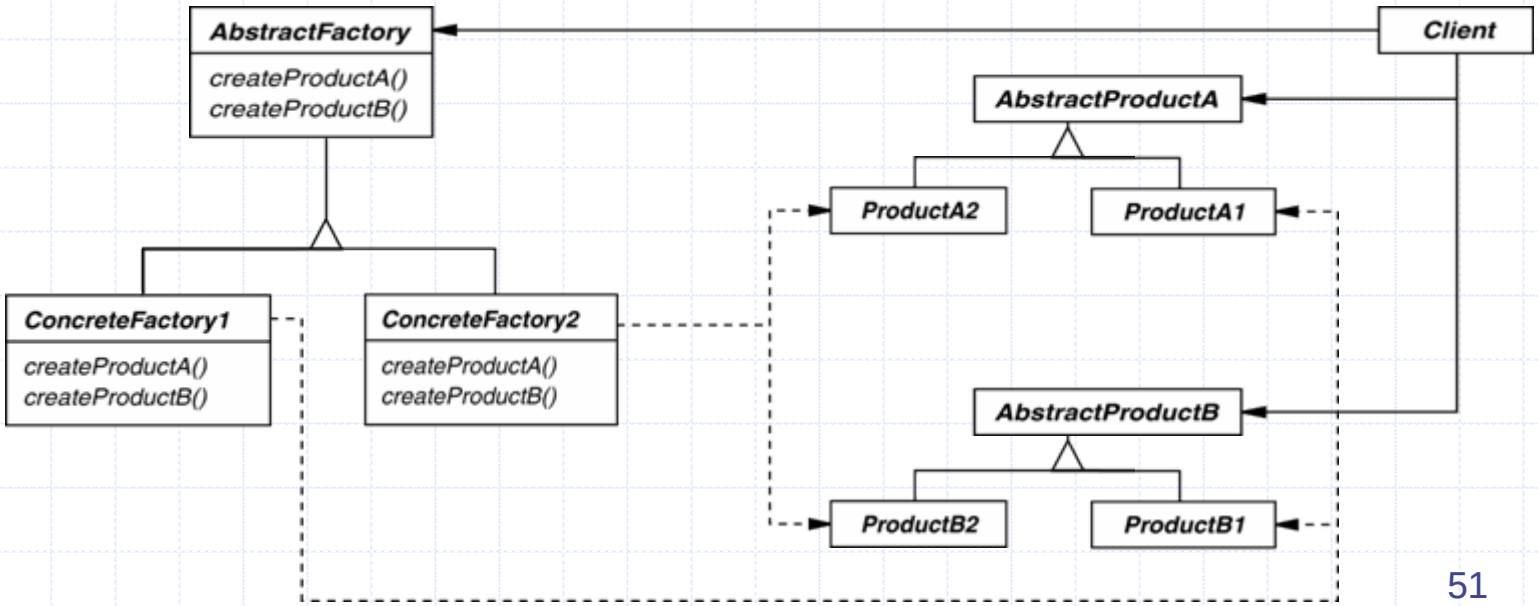
Intent

create families of related objects without specifying subclass names

Applicability

when clients cannot anticipate groups of classes to instantiate

Structure



Multiple Look & Feels (cont'd)

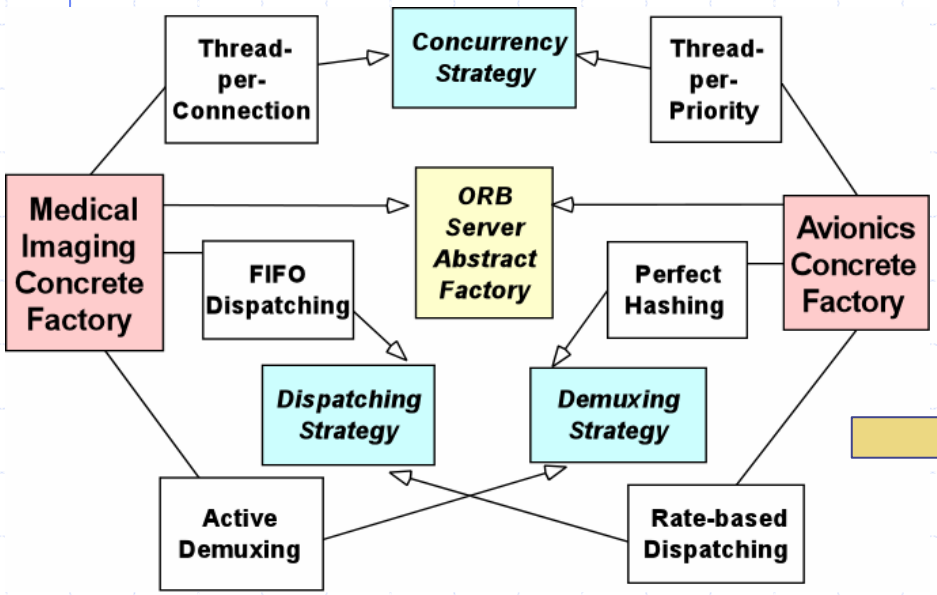
ABSTRACT FACTORY — object creational

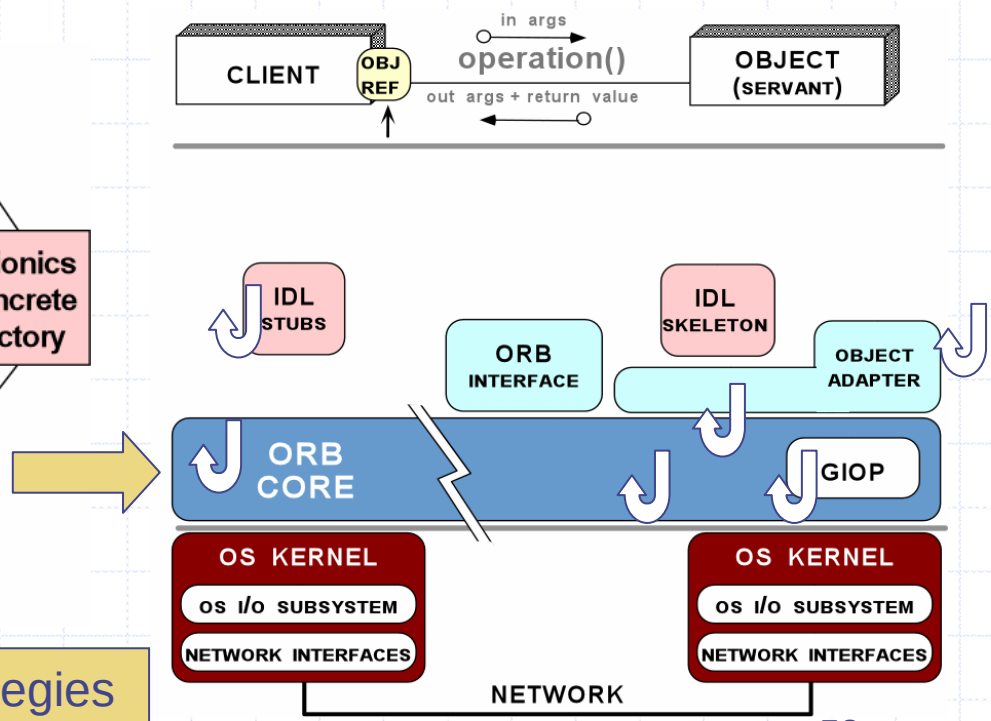
```
class CompositionFactory {
public:
  virtual Compositor *create_compositor() =
    0;
  // ...
};

class TexCompositionFactory :
  public CompositionFactory {
{
public:
  virtual Compositor *create_compositor() {
    return new TexCompositor;
  }
};

CompositionFactory *
create_composition_factory
  (const std::string &factory)
{
  if (factory == "TexCompositor")
    return new TexCompositionFactory;
  else
    ...
}
```

```
class Compositor {
public:
  void set_context
    (Composition &context);
  virtual void compose () = 0;
  // ...
};

class SimpleCompositor
  : public Compositor {
public:
  virtual void compose ()
  { /* ... */}
};

class TexCompositor
  : public Compositor {
public:
  virtual void compose ()
  { /* ... */}
};
```

Multiple Look & Feels (cont'd)

ABSTRACT FACTORY — object creational





Concrete factories create groups of strategies

ABSTRACT FACTORY (cont'd) object creational

Consequences

+ flexibility: removes type (i.e., subclass) dependencies from clients
+ abstraction & semantic checking: hides product's composition
– hard to extend factory interface to create new products

Implementation

- parameterization as a way of controlling interface size
- configuration with Prototypes, i.e., determines who creates the factories
- abstract factories are essentially groups of factory methods

Known Uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- The ACE ORB (TAO)

Multiple Window Systems

Goals:

- make composition appear in a window
- support multiple window systems

Constraints/forces:

- minimize window system dependencies in application & framework code

Multiple Window Systems (cont'd)
Solution: Encapsulate Implementation Dependencies

Window

- user-level window abstraction
- displays a glyph (structure)
- window system-independent
- task-related subclasses
  (e.g., IconWindow, PopupWindow)

Multiple Window Systems (cont'd)

`Window` Interface

```
class Window {
public:
    ...
    void iconify();          // window-management
    void raise();
    ...
    void drawLine(...);      // device-independent
    void drawText(...);      // graphics interface
    ...
};
```

Multiple Window Systems (cont'd)

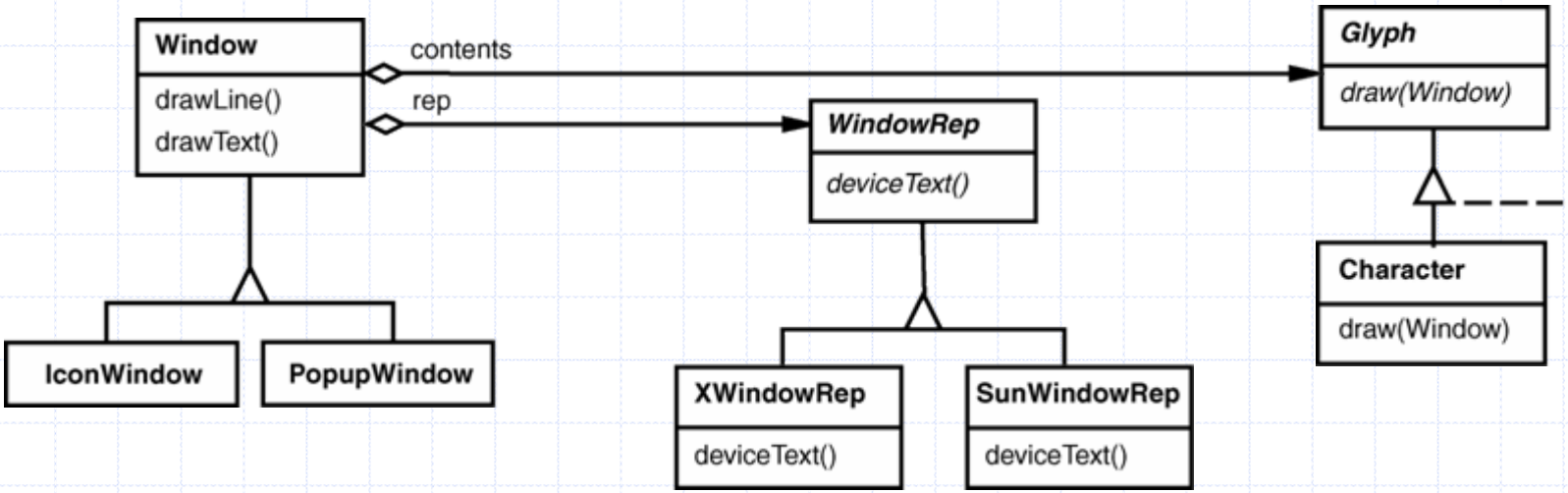
Window uses a **WindowRep**

- abstract implementation interface
- encapsulates window system dependencies
- window systems-specific subclasses (e.g., XWindowRep, SunWindowRep)

An Abstract Factory can produce the right WindowRep!

Multiple Window Systems (cont'd)

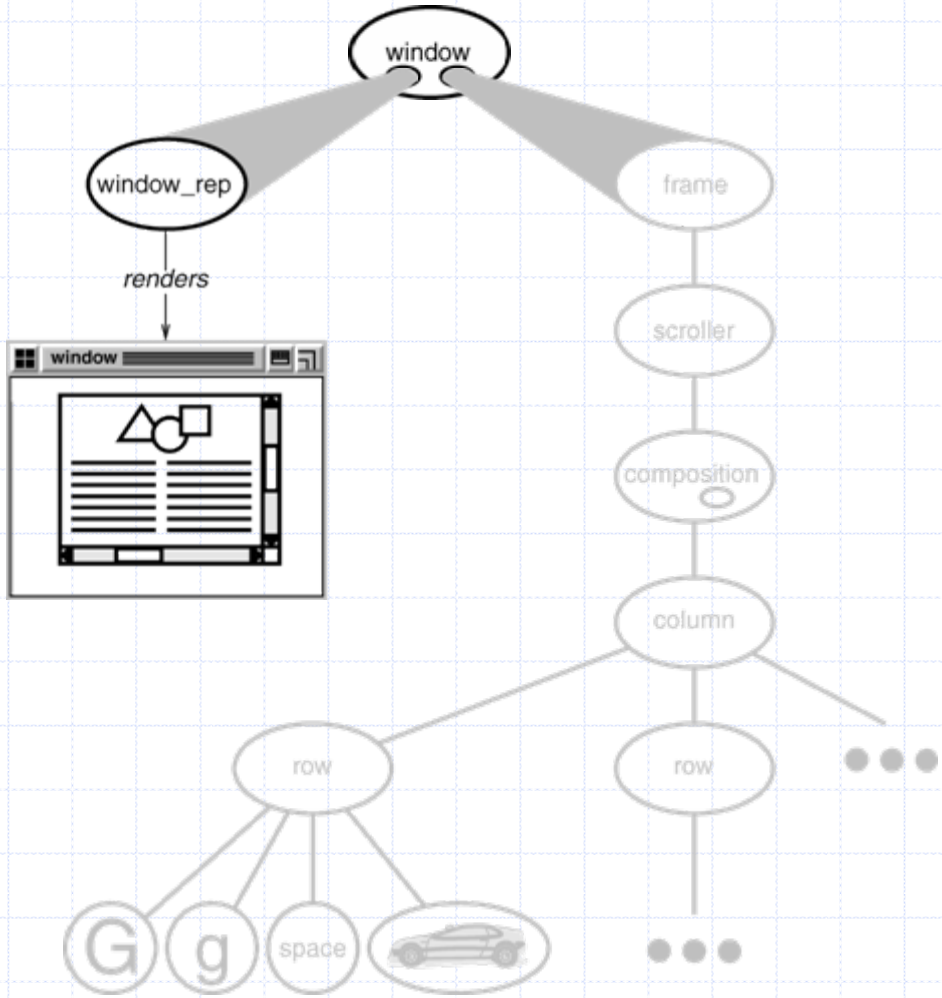
Window/WindowRep Structure




```
void Character::draw (Window &w) {
    w.drawText(...);
}
void Window::drawText (...) {
    rep->deviceText(...);
}
void XWindowRep::deviceText (...) {
    XText(...);
}
```

Multiple Window Systems (cont'd)

New Object Structure



Note the decoupling between the logical structure of the contents in a window from the physical rendering of the contents in the window

Multiple Window Systems (cont'd)

BRIDGE — object structural

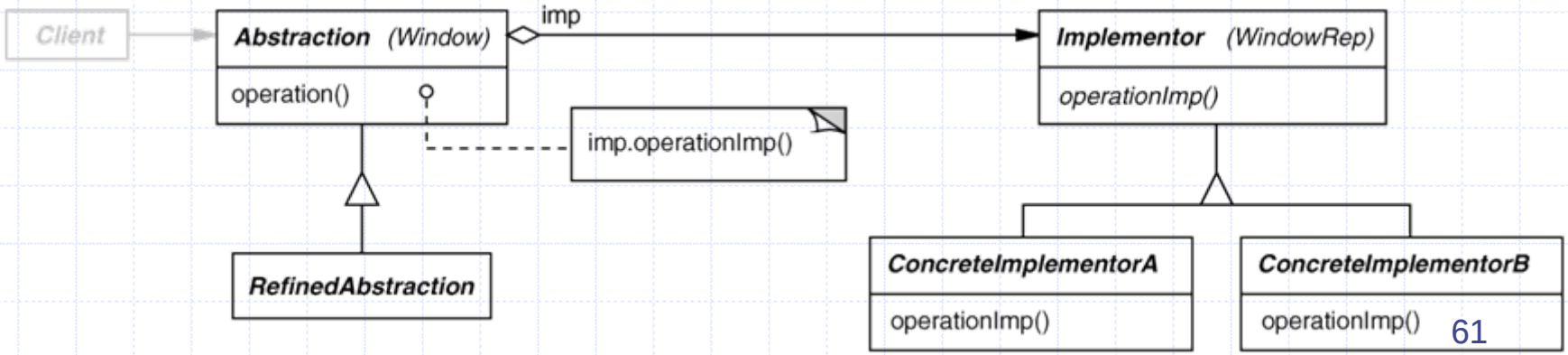
Intent

separate a (logical) abstraction interface from its (physical) implementation(s)

Applicability

- when interface & implementation should vary independently
- require a uniform interface to interchangeable class hierarchies

Structure



Multiple Window Systems (cont'd)

BRIDGE (cont'd) object structural

Consequences

- \+ abstraction interface & implementation are independent
- \+ implementations can vary dynamically
- – one-size-fits-all Abstraction & Implementor interfaces

Implementation

- sharing Implementors & reference counting
- creating the right Implementor (often use factories)

Known Uses

- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- AWT Component/ComponentPeer

User Operations

Goals:

- support execution of user operations
- support unlimited-level undo/redo

Constraints/forces:

- scattered operation implementations
- must store undo state
- not all operations are undoable

User Operations (cont'd)

Solution: Encapsulate Each Request

A **Command** encapsulates

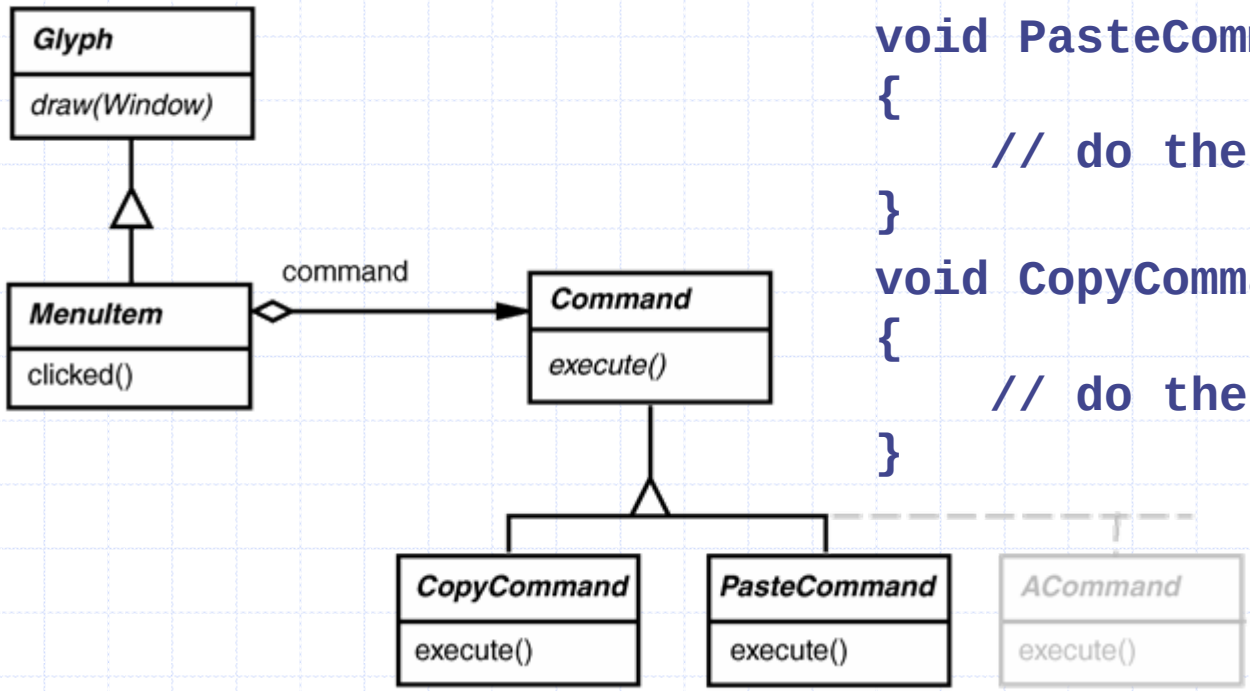
- an operation (**`execute()`**)
- an inverse operation (**`unexecute()`**)
- a operation for testing reversibility (**`boolean reversible()`**)
- state for (un)doing the operation

Command may

- implement the operations itself, *or*
- delegate them to other object(s)

User Operations (cont'd)

Command Hierarchy




```
void PasteCommand::execute ()
{
    // do the paste
}
void CopyCommand::execute ()
{
    // do the copy
}
```

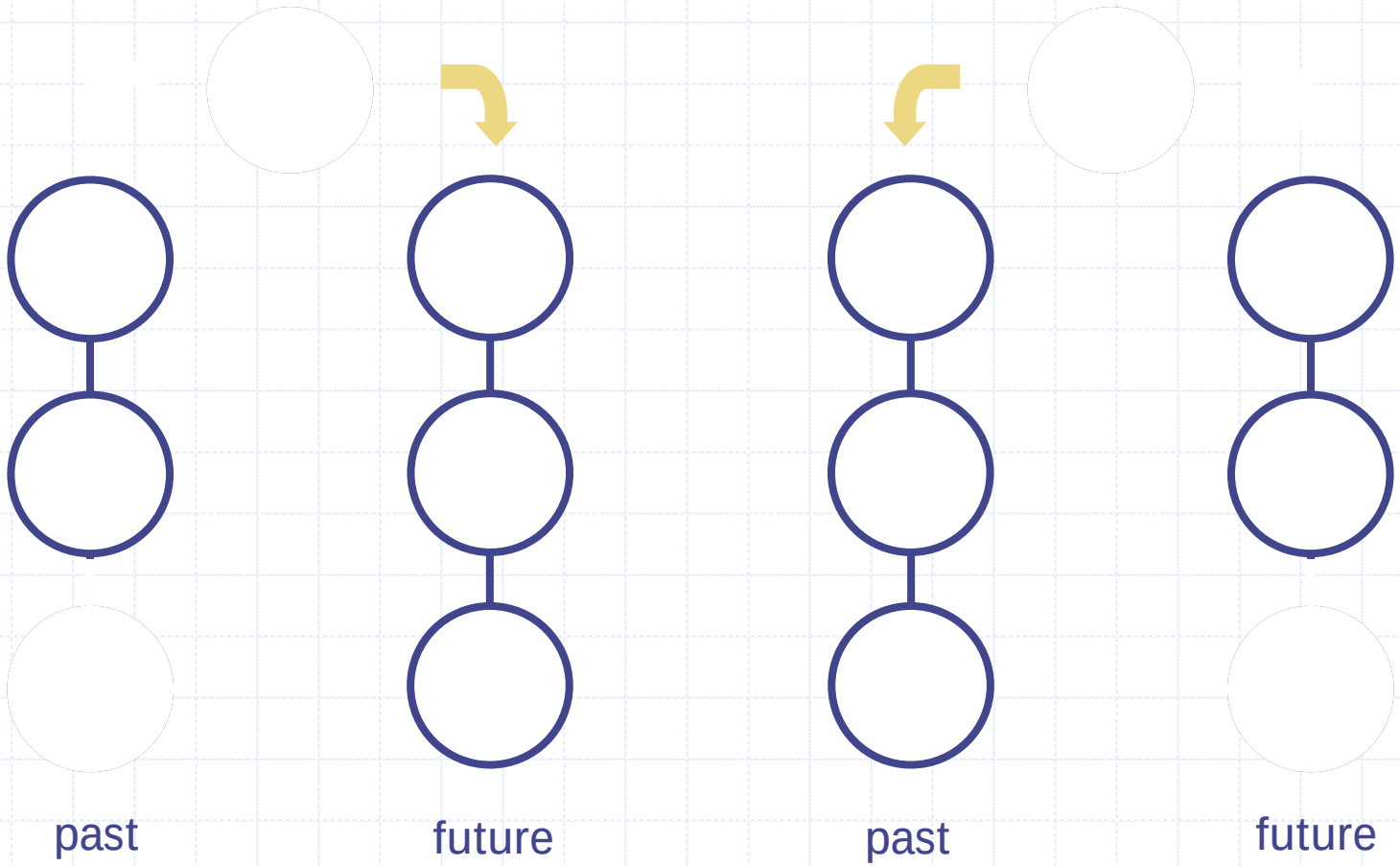
```
void MenuItem::clicked ()
{
    command->execute();
}
```

User Operations (cont'd)

List of Commands = Execution History

Undo:

Redo:



User Operations (cont'd)

COMMAND — object behavioral

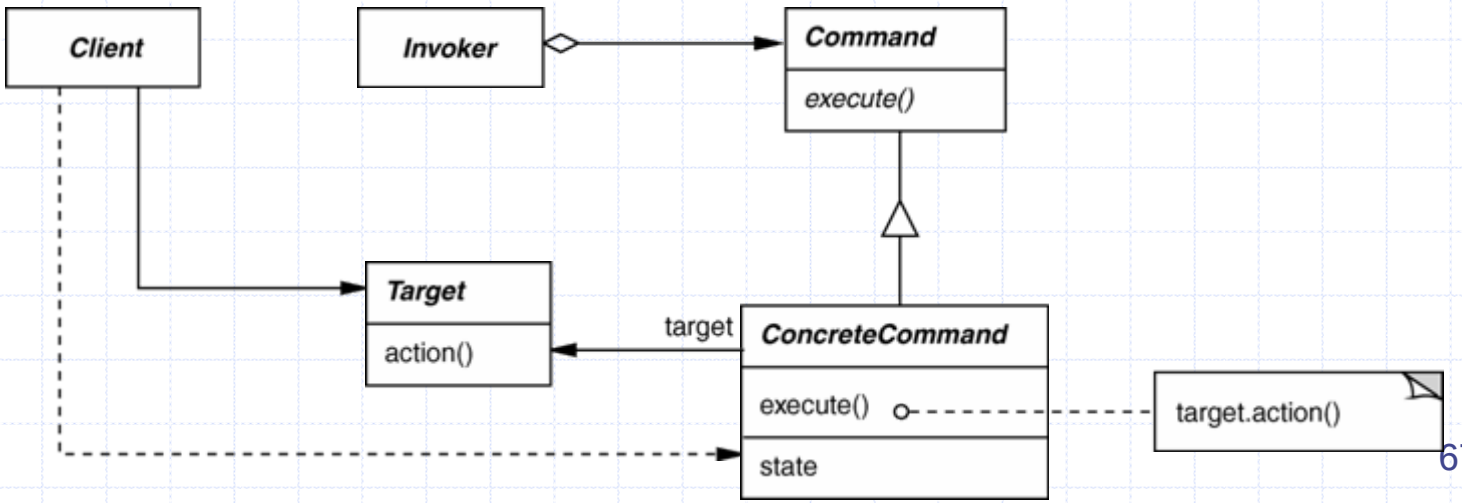
Intent

encapsulate the request for a service

Applicability

- to parameterize objects with an action to perform
- to specify, queue, & execute requests at different times
- for multilevel undo/redo

Structure



COMMAND (cont'd) object behavioral

Consequences

+ abstracts executor of a service
+ supports arbitrary-level undo-redo
+ composition yields macro-commands
– might result in lots of trivial command subclasses

Implementation

- copying a command before putting it on a history list
- handling hysteresis
- supporting transactions

Known Uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- Emacs

Spelling Checking & Hyphenation

Goals:

- analyze text for spelling errors
- introduce potential hyphenation sites

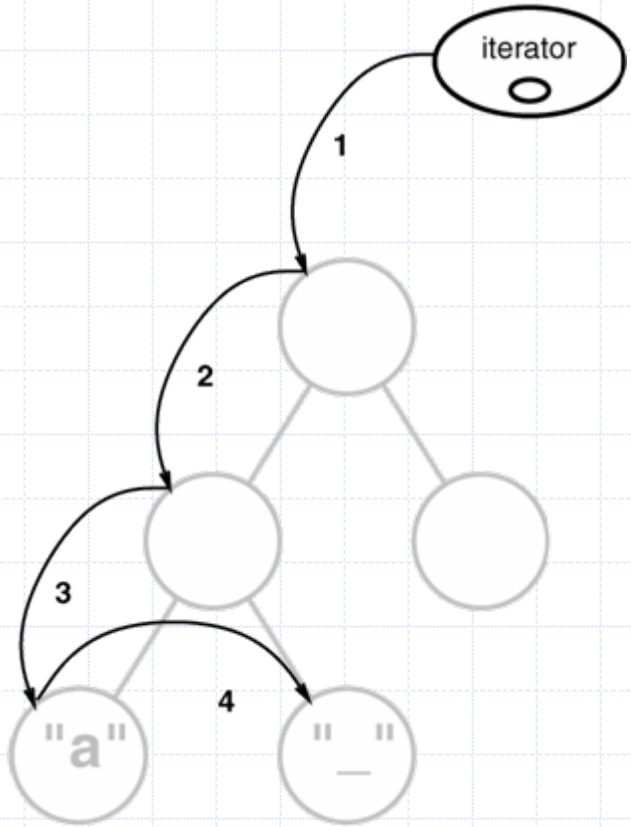
Constraints/forces:

- support multiple algorithms
- don’t tightly couple algorithms with document structure

Solution: Encapsulate Traversal

Iterator

- encapsulates a traversal algorithm without exposing representation details to callers
- uses Glyph's child enumeration operation
- This is an example of a "preorder iterator"



Spelling Checking & Hyphenation (cont'd)

ITERATOR — object behavioral

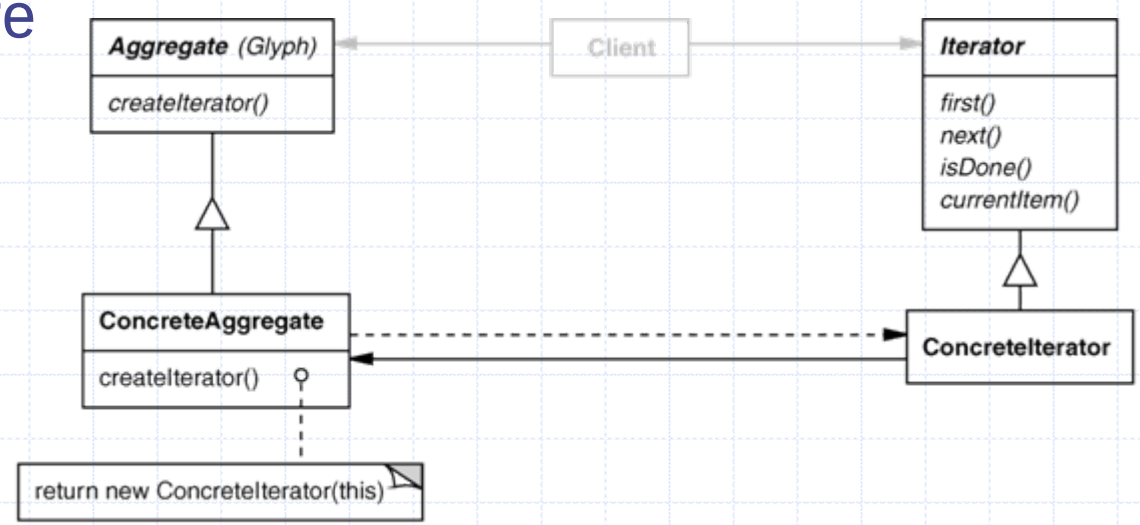
Intent

access elements of a container without exposing its representation

Applicability

- require multiple traversal algorithms over a container
- require a uniform traversal interface over different containers
- when container classes & traversal algorithm must vary independently

Structure



ITERATOR (cont'd) object behavioral

Iterators are used heavily in the C++ Standard Template Library (STL)

```
int main (int argc, char *argv[]) {
   vector<string> args;
   for (int i = 0; i < argc; i++)
       args.push_back (string (argv[i]));
   for (vector<string>::iterator i (args.begin ());
        i != args.end ();
        i++)
     cout << *i;
   cout << endl;
   return 0;
}
```

The same iterator pattern can be applied to any STL container!

```
for (Glyph::iterator  i = composition.begin ();
     i != composition.end ();
     i++)
   ...
```

Spelling Checking & Hyphenation (cont'd)

ITERATOR (cont'd) — object behavioral

Consequences

+ flexibility: aggregate & traversal are independent
+ multiple iterators & multiple traversal algorithms
– additional communication overhead between iterator & aggregate

Implementation

- internal versus external iterators
- violating the object structure's encapsulation
- robust iterators
- synchronization overhead in multi-threaded programs
- batching in distributed & concurrent programs

Known Uses

- C++ STL iterators
- JDK Enumeration, Iterator
- Unidraw Iterator

Spelling Checking & Hyphenation (cont'd)

Visitor

- defines action(s) at each step of traversal
- avoids wiring action(s) into Glyphs
- iterator calls glyph's **`accept(Visitor)`** at each node
- **`accept()`** calls back on visitor (a form of "static polymorphism" based on method overloading by type)

```
void Character::accept (Visitor &v) { v.visit (*this); }

class Visitor {
public:
    virtual void visit (Character &);
    virtual void visit (Rectangle &);
    virtual void visit (Row &);
    // etc. for all relevant Glyph subclasses
};
```

Spelling Checking & Hyphenation (cont'd)

SpellingCheckerVisitor

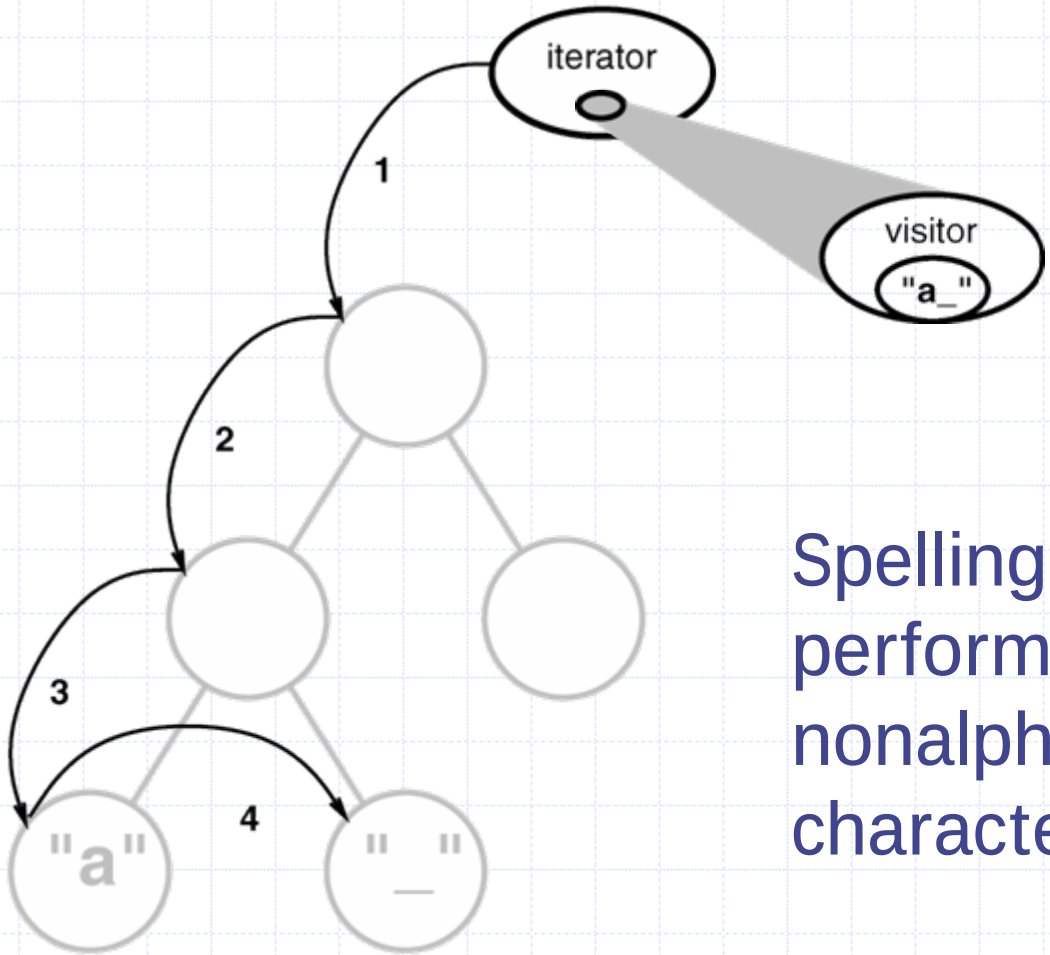
- gets character code from each character glyph

  Can define **`getCharCode()`** operation just on **`Character()`** class
- checks words accumulated from character glyphs
- combine with **PreorderIterator** traversal algorithm

```
class SpellCheckerVisitor : public Visitor {
public:
    virtual void visit (Character &);
    virtual void visit (Rectangle &);
    virtual void visit (Row &);
    // etc. for all relevant Glyph subclasses
Private:
  std::string accumulator_;
};
```

Spelling Checking & Hyphenation (cont'd)

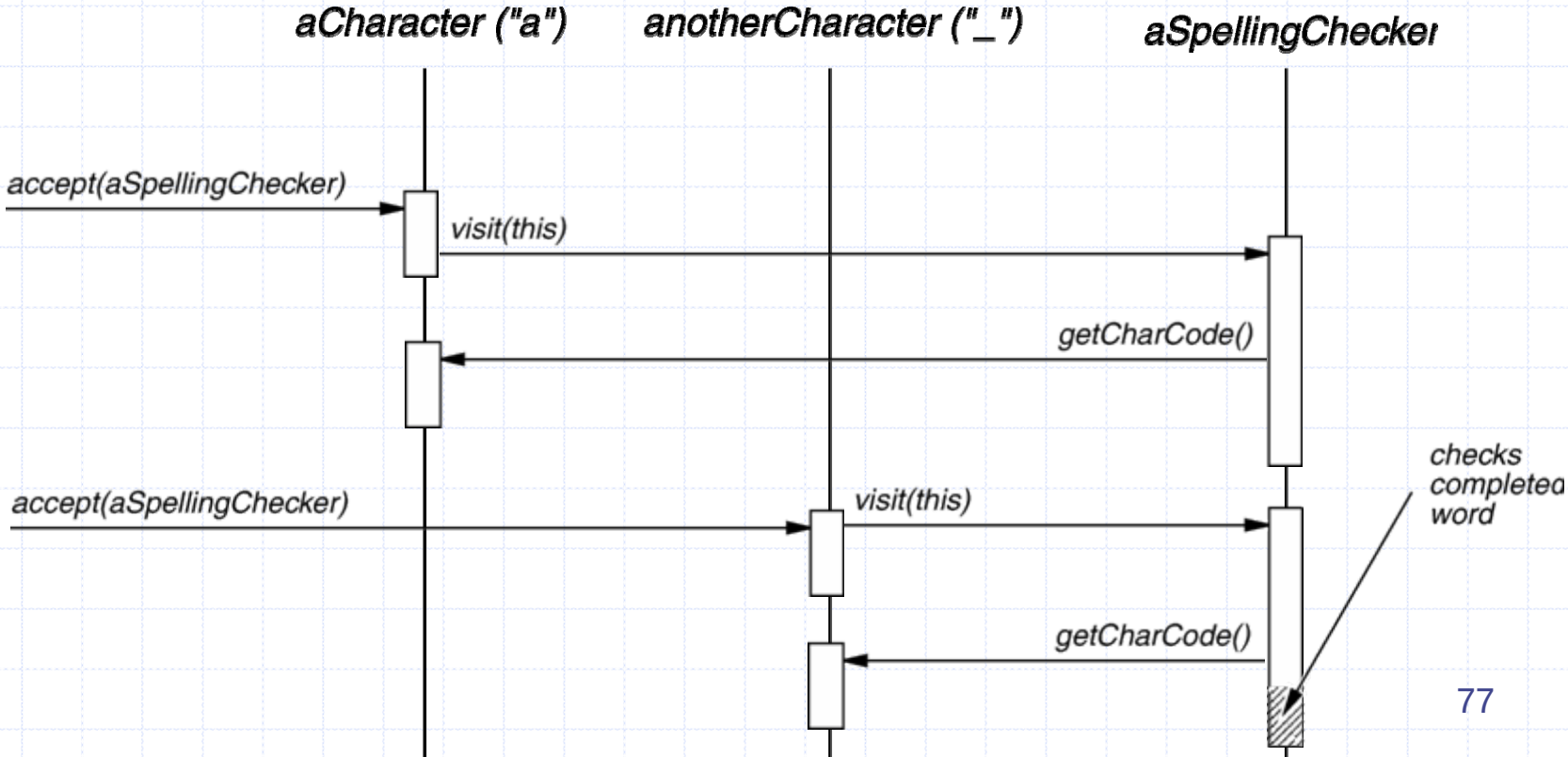
Accumulating Words



Spelling check performed when a nonalphabetic character it reached

Interaction Diagram

- The iterator controls the order in which accept() is called on each glyph in the composition
- accept() then "visits" the glyph to perform the desired action
- The Visitor can be subclassed to implement various desired actions



Spelling Checking & Hyphenation (cont'd)

HyphenationVisitor

- gets character code from each character glyph
- examines words accumulated from character glyphs
- at potential hyphenation point, inserts a...

```
class HyphenationVisitor : public Visitor {
public:
    void visit (Character &);
    void visit (Rectangle &);
    void visit (Row &);
    // etc. for all relevant Glyph subclasses
};
```

Spelling Checking & Hyphenation (cont'd)

Discretionary Glyph

- looks like a hyphen when at end of a line
- has no appearance otherwise
- Compositor considers its presence when determining linebreaks

aluminum alloy *or* aluminum al-
loy

Spelling Checking & Hyphenation (cont'd)

VISITOR — object behavioral

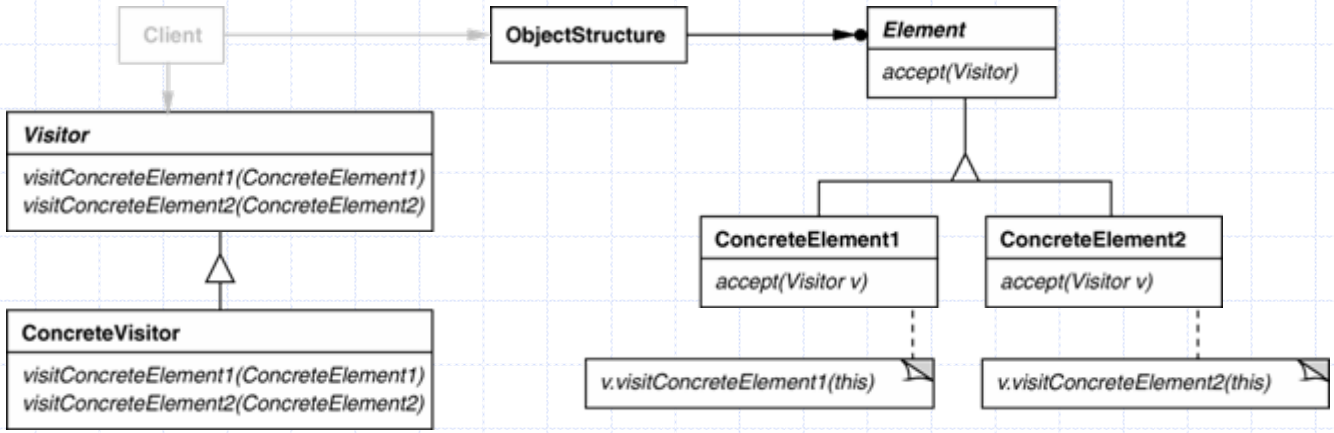
Intent

centralize operations on an object structure so that they can vary independently but still behave polymorphically

Applicability

- when classes define many unrelated operations
- class relationships of objects in the structure rarely change, but the operations on them change often
- algorithms keep state that's updated during traversal

Structure



VISITOR (cont'd) object behavioral

```
SpellCheckerVisitor spell_check_visitor;

for (Glyph::iterator i = composition.begin ();
     i != composition.end ();
     i++) {
   (*i)->accept (spell_check_visitor);
}

HyphenationVisitor hyphenation_visitor;

for (Glyph::iterator i = composition.begin ();
     i != composition.end ();
     i++) {
   (*i)->accept (hyphenation_visitor);
}
```

VISITOR (cont'd) object behavioral

Consequences

+ flexibility: visitor & object structure are independent
+ localized functionality
– circular dependency between Visitor & Element interfaces
– Visitor brittle to new ConcreteElement classes

Implementation

- double dispatch
- general interface to elements of object structure

Known Uses

- ProgramNodeEnumerator in Smalltalk-80 compiler
- IRIS Inventor scene rendering
- TAO IDL compiler to handle different backends

Part III: Wrap-Up

Observations

Patterns are applicable in all stages of the OO lifecycle

- analysis, design, & reviews
- realization & documentation
- reuse & refactoring

Patterns permit design at a more abstract level

- treat many class/object interactions as a unit
- often beneficial *after* initial design
- targets for class refactorings

Variation-oriented design

- consider what design aspects are variable
- identify applicable pattern(s)
- vary patterns to evaluate tradeoffs
- repeat

But...

- Don't apply them blindly
  - Added indirection can yield increased complexity, cost
- Resist branding everything a pattern
  - Articulate specific benefits
  - Demonstrate wide applicability
  - Find at least *three* existing examples from code other than your own!
- Pattern design even harder than OO design!

Part III: Wrap-Up (cont'd)

Concluding Remarks

- *design* reuse
- uniform design vocabulary
- understanding, restructuring, & team communication
- provides the basis for automation
- a "new" way to think about design

Pattern References

Books

Timeless Way of Building, Alexander, ISBN 0-19-502402-8

A Pattern Language, Alexander, 0-19-501-919-9

Design Patterns, Gamma, et al., 0-201-63361-2 CD version 0-201-63498-8

Pattern-Oriented Software Architecture, *Vol. 1*, Buschmann, et al., 0-471-95869-7

Pattern-Oriented Software Architecture, Vol. 2, Schmidt, et al., 0-471-60695-2

Pattern-Oriented Software Architecture, Vol. 3, Jain & Kircher, 0-470-84525-2

Pattern-Oriented Software Architecture, Vol. 4, Buschmann, et al., 0-470-05902-8

Pattern-Oriented Software Architecture, Vol. 5, Buschmann, et al., 0-471-48648-5

Pattern References (cont'd)

More Books

Analysis Patterns, Fowler; 0-201-89542-0

Concurrent Programming in Java, 2nd ed., Lea, 0-201-31009-0

Pattern Languages of Program Design

Vol. 1, Coplien, et al., eds., ISBN 0-201-60734-4
Vol. 2, Vlissides, et al., eds., 0-201-89527-7
Vol. 3, Martin, et al., eds., 0-201-31011-2
Vol. 4, Harrison, et al., eds., 0-201-43304-4
Vol. 5, Manolescu, et al., eds., 0-321-32194-4

AntiPatterns, Brown, et al., 0-471-19713-0

Applying UML & Patterns, 2nd ed., Larman, 0-13-092569-1

Pattern Hatching, Vlissides, 0-201-43293-5

The Pattern Almanac 2000, Rising, 0-201-61567-3

Pattern References (cont'd)

Even More Books

Small Memory Software, Noble & Weir, 0-201-59607-5

Microsoft Visual Basic Design Patterns, Stamatakis, 1-572-31957-7

Smalltalk Best Practice Patterns, Beck; 0-13-476904-X

The Design Patterns Smalltalk Companion, Alpert, et al., 0-201-18462-1

Modern C++ Design, Alexandrescu, ISBN 0-201-70431-5

Building Parsers with Java, Metsker, 0-201-71962-2

Core J2EE Patterns, Alur, et al., 0-130-64884-1

Design Patterns Explained, Shalloway & Trott, 0-201-71594-5

The Joy of Patterns, Goldfedder, 0-201-65759-7

The Manager Pool, Olson & Stimmel, 0-201-72583-5

Pattern References (cont'd)

Early Papers

"Object-Oriented Patterns," P. Coad; Comm. of the ACM, 9/92

"Documenting Frameworks using Patterns," R. Johnson; OOPSLA '92

"Design Patterns: Abstraction & Reuse of Object-Oriented Design," Gamma, Helm, Johnson, Vlissides, ECOOP '93

Articles

Java Report, Java Pro, JOOP, Dr. Dobb's Journal, Java Developers Journal, C++ Report

Pattern-Oriented Conferences

PLoP 2007: Pattern Languages of Programs October 2007, Collocated with OOPSLA

EuroPLoP 2008, July 2008, Kloster Irsee, Germany

...

See hillside.net/conferences/ for up-to-the-minute info.

Mailing Lists

patterns@cs.uiuc.edu: present & refine patterns

patterns-discussion@cs.uiuc.edu: general discussion

gang-of-4-patterns@cs.uiuc.edu: discussion on *Design Patterns*

siemens-patterns@cs.uiuc.edu: discussion on *Pattern-Oriented Software Architecture*

ui-patterns@cs.uiuc.edu: discussion on user interface patterns

business-patterns@cs.uiuc.edu: discussion on patterns for business processes

ipc-patterns@cs.uiuc.edu: discussion on patterns for distributed systems

See http://hillside.net/patterns/mailing.htm for an up-to-date list.