CS 472/572: Object-Oriented Design Patterns
Slides

Chapter 4: fe Scanning and Parsing

CS 472/572: Object-Oriented Design Patterns

- Roster and passwords
- Our `pub` directory:

  `onyx:~jbuffenb/classes/472/pub`

  `pub/ch1/intf/Table.java`

  Canvas

  GitHub
- Our lecture slides, table of contents, and code:

  `pub/slides/slides.pdf`

  `pub/slides/code.tar`
- Review syllabus

  `http://csweb.boisestate.edu/~buff`

  `pub/syllabus/syllabus.pdf`
- Introduction (textbook)

Introduction (1 of 3)

- Our goal is to learn how to design *good* object-oriented (OO) software systems.
- By "good," we mean reusable and flexible.
- How do we "reuse" software?
  - Assuming it is modular, we can copy and edit a module.
  - We can use a module without changing it.
  - We can refer to it via acquaintance.
  - We can inherit from it.

Introduction (2 of 3)

- We want to write source files that require *no* modification when used in future systems. We don't want to have to add members or change types.
- We'll admit that OO design is difficult, and that creating reusable designs is even more difficult.
- How do you decide on classes and methods? Hopefully, you consider nouns and verbs in the requirements.
- For reusable designs, we'll need other classes, which do not correspond to nouns in the problem domain.
- The OO patterns that we'll study tell us what the other classes are.

Introduction (3 of 3)

- Historically, a design should not be called reusable until it has been used, without modification, several times.
- The OO design patterns that we'll study are each the result of iterations that have produced reusable designs. By understanding them, our "first" designs can be reusable.
- If you don't need reusable designs, you don't want to use these patterns, because they add complexity.

What is a design pattern?

- Our OO patterns are modeled after Christopher Alexander's towns, buildings, and construction patterns:

  `pub/ch1/ca-cover.txt`

  `pub/ch1/ca-pattern.txt`
- Example: *Light on Two Sides of Every Room (159)*:

  `pub/ch1/p159.0.pdf`

  `pub/ch1/p159.1.pdf`

  `pub/ch1/p159.2.pdf`
- The idea of categorizing programming patterns was also considered by Charles Rich and Richard Waters, at MIT, in the late 1970's. Their Programmer's Apprentice project maintained a database of plans or cliches.

Describing design patterns (1 of 3)

- Each OO pattern has several essential parts:
  - pattern name: for example, Iterator
  - problem: when the pattern applies
  - solution: how the pattern addresses the problem
  - consequences: pros and cons of using the pattern
- Most of our textbook is a catalog of patterns.

Describing design patterns (2 of 3)

- Each catalog entry has the following sections:
  - Name (and aliases)
  - Motivation (with examples)
  - Applicability
  - Structure (in UML: Unified Modeling Language)
  - Participants (abstract and according to a Motivation example)
  - Collaborations
  - Consequences
  - Implementation
  - Sample Code (usually, C++)
  - Known Uses (usually, graphical frameworks)
  - Related Patterns (similar and dependent patterns)

Describing design patterns (3 of 3)

- The patterns are abstract and mostly language independent, though some patterns require particular features, like exception-handling. Therefore, any OO language could be used for examples.
- Our textbook uses C++ and Smalltalk.
- I'll use Java 1.5, or later, for several reasons:
  - It clearly distinguishes interface inheritance from implementation inheritance. C++ requires multiple inheritance, while Java permits only multiple interface inheritance.
  - It supports all four kinds of polymorphism, to be described soon.
  - It is the language of CS 121 and 221.

Polymorphism in Java and C++ (1 of 2)

- *Polymorphism* allows one type to be used as another type.
- This broad definition also allows a value of one type to be used as a value of another type.
- Since our values are objects, and a type is just the name of an interface, polymorphism allows an object with one interface to be used as an object with another interface.
- For example, a `Student` object might be used as a `Person` object, by changing its marital status.

Polymorphism in Java and C++ (2 of 2)

- Polymorphism can be categorized (a taxonomy of taxonomy), as follows:
  - ad hoc:
    - coercion
    - overloading
  - universal:
    - parametric (generics)
    - inclusion (inheritance)

Ad-hoc polymorphism in Java and C++ (1 of 3)

- Coercion in C++ can be implicit (i.e., a conversion) or explicit (i.e., a cast).
- The cast operator can be overloaded.
- C++ has several kinds of cast. We'll only consider the (compile-time) C-style cast and `static_cast<T>()`. According to the *C++ FAQ*:

  > The C++ `static_cast<T>()` is better than C-style casting because it stands out in the code and explicitly states the programmer's understanding and intentions. It also understands and respects `const` and access controls.

- For example:

  pub/ch1/poly/coerc/Coercion.c++

- Welcome to C++!

Ad-hoc polymorphism in Java and C++ (2 of 3)

- In Java, coercion tries to be "safer" than in C++.
- Java has no user-defined operator overloading.
- For example:

  pub/ch1/poly/coerc/Coercion.java

Ad-hoc polymorphism in Java and C++ (3 of 3)

- Both languages allow regular (non-operator) methods to be overloaded.
- Candidates are distinguished by argument types, but not return type.
- A Java example:
  `pub/ch1/poly/overl/Overloading.java`
- A C++ example:
  `pub/ch1/poly/overl/Overloading.c++`

Universal polymorphism in Java and C++ (1 of 4)

- In an OO system, universal polymorphism is far more useful than coercion and overloading.
- Parametric polymorphism is available in Java and C++ as generics and templates, respectively. It allows the design of homogeneous containers. This is a compile-time mechanism.
- A C++ template parameter can also be a value (e.g., an integer), rather than a class/type. A parameter can also have a default value.
- A Java example:

  pub/ch1/poly/param/Parametric.java
  pub/ch1/poly/param/Pair.java
- A C++ example:

  pub/ch1/poly/param/Parametric.c++
  pub/ch1/poly/param/Pair.h

Aside: C++ alternatives

- In C++, a complete module that is only a single `.h` file is nice for slide shows. However, a *real* module has separate interface and implementation files:

  `pub/ch1/poly/param/Pair1.h`

  `pub/ch1/poly/param/Pair1.c++`
- In C++, a variable can be declared, defined, and used in more than one way:

  `pub/ch1/poly/param/VarZoo.c++`

Universal polymorphism in Java and C++ (2 of 4)

- Inclusion polymorphism is due to inheritance and dynamic binding.
- It allows the design of heterogeneous containers.
- This is an execution-time mechanism.
- A Java example:

  `pub/ch1/poly/inclu/Inclusion.java`

  `pub/ch1/poly/inclu/InclRand.java`
- Oddly, in C++ the overridden method must be explicitly declared to be a virtual function:

  `pub/ch1/poly/inclu/Inclusion.c++`
- In Java, classes form a tree; in C++, a forest, or even an acyclic graph.

Universal polymorphism in Java and C++ (3 of 4)

- Often, the superclass has methods with no implementation; they must be provided by subclasses.
- Of course, the superclass can still be parameterized.
- In Java, such a class is called an interface, where every method is public and abstract.
- A Java example:

  pub/ch1/poly/inclu/Box.java
  pub/ch1/poly/inclu/BoxMagic.java
  pub/ch1/poly/inclu/BoxPlain.java
  pub/ch1/poly/inclu/TryBox.java

Universal polymorphism in Java and C++ (4 of 4)

- In C++, such a class is called an abstract base class (aka, an ABC), where at least one method is a pure virtual function. A body of =0 makes it pure: a subclass must provide a real body.
- A C++ example:

  pub/ch1/poly/inclu/Box.h

  pub/ch1/poly/inclu/BoxMagic.h

  pub/ch1/poly/inclu/BoxPlain.h

  pub/ch1/poly/inclu/TryBox.c++

Constructors et al. in Java and C++ (1 of 2)

- C++ classes often need a destructor to deallocate memory for instance variables.
- A C++ destructor's name is the same as the constructor, but prefixed with a twiddle (i.e., ~).
- Since Java has garbage collection, destructors, named `finalize()`, are rarely needed.
- C++ has assignment operators and copy constructors, which are similar to Java `clone()` methods.

Constructors et al. in Java and C++ (2 of 2)

- Several relevant coding guidelines for C++ can be found at:
  https://isocpp.org/faq
- Here is an example demonstrating them:
  pub/ch1/Foo.c++

How Design Patterns Solve Design Problems (1 of 2)

- Design patterns help you decompose a system into objects:
  - An object encapsulates data, and methods that operate on the data.
  - We'll expect an object's data to be private. The only way to change it is via a method.
  - OO design methodologies typically produce designs where objects correspond to nouns in the problem statement: real-world entities.
  - But, good designs often end up with objects that have no real-world counterpart (e.g., Adapter).
  - Patterns help us include these objects in our designs, increasing flexibility and reusability.

How design patterns solve design problems (2 of 2)

- They help us determine object granularity:
  - Objects can be tiny or huge. We can need one object of a class, or thousands.
  - Some patterns describe objects that create objects, compose objects, or implement requests between objects.
- They help us specify object interfaces:
  - A method's name, argument types, return type, and exceptions compose its *signature*.
  - The set of public-method signatures defined by an object is its *interface*.
  - A *type* is a name denoting an interface.

Types and interfaces (1 of 5)

- For example, in Java:
  pub/ch1/intf/Table.java
  pub/ch1/intf/HashTable.java
  pub/ch1/intf/TryTable.java
- For example, in C++:
  pub/ch1/intf/Table.h
  pub/ch1/intf/HashTable.h
  pub/ch1/intf/TryTable.cc

  or, with references:
  pub/ch1/intf/TryTableRef.cc

Types and interfaces (2 of 5)

- In the previous three table examples:
  - The type is Table.
  - The interface has two signatures.
  - Since every non-member interface is public, don't use the `public` modifier, unless you are using packages.
  - Since every interface is abstract, don't use the `abstract` modifier.
  - Why use a pointer, in C++? So we don't make copies when we pass it around. So we can `delete` it when we want.

Types and interfaces (3 of 5)

- Objects constructed from different classes may have the same interface. For example:

  pub/ch1/intf/Stack.java

  pub/ch1/intf/Queue.java

- Alternatively, an object's interface may include the interfaces named by multiple types. For example, a List interface may provide methods for Stack and Queue access:

  pub/ch1/intf/List.java

Types and interfaces (4 of 5)

- Since a type is the name of a set, we can say: "If a type is a *subtype* of a *supertype*, the supertype's interface is a subset of the subtype's interface." In other words, subtyping can enlarge an interface.
- A subtype *inherits* (aka, *extends*) the interface of its supertype.
- An object is used only via its interface.
- Ignoring reflection, or run-time type identification, an object cannot be queried, or asked to do anything, without going through its interface.
- An interface does not provide an implementation.

Types and interfaces (5 of 5)

- Objects with identical interfaces can have different implementations (e.g., Stack and Queue).
- An object A can send a request to an object B by invoking a method whose signature is part of a type T, which is part of B's interface.
  - A need not know B's complete interface, just that T is part of it.
  - The association of A's request to B's method is called *dynamic binding*.

Dynamic binding

- We saw an earlier example. Here's another:

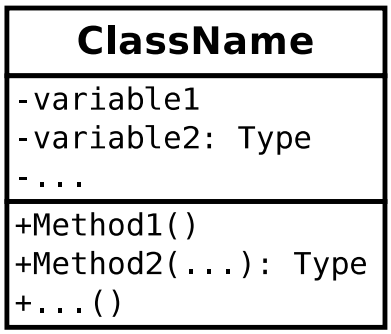
  pub/ch1/DynamicBinding.java
- This substitutability is a form of *polymorphism*, called *inheritance polymorphism* (aka, inclusion polymorphism).

How design patterns solve design problems (summary)

- Patterns suggest what belongs and does not belong in an interface, to establish appropriate coupling between objects.
- Patterns specify relationships between interfaces: requiring similarity or placing constraints.

Specifying object implementations (1 of 4)

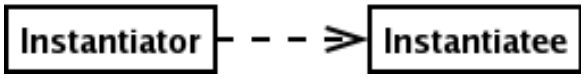
- An object's implementation is defined by a *class*, which specifies internal (private) data and defines methods (public and private).
- Our textbook uses a UML-ish notation:



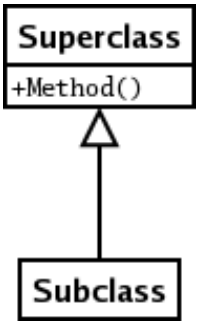
- I use a program named `dia` to edit these diagrams.
- An object is created by *instantiating* a class. This allocates memory for the object's data.

Specifying object implementations (2 of 4)

- A *dashed arrow* from one class to another indicates that an object of the first class instantiates an object of the second class:



- A *subclass* can be declared by inheriting from a superclass. The subclass inherits all methods and data from the parent:

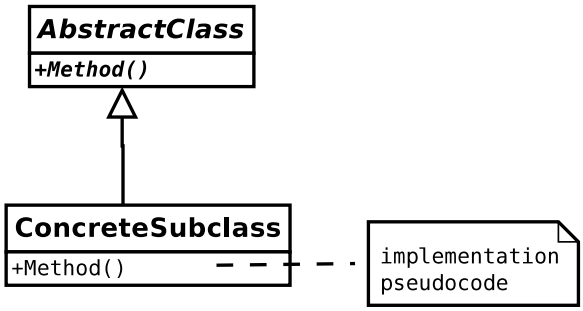


Specifying object implementations (3 of 4)

- An *abstract class* declares a common interface for subclasses (e.g., a Java interface).
- Its methods are *abstract*: declared but not defined (there is no implementation).
- Such a method is also known as *deferred*, because a subclass must implement it before an object of the class can be created.
- A class can have a mixture of abstract and implemented methods.
- A class with no abstract methods is a *concrete* class. Only a concrete class can be instantiated.
- In UML, slanted type denotes an abstract class or method name.
- A method in a subclass can override the method in a superclass.

Specifying object implementations (4 of 4)

- Our diagrams can include pseudocode:



- Our textbook describes a *mixin* class as one that can be inherited to provide "an optional interface or functionality."
- A Java class can `implement` multiple interfaces, but can only `extend` one class. So, we won't have mixin classes.
- A Java interface can `extend` multiple interfaces.

Class versus interface inheritance (1 of 5)

- What's the difference between an object's class and its type?
  - Its class provides its implementation:
    - variables (its state)
    - method bodies (its methods' statements)
  - Its type is just the name of part of its interface: a set of public-method signatures.
- An object can have many types.
- Objects of different classes can have the same type.
- Of course, an object's class must implement the methods named by its types.

Class versus interface inheritance (2 of 5)

- C++ uses classes for both "class" and "type." Pure virtual member functions support "type" specification:
  `pub/ch1/intf/Table.h`
- Java uses classes and interfaces for "class" and "type," respectively.
- This leads us to consider the two kinds of inheritance we'll be using:
  - class (implementation) inheritance: Java's `extends`.
  - type (interface) inheritance: Java's `implements`.
- Some of the design patterns we'll study depend on this distinction.
  - Sometimes, multiple objects are of the same type, but not class.
  - Sometimes, multiple objects are of the same class.

Class versus interface inheritance (3 of 5)

- Class inheritance is a quick and simple way to reuse a parent's functionality.
- Type inheritance allows us to define families of objects with different classes, but identical types.
- Type inheritance provides an important benefit: A client is unaware of the classes of objects it uses, as long as the objects adhere to the interface that the client expects.

Class versus interface inheritance (4 of 5)

- This benefit requires that a subclass only add to, or override, methods of its parent. It must not hide methods of its parent.
- C++ allows this, but Java does not:
  `pub/ch1/Penguin.cc`
  `pub/ch1/Penguin.java`
- C++ goes even further: the 2011 standard allows a method to be "deleted," with `foo()=delete;`, but not an inherited virtual function, like `fly`.

Class versus interface inheritance (5 of 5)

- This benefit of type inheritance so greatly reduces coupling between subsystems, we'll adopt the First Principle of Object-Oriented Design:

  Program to an interface, not an implementation.
- In other words, don't declare variables (like `t`) to contain references to objects of concrete classes:

  pub/ch1/intf/TryTable.java
- Where should we instantiate concrete classes? We'll see creational patterns that abstract the process of object creation, allowing a system to be written in terms of interfaces.

Putting reuse mechanisms to work

- The two most common forms of OO reuse are inheritance and object composition, which model the is-a and has-a relationship, respectively.
  - With inheritance, the subclass can access the parent's private (with respect to clients) methods and data, so it's called *white-box* or *clear-box* reuse.
  - With composition, clients cannot access private members, so it's called *black-box* reuse.

Reuse by inheritance

- Form:

```
public class Child extends Parent ···
```

- Child reuses members of Parent.
- Reuse occurs statically, at compile time.
- Child can modify the implementation of Parent, by overriding its methods, but only at compile time.
- Parent often defines part of Child's implementation.
- A change to Parent's implementation probably requires a change to Child. Thus, "inheritance breaks encapsulation." One solution is to only inherit from abstract classes.
- This is overused by designers.

Reuse by composition (1 of 2)

- Form:

```
public class Client {
    ...
    private Part p;
    ...
    p=...;
    ...
}
```

- Variable `p` can be assigned a reference to any object of type Part, not just an object of class Part.
- Client reuses members of object referenced by `p`.
- Reuse occurs dynamically, at run time.
- Client can modify the behavior of the object referenced by `p`, at run time, by invoking that object's methods and passing arguments that are objects.

Reuse by composition (2 of 2)

- A change to the implementation of the object referenced by `p` probably does not require a change to Client.
- Client must respect Part's interface. Encapsulation is preserved.
- Classes and class hierarchies remain small.
- A system will have more objects, but fewer classes.
- This leads to more reusable designs.
- The Go language has a nice approach:

  `pub/ch1/reuse.go`
- These ideas form the Second Principle of Object-Oriented Design:

  Favor object composition over class inheritance.

Delegation (1 of 3)

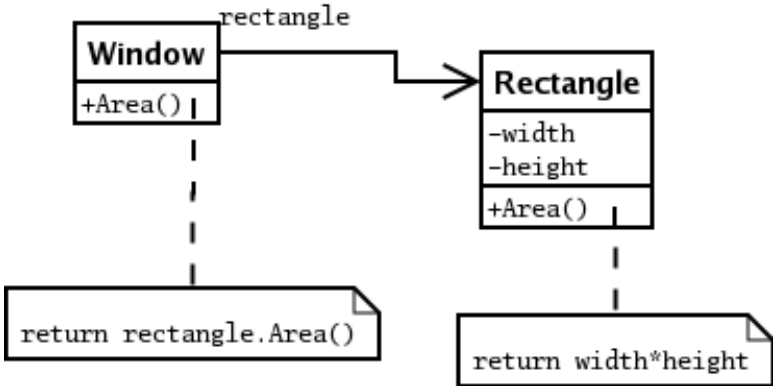
- With inheritance, a subclass can defer requests to a superclass, by not overriding a superclass method.
- Delegation is the object-composition way of doing this. A Delegator object can delegate a request to a Delegate object.
- Thus, composition can always replace inheritance for code reuse.
- Delegation can look like this:

  pub/ch1/Delegator.java

  pub/ch1/Delegate.java

Delegation (2 of 3)

- Here's a another delegation example, that does not use `this`:



- A solid arrow denotes that an object of one class keeps a reference to an object of another class. The label is a variable name.
- Delegation allows us to compose behaviors during execution, and change them during execution.
- Delegation can increase complexity. It should be used according to patterns, only when it simplifies more than it complicates.

Delegation (3 of 3)

- Java's AWT uses delegation.

  pub/ch1/ls/LightSwitch.java

  pub/ch1/ls/ToggleButton.java
- When you click the ToggleButton object, the LightSwitch object (Delegator) delegates the request (`getLabel()`) to the ToggleButton object (Delegate).

Inheritance versus parameterized types (1 of 2)

- Parameterized types are called generics in Java, and templates in C++.
- Parameterized types allow you to define (and implement) a type without specifying all the other types it uses.
- For example, recall our Table type:

  pub/ch1/intf/Table.java
- This is a third way to reuse code.

Inheritance versus parameterized types (2 of 2)

- Consider a sorting routine:
  - We could implement the comparison in a subclass, using inheritance.
  - We could delegate the comparison to another object, using object composition.
  - We could require the class of objects being compared to provide a comparison method, using generics.
- Only object composition allows you to change the comparison method during execution.

Relating run-time and compile-time structures (1 of 3)

- An OO system's static (compile-time) and dynamic (run-time) structures often vary drastically.
- Static relationships between classes are typically due to inheritance and aggregation.
- *Aggregation* is when an object of one class contains an object of another. Their lifetimes (i.e., their extents) are the same.
- Java does not allow this, but C++ does.

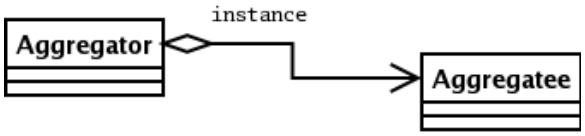
Relating run-time and compile-time structures (2 of 3)

- Dynamic relationships between objects, other than aggregation, are typically due to acquaintance.
- *Acquaintance* is when an object has a reference to another object. Their lifetimes can be independent.
- This is also called *association* or *using*, and is a weaker form of coupling than aggregation.
- These relationships can change during execution.

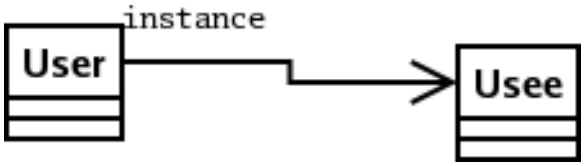
Relating run-time and compile-time structures (3 of 3)

- For example:

  pub/ch1/AggAcq.cc
- Aggregation in UML:



- Acquaintance in UML:



- Compile-time structure can be understood by reading the source code.
- Run-time structure is imposed more by the designer than the language, and isn't usually clear from reading the code, until you understand the patterns.

Designing for change (1 of 2)

- Not surprisingly, you need to anticipate what will change, and design accordingly.
- Common causes of redesign:
  - You create an object by explicitly specifying its implementation. We'll see how to create an object indirectly.
  - You call a method by explicitly specifying its name. We'll see how to call a method indirectly.
  - You depend on a hardware or software platform. We'll see how to insulate an application from the platform.
  - You depend on an object's implementation. We'll see how to decouple an implementation from its interface.
  - You depend on a particular algorithm. We'll see how to encapsulate and isolate an algorithm.

Designing for change (2 of 2)

- You allow tight coupling. We'll see how to use abstract coupling and layering.
- You customize by subclassing. We'll see how to use object composition and subclassing, but this can lead to many classes. We'll see how to add one subclass and compose its instances with objects of existing classes.
- You need to change a class that you can't (e.g., you don't have its source code or you can't risk affecting current clients). We'll see how to wrap such classes, effectively changing them.

Categories of software

- Design patterns are effective in (at least) three broad categories of software:
  - In an application program, you are mainly interested in reusing your own classes.
  - In a *toolkit*, you try to write code that other people can reuse in a bottom-up way (e.g., a subroutine library).
  - A *framework* is a set of cooperating classes that make up a reusable design (i.e., not just code) for a specific class of software (e.g., for GUIs). Here, you write code that calls other people's code. This is sort of an inverted toolkit (e.g., Java's applet framework). This is the most difficult category of software to design.
- How do design patterns differ from frameworks? Patterns are more abstract, smaller, and domain independent.

Structure of the Catalog

- There are 23 patterns.
- There are 3 categories:
  - Creational patterns determine how objects are created.
  - Structural patterns determine how objects look.
  - Behavioral patterns determine how objects act.

A final warning

- Don't apply patterns without careful analysis.
- They can complicate a system and reduce performance.
- Use them when the added flexibility is worth the cost.

A case study: Lexi

- Lexi is a WYSIWYG document editor:
  - It is based on Doc, a 1992 object-oriented (OO) implementation.
  - It supports text and graphics.
  - It has various formatting styles.
  - It has pull-down menus, scroll bars, icons, etc.

Design problems

- We'll consider seven design problems, using eight design patterns to solve them.
- These are the design problems:
  - What is the *internal* representation of a document?
  - What is the *external* representation of a document, and how will "formatting" create it from the internal one?
  - How can widgets be added?
  - How can we support multiple "look and feels?"
  - How can we support different windowing systems?
  - How can different widgets invoke the same operations? How can we support an "undo" operation?
  - How can we add analytical operations, like spellchecking and hyphenation, without modifying many classes?

Document structure
(internal representation)
(1 of 4)

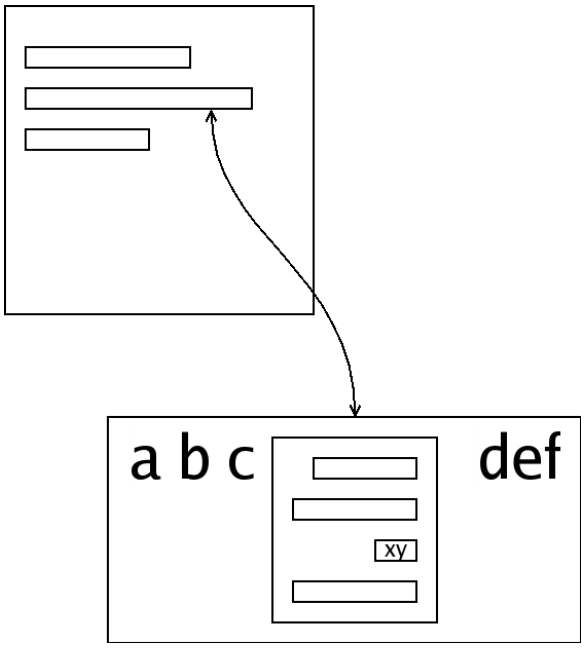
- A document is an arrangement of graphical elements, called *glyphs*. For example:
  - characters
  - rectangles (axis aligned)
  - polygons
  - rows
  - columns
- Should we expect this list to grow, over time?
- Thus, a document is a structure of substructures, to an arbitrary depth. We'll need to maintain this structure, and produce external (i.e., visual) representations.

Document structure (internal representation) (2 of 4)

- Our columns fill top-to-bottom, our rows fill left-to-right, but other structures are possible.
- A left-justified column might contain three rows.
- One of the top-justified rows might contain six characters and a right-justified column of four rows.
- Its third row might contain two characters.

Document structure (internal representation) (3 of 4)

- Graphically:



- The boxes in the picture are to show you where the rows and columns are: they are *not* rectangles.

Document structure (internal representation) (4 of 4)

- We'll need to map display position, pointed at with the mouse, to internal representation elements.
- We want to treat text, graphics, and structures uniformly.
- However, for example, hyphenating a rectangle does not make sense.

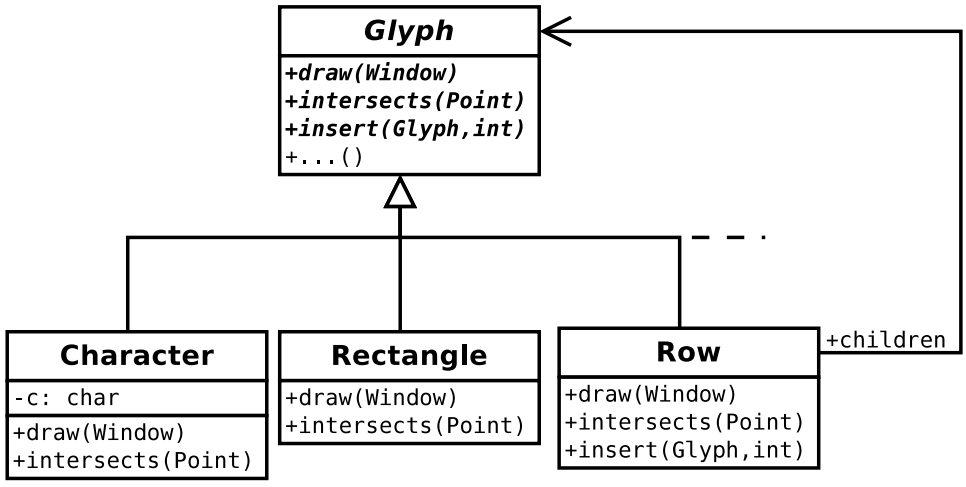
Recursive composition
(1 of 2)

- We'll use an object for each character and graphical element.
- We'll use an object for each substructure (e.g., row).
- Collectively, the objects will be of type Glyph. A Glyph has three responsibilities:
  - It can draw itself.
  - It knows what visual space it occupies.
  - It knows its children and parent, with regard to document structure, not inheritance.

Recursive composition (2 of 2)

- Here's a rudimentary Glyph interface (Table 2.1, page 39):

  pub/ch2/Glyph.java
- You'll need to change this for the homework.
- Here's the Glyph hierarchy:



- We'll use the Composite(163) pattern for this.

Composite(163) (1 of 3)

- Applicability:
  - You want to model part/whole hierarchies.
  - You want elements to be able to treat leaf and composite objects uniformly.
- Structure:

  pub/patterns/Composite.pdf

Composite(163) (2 of 3)

- Participants:
  - Component (Glyph)
    - It is the interface for Leaf and Composite.
    - It implements default behavior.
  - Leaf (Character)
    - It models atomic components.
    - It defines behavior.
  - Composite (Row)
    - It defines behavior.
    - It stores child components.
    - It implements `add`, `remove`, and `getChild`.
  - Client (Lexi)
    - It references a Component, usually a Composite.

Composite(163) (3 of 3)

- Consequences:
  - The client sees a uniform interface.
  - It simplifies the client.
  - It simplifies adding components: both leaves and composites.
  - It can make your design overly general. Sometimes, you don't want so much uniformity. You can use run-time type checking, though (ugh: we won't).
- Implementation:
  - Parent references can be tricky.
  - Try to maximize the Component interface.
  - Child-management operations can be tricky. Where do they belong?
  - Where should children be stored?

Formatting (1 of 6)

- We now have an *internal* representation. We need a way to construct various *external* representations.
- These two concepts are distinct.
- Different rows and columns of a given document may need to be formatted according to different algorithms.
- Ideally:
  - We want to be able to add a new Glyph subclass without modifying existing formatting algorithms.
  - We want to be able to add a new formatting algorithm without modifying existing Glyph subclasses.
  - We don't want to subclass existing Glyph subclasses (e.g., Row) to implement each formatting algorithm. This would result in a multiplicative number of classes. We want an additive number of classes.

Formatting (2 of 6)

- Formatting is *not* drawing. Formatting sets the bounds of a glyph. Drawing simply displays the glyph, making it visible in a window, according to those bounds.
- We'll define a separate class hierarchy for formatting:
  - The root will define an interface for formatting algorithms.
  - Each subclass will implement the interface for a particular algorithm.
- The interface is named Compositor:

  `pub/ch2/Compositor.java`

Formatting (3 of 6)

- A Composition is a subclass of Glyph, objects of which `compose` can format (e.g., Row).
- Objects of subclasses of Compositor format Composition objects differently (e.g., left justify or center).
- Our textbook describes a SimpleCompositor that formats a Composition structure by adding Row and Column objects. This approach makes sense for line breaking, but sounds too difficult for us. For example, a subsequent formatting pass might need to remove Row and Column objects.

Formatting (4 of 6)

- For simplicity, we'll assume that a Composition structure already has Row and Column objects. Formatting the structure simply requires setting the bounds of each Glyph object.
- At minumum, the visible glyphs should not overlap.
- Thus, our idea of row and column is more related to the logical structure of the document.

Formatting (5 of 6)

- When does a Glyph need to be formatted?
- Suppose the `insert` or `remove` method is called on a Glyph object. How much of the document must be formatted?
- Therefore, formatting must begin by traversing from that object to the "root" Glyph.
- Then, starting at the root, the Compositor's formatting algorithm recursively formats:
  - the current, parent, Glyph
  - its children Glyphs

  by calling methods on them.

Formatting (6 of 6)

- Different kinds of Glyph implement those methods in different ways.
- I used multiple methods, per Glyph, and called them in an interleaved (i.e., parent/child/parent $\cdots$) way.
- My SimpleCompositor keeps track of its "current positions" in Bounds objects I call cursors.
- Pseudocode:

  `pub/ch2/SimpleCompositor`
- The Composition-Compositor split is an example of the Strategy(315) pattern.

Strategy(315) (1 of 4)

- Aliases: It is also known as Policy.
- Applicability:
  - You have many related classes that differ only in their behavior.
  - You need different variants of an algorithm.
  - Your algorithm uses data that a client should not know about.
- Structure:

  pub/patterns/Strategy.pdf

Strategy(315) (2 of 4)

- Participants:
  - Strategy (Compositor)
    - It is the common interface for the algorithms.
  - ConcreteStrategy (SimpleCompositor)
    - These classes implement the various algorithms.
  - Context (Composition)
    - It references a particular object of type Strategy.
    - It may let a Strategy object access its data.

Strategy(315) (3 of 4)

- Consequences:
  - The pattern defines a family of algorithms.
  - This is an alternative to subclassing, but it allows you to change algorithms at run time.
  - It eliminates conditional statements.
  - It allows multiple implementations of the same behavior (e.g., for time/space tradeoffs).
  - Some strategies may not use the whole Strategy interface.
  - It adds objects to a system.

Strategy(315) (4 of 4)

- Implementation:
  - Algorithms need access to context data:
    - You can pass parameters to strategy methods.
    - You can pass a `this` reference.
  - A Context object may implement a default algorithm, only using the Strategy object if the reference is defined.

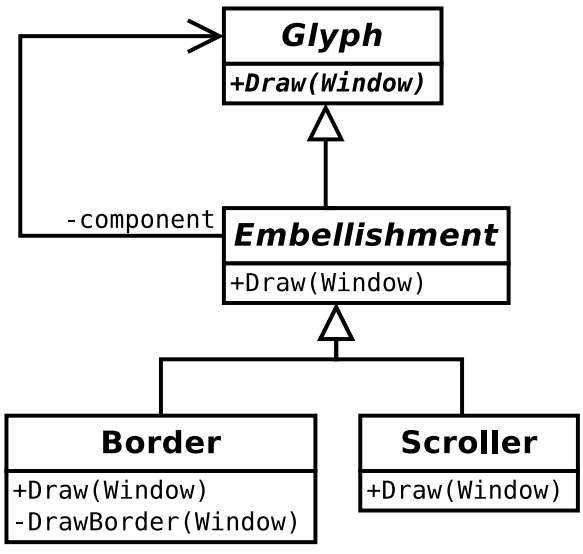
Embellishing the user interface (1 of 3)

- We want to add borders and scroll bars, at run time.
- If we used inheritance, we could subclass Composition to get:

  BorderedComposition
  ScrollableComposition
  ⋮
  BorderedScrollableComposition
  ScrollableBorderedComposition
  ⋮
- Instead, we'll use object composition. In particular, we'll use the concept of *transparent enclosure*:
  - single-component composition
  - compatible interface

Embellishing the user interface (2 of 3)

- For example, a Border object will contain a reference to a Composition object.
- When you ask a Border to draw itself, it asks the Composition to draw itself.
- We'll define a Glyph subclass, named Embellishment, to serve as an abstract class for all embellishment Glyphs (e.g., Border)



Embellishing the user interface (3 of 3)

- A C++ example:
  `pub/ch2/Embellishment.cc`
- Embellishment is an example of the Decorator(175) pattern.

Decorator(175) (1 of 3)

- Intent:
  - This pattern attaches additional capabilities/responsibilities to an object, dynamically.
- Aliases: It is also known as Wrapper.
- Motivation:
  - For Lexi, add some combination of borders and scroll bars, in any order.
- Applicability:
  - You want to add responsibilities to an object dynamically and transparently.
  - You want to add responsibilities that can be withdrawn.
  - You want to avoid extension by inheritance, because it is impractical.
- Structure:

  pub/patterns/Decorator.pdf

Decorator(175) (2 of 3)

- Participants:
  - Component (Glyph)
    - It is the interface for Decorator objects and the objects they decorate.
  - ConcreteComponent (Row)
    - It is the class of objects to be decorated.
  - Decorator (Embellishment)
    - It is the superclass for ConcreteDecorator classes.
    - It has a reference to a decorated object.
    - It forwards requests to its decorated object.
  - ConcreteDecorator (Border)
    - It is a subclass of Decorator.
    - It adds responsibilities/capabilities.

Decorator(175) (3 of 3)

- Consequences:
  - It is more flexible than inheritance.
  - It can add a property twice.
  - It avoids feature-laden superclasses.
  - A decorated object is a different object, which can complicate object identity.
  - It produces many objects.
- Implementation:
  - A Decorator object must conform to a Component object's interface.
  - Decorator(175) changes the "skin" of an object. Strategy(315) changes the "guts" of an object.

Supporting multiple look-and-feel standards (1 of 6)

- Look-and-Feel (L&F) standards seek to enforce uniformity across applications.
- For example, Java's Swing framework supports hundreds of L&F's (e.g., Java, Macintosh, and Windows). You can change one statement in your program to change the entire program's L&F.
- How does Swing do that? We would like to be able to switch Lexi's L&F easily, perhaps at run time.

Supporting multiple look-and-feel standards (2 of 6)

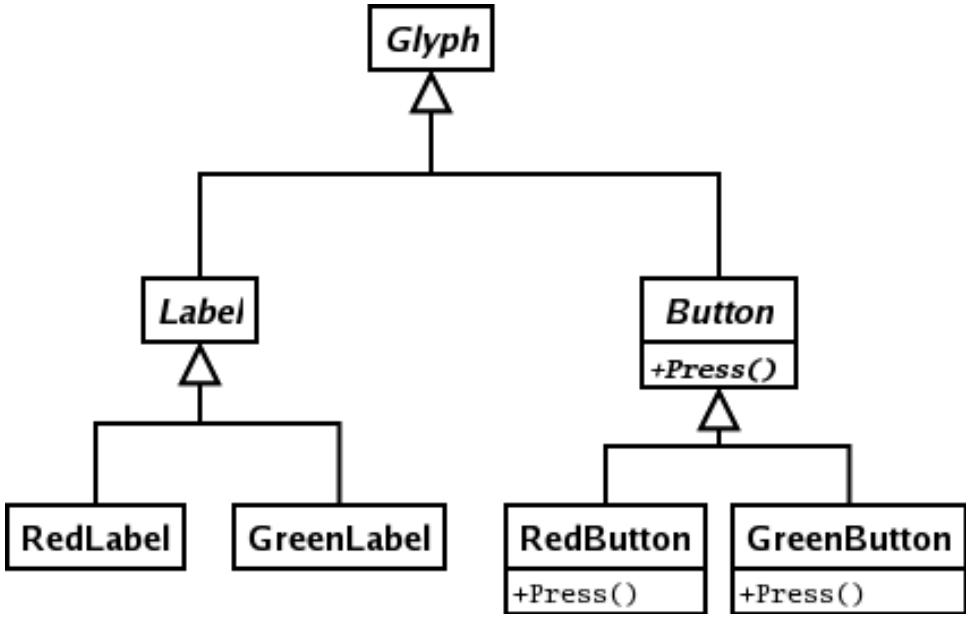
- Suppose we want to have two (but imagine many) new kinds of Glyph:
  - Button
  - Label
- Suppose we want to have two (but imagine many) L&F's:
  - red: buttons and labels are red
  - green: buttons and labels are green
- Real L&F's have many characteristics (e.g., language, shape, and decorations).

Supporting multiple look-and-feel standards (3 of 6)

- Changing the L&F during execution does not "convert" Button and Label objects that have already been created, but newly created Button and Label objects will have the new L&F.
- So, you probably only want to set the L&F when the program starts (e.g., according to the value of an environment variable), not after it has been executing awhile.

Supporting multiple look-and-feel standards (4 of 6)

- We'll use two sets of classes:
  - A set of abstract Glyph subclasses provides interfaces.
  - A set of concrete classes for each abstract class provides implementation.



Supporting multiple look-and-feel standards (5 of 6)

- We do not want code like:

```
RedButton button=new RedButton();
RedLabel label=new RedLabel();
```

  or even:

```
Button button=new RedButton();
Label label=new RedLabel();
```

  scattered throughout Lexi.
- Instead, we'll use:

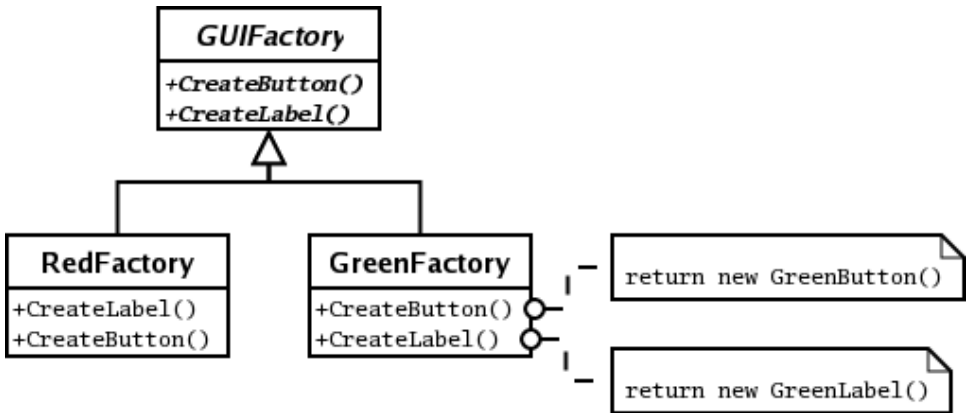
```
Button button=guiFactory.createButton();
Label label=guiFactory.createLabel();
```

- At exactly one place in Lexi, we'll have a statement like:

```
GUIFactory guiFactory=new RedFactory();
```

Supporting multiple look-and-feel standards (6 of 6)

- Here's the GUIFactory hierarchy:



- We are using the AbstractFactory(87) pattern.
- The method `createButton` is an example of the FactoryMethod(107) pattern.
- We usually only need one AbstractFactory object. The Singleton(127) pattern enforces this constraint.

AbstractFactory(87) (1 of 5)

- Intent:
  - This pattern provides an interface for creating sets of related objects.
  - Such an object is called a *product*. A factory creates products
  - A client need not specify concrete classes to create objects of those classes.
- Aliases: It is also known as Kit.
- Motivation:
  - For Lexi, we want multiple L&F's.
  - Each L&F provides a set of widget classes (e.g., Buttons).
  - We want to change the L&F without changing the client.
  - We want to prohibit mixing L&F's.

AbstractFactory(87) (2 of 5)

- Applicability:
  - You want to decouple object creation from clients.
  - You want to simplify client configurability.
  - You want to prohibit mixing of configurations.
  - You want to hide implementations.
- Structure:

  pub/patterns/AbstractFactory.pdf

AbstractFactory(87) (3 of 5)

- Participants:
  - AbstractFactory (GUIFactory)
    - It is the interface for factories.
  - ConcreteFactory (RedFactory, GreenFactory)
    - These are the implementations of AbstractFactory.
    - They have methods to create product objects (e.g., Buttons).
    - A dashed arrow models the "creates" relationship.
  - AbstractProduct (Button, Label)
    - These are the interfaces for product objects.

AbstractFactory(87) (4 of 5)

- Participants (continued):
  - ConcreteProduct (RedButton, RedLabel)
    - These are the implementations of AbstractProduct.
    - Their methods implement a product (e.g., RedButton).
  - Client (Lexi)
    - It uses AbstractFactory and AbstractProduct interfaces.

AbstractFactory(87) (5 of 5)

- Consequences:
  - It isolates concrete classes.
  - It simplifies changing sets of product objects: just create a different factory.
  - It promotes consistency between sets of product objects.
  - Adding new product classes is difficult (e.g., a Slider).
- Implementation:
  - Each ConcreteFactory (e.g., RedFactory) is often a Singleton(127).
  - Each product-creation method (e.g., `createButton`) is often a FactoryMethod(107).

FactoryMethod(107) (1 of 7)

- Intent:
  - This pattern provides an interface for creating objects.
  - It lets subclasses decide which class to instantiate.
- Aliases: It is also known as VirtualConstructor.
- Motivation:
  - For Lexi, we want to construct one kind of widget (e.g., Buttons).
  - Subclasses decide the L&F (e.g., red or green).
- Applicability:
  - A class can't anticipate what class of objects to create.
  - A class wants subclasses to decide.
  - You want to centralize which delegate is being used.

FactoryMethod(107) (2 of 7)

- Structure:
  pub/patterns/FactoryMethod.pdf
- Participants:
  - Product (Label)
    * It provides the interface of objects created by the factory.
  - ConcreteProduct (RedLabel)
    * It is the implementation of Product.
  - Creator (GUIFactory)
    * It declares the factory method:

```
protected abstract Label
  labelFactoryMethod();
```

    * It defines a `create` method that may do more than just return the result of the factory method:

```
public final Label createLabel() {
  return labelFactoryMethod();
}
```

FactoryMethod(107) (3 of 7)

- Creator (GUIFactory) (continued)
  - You might be tempted to declare the factory method `private`, but Java won't let you.
- ConcreteCreator (RedFactory)
  - A dashed arrow models the "creates" relationship.
  - It defines the factory method:

```
protected Label
    labelFactoryMethod() {
  return new RedLabel();
}
```

FactoryMethod(107) (4 of 7)

- Consequences:
  - Client code does not refer to concrete client classes (e.g., RedLabel).
  - It is more flexible than creating objects directly. Since creation is centralized, an "extended" version of an object can be provided instead.
  - It can connect parallel class hierarchies. For example, suppose we had a Button/Label/$\cdots$ manipulation class hierarchy. GUIFactory could provide both:

```
protected abstract Label
  labelFactoryMethod();
protected abstract LabelManipulator
  labelManipulatorFactoryMethod();
```

FactoryMethod(107) (5 of 7)

- Implementation:
  - The factory method can be abstract, as we've seen. It can also be concrete, providing a default implementation.
  - The factory method can be parameterized, rather than overridden by subclasses:

```
protected Label
    labelFactoryMethod(String kind) {
  if (kind.equals("red"))
    return new RedLabel();
  ⋮
  return null;
}
```

FactoryMethod(107) (6 of 7)

- Implementation (continued):
  - Normally, we don't like this sort of structure, but the class containing this method can be subclassed to *add* more kinds:

```
protected Label
    labelFactoryMethod(String kind) {
  if (kind.equals("mauve"))
    return new MauveLabel();
  ⋮
  return
    super.labelFactoryMethod(kind);
}
```

FactoryMethod(107) (7 of 7)

- Implementation (continued):
  - Generics (templates) can avoid subclassing Creator:
    pub/ch2/GUIFactory.cc
  - Unfortunately, this doesn't work in Java, because you can't reference a type-parameter's constructor like that:
    pub/ch2/TryGUIFactory.java
  - However, you can avoid subclassing Creator like this, which is similar to Prototype(117):
    pub/ch2/TryFactory.java
    A product is now, also, a factory.
  - Or even, with reflection:
    pub/ch2/TryFactory2.java

Singleton(127) (1 of 6)

- Intent:
  - Enforce that a class has only one instance.
  - Provide global access to that instance.
- Motivation:
  - For Lexi, we want only one GUIFactory object to be created.
- Applicability:
  - You want only one instance.
  - The singleton class should be extensible by subclassing.
- Structure:

  pub/patterns/Singleton.pdf

Singleton(127) (2 of 6)

- Participants:
  - Singleton (GUIFactory)
    - It defines the method `instance`, to return a reference to the object, creating it if needed.
    - The method should be `public` and `static`.
    - The constructor *must* be `private`.
- Consequences:
  - Access to the singleton instance can be controlled.
  - It avoids global variables.
  - The singleton class can be subclassed.
  - It permits a variable number of instances, if that's what you want.

Singleton(127) (3 of 6)

- Consequences (continued):
  - It is more flexible than static operations. For example:

```
Singleton s=Singleton.instance();
s.foo();
s.bar();
```

versus:

```
Singleton.foo();
Singleton.bar();
```

because in C++ and Java, you cannot declare a static method to be virtual/abstract:

pub/ch2/OverrideStatic.cc

pub/ch2/OverrideStatic.java

Singleton(127) (4 of 6)

- Implementation:
  - Hide the constructor call behind a static method.
  - The singleton class can be subclassed, but making it a factory and enforcing visibility, at least in Java, can be tricky:
    * The superclass (e.g., WidgetFactory) should be abstract, contain a variable to hold the instance, and define its `instance` method. This method should select and create the appropriate instance object from one of its subclasses.

Singleton(127) (5 of 6)

- The singleton class $\cdots$ (continued)
  - The concrete subclasses (e.g., GreenFactory) must be singletons, too. They should each define their own `instance` method. Ensure that a devious user cannot obtain inconsistent objects by calling the `instance` methods of multiple subclasses.
  - The product constructors (e.g., GreenButton) must be hidden, too, enforcing consistency. They can be hidden in packages or nested `private` classes.
- You can create a "registry" of different singletons, which maps strings to singletons.

Singleton(127) (6 of 6)

- There are many ways to use this pattern. Some are simplifications of the general pattern.
- For example, I wrote this C++ code, a while ago:

  pub/ch2/log.h

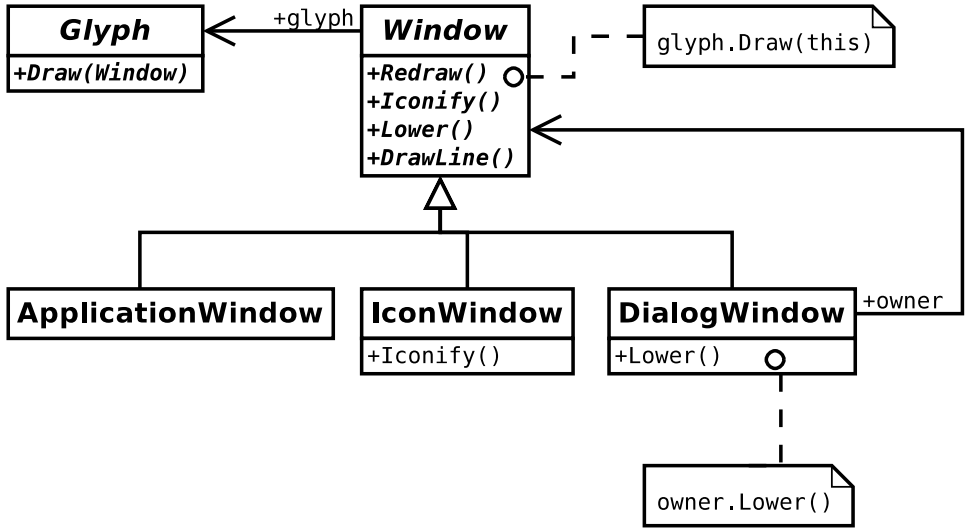
  pub/ch2/log.cc

Supporting multiple window systems (1 of 4)

- Our textbook uses the example of supporting MIT's X11 and IBM's Presentation Manager (PM).
- My example uses Sun's JDK's Swing and Abstract Windowing Toolkit (AWT).
- Why can't we just use AbstractFactory(87), like we did for multiple L&F's?
  - Each window system has multiple L&F's.
  - Two L&F's of a given product (e.g., Button) had the same interface. For example, RedButton and GreenButton had the same interface. We can't expect Buttons from two window systems to have the same interface.
- However, we can create a common interface for our window systems, and let clients program to it.

Supporting multiple window systems (2 of 4)

- Suppose we have more than one kind of window.



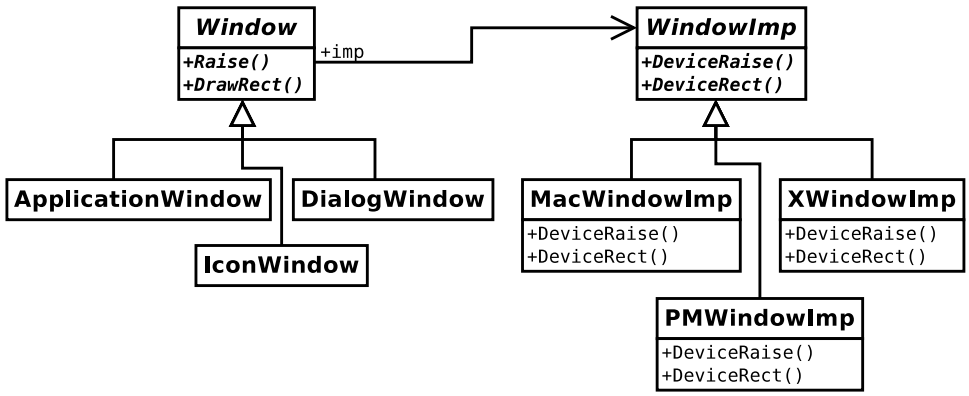
- Lexi is an application window.
- We could subclass to get:

SwingApplicationWindow	AwtApplicationWindow
SwingIconWindow	AwtIconWindow
SwingDialogWindow	AwtDialogWindow

but, we've seen that kind of explosion before, and we don't like it.

Supporting multiple window systems (3 of 4)

- The general technique we'll use is:
  Encapsulate the concept that varies.
- We want to isolate window-system-specific code in a class, which is a subclass of WindowImp:



- This arrangement has several benefits:
  - It isolates each window-system implementation.
  - It is easy to add new window systems.
  - Window subclasses can be further subclassed, if necessary.

Supporting multiple window systems (4 of 4)

- Benefits (continued):
  - Clients program to the interface of Window, the abstract windowing class, whose interface may differ significantly from that of WindowImp.
- The Window/WindowImp split is an example of the Bridge(151) pattern.
- WindowImp can be initialized according to the techniques used earlier: AbstractFactory(87), FactoryMethod(107), and Singleton(127).

Bridge(151) (1 of 5)

- Aliases: It is also known as Handle/Body, where Handle is the abstract side and Body is the concrete side.
- Motivation:
  - For Lexi, we want to support multiple window systems (e.g., Swing and AWT).
  - The Window abstraction is in one class hierarchy, and implementation classes are in another.

Bridge(151) (2 of 5)

- Applicability:
  - You want to decouple an abstraction from its implementations.
  - You want both the abstraction and its implementations to be extensible by subclassing.
  - You want to prevent an implementation change from affecting clients (e.g., by forcing recompilation).
  - You want to avoid an "explosion" of classes: a class for each combination.
- Structure:

  pub/patterns/Bridge.pdf

Bridge(151) (3 of 5)

- Participants:
  - Abstraction (Window)
    - It provides an abstract interface for clients.
    - It holds a reference to an object of type Implementor.
  - RefinedAbstraction (ApplicationWindow)
    - It is the actual interface for clients.
    - It "abstractly" implements Abstraction.
  - Implementor (WindowImp)
    - It is the interface for objects of class ConcreteImplementor.
    - It may be similar to, or quite different from, the interface of Abstraction.

Bridge(151) (4 of 5)

- Participants (continued):
  - ConcreteImplementor (SwingWindow, AwtWindow)
    - It implements the interface of Implementor.
- Collaborations:
  - An object of type Abstraction forwards client requests to an object of class ConcreteImplementor.
- Consequences:
  - It decouples Abstraction and Implementor.
  - It encourages layering.
  - It improves extensibility, since hierarchies can be extended independently.
  - It hides implementation details from clients.

Bridge(151) (5 of 5)

- Implementation:
  - Sometimes, you only need one Implementor.
  - AbstractFactory(87) or FactoryMethod(107) can be used to create a ConcreteImplementor.

User operations (1 of 6)

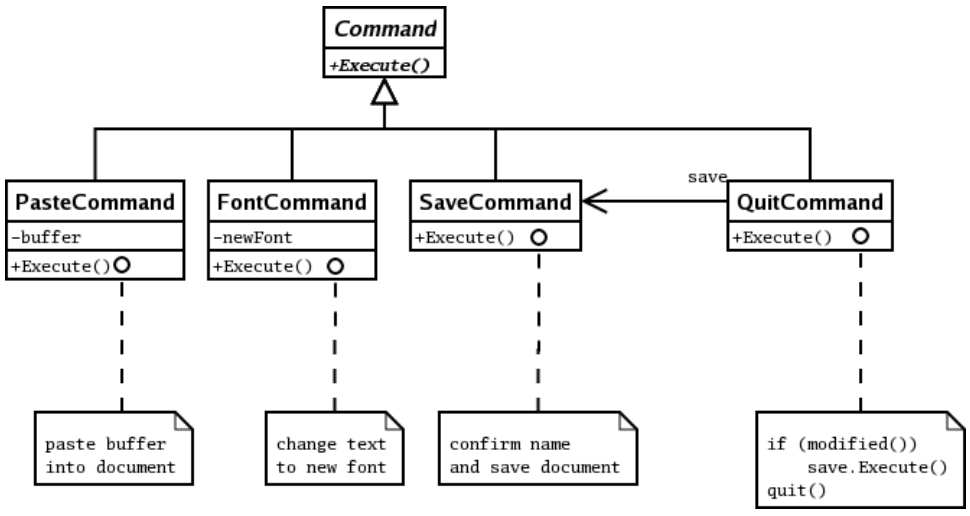
- A user controls execution of a program by performing "commands" (e.g., keystrokes or clicks). Sometimes, a keystroke performs the same command as a click.
- We also want to provide undo and redo commands. Some commands are not undoable (e.g., save, exit, and undo(?)).
- A *keymap* is a data structure specifying what command each keystroke should perform. An unmapped keystroke is ignored.
- We can augment our Glyph interface with a `click` method, which returns the command to perform when that Glyph object is clicked. We'll use Java's `null` to indicate that no command should be performed.

User operations (2 of 6)

- Do we want a separate Glyph subclass for each command that can be performed? No!
- We want to configure each clickable Glyph object with a particular Command object.
- A clickable Glyph's `click` method can simply return its Command object, for subsequent execution.
- Our keymap can map String objects (e.g., `"i"`) to Command objects (e.g., `new IncrFontSizeCommand()`).
- When our window abstraction is notified of a keystroke:
  - It consults its keymap.
  - It executes the command.
  - It adjusts a command-history structure.

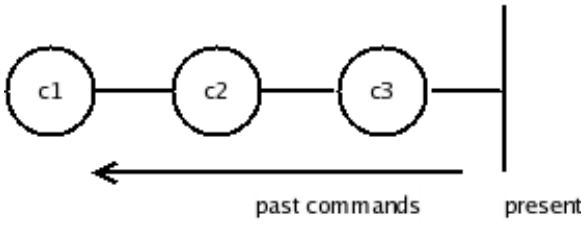
User operations (3 of 6)

- When our window abstraction is notified of a click:
  - It asks its Glyph object to find the child Glyph object occupying the click's x, y position.
  - It invokes that child's `click` method to get the command.
  - It executes the command.
  - It adjusts a command-history structure.
- Here is the Command part of the class hierarchy:

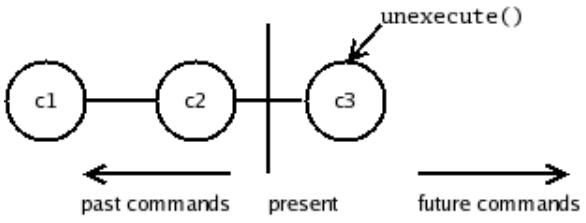


User operations (4 of 6)

- For undo and redo commands:
  - A Command object needs a method indicating whether it is undoable.
  - If it is, it needs enough state to undo itself.
  - It also needs a method to unexecute itself.
- An application needs a centralized data structure containing a subsequence of recently executed commands:

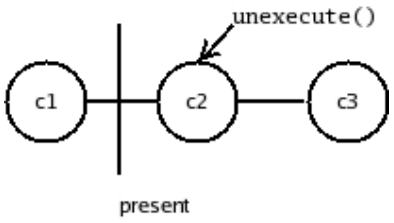


- To undo:

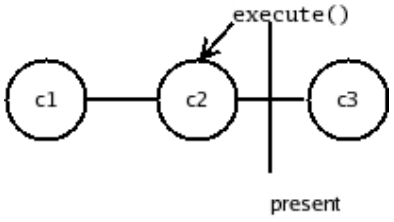


User operations (5 of 6)

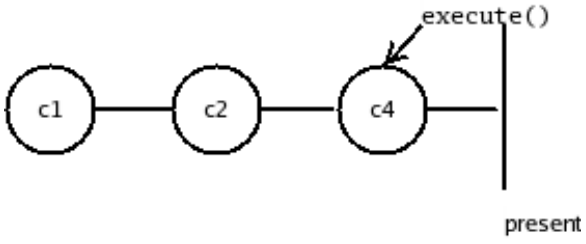
- To undo again:



- To redo:



- After a new command:



User operations (6 of 6)

- Lexi's command class is an example of the Command(233) pattern.
- A command needs to make copies of itself, so a copy can be added to the command-history structure. We'll use the Prototype(117) pattern.
- We want only one command-history structure, with global access, so we need Singleton(127) again.
- A Glyph object's `find` method is an example of ChainOfResponsibility(223).

Command(233) (1 of 5)

- Intent:
  - It encapsulates a request as an object:
    - A request can be passed as an argument.
    - A request can be queued or logged.
    - Undo and redo can be supported.
- Aliases: It is also known as Action or Transaction.
- Motivation:
  - For Lexi, a particular command should be invocable as a click or a keystroke.
  - We want undo and redo commands.
  - We want "macro" commands: a command that performs a sequence of other commands.

Command(233) (2 of 5)

- Applicability:
  - You want to parameterize an object with an action to perform (e.g., a Button). This is like a "callback function," but more flexible.
  - You want to queue an action for later execution.
  - You want to support undo and redo, with history and logging.
- Structure:

  pub/patterns/Command.pdf

Command(233) (3 of 5)

- Participants:
  - Command (Command)
    - It is the interface for all commands.
  - ConcreteCommand (SetFontSizeCommand)
    - It implememements the Command interface.
    - It provides state for execution and unexecution (e.g., new and old font size).
    - It provides state for the Receiver (e.g., Window).
  - Client (Lexi)
    - It creates Command objects.
  - Invoker (Window)
    - It invokes the `execute` method of a Command object.

Command(233) (4 of 5)

- Participants (continued):
  - Receiver (Glyph, Window)
    - These are the objects needed to perform the `execute` method.
- Collaborations:
  - A Client creates a ConcreteCommand and sets its Receiver.
  - The Client passes a ConcreteCommand to Invoker, for storage.
  - The Invoker invokes the ConcreteCommand's `execute` method.
  - The ConcreteCommand invokes methods on Receiver, saving previous state as necessary for undo.

Command(233) (5 of 5)

- Implementation:
  - Undo and redo are interesting:
    * Before performing its actions, the Command must save the current state. Its `unexecute` method uses this state to invoke methods on the Receiver to undo the effect of `execute`. Redo just invokes `execute` again.
    * For multiple levels of undo and redo, maintain a command-history data structure, as a Singleton(127).
    * Most commands must be copied before being placed in the structure, since they contain state information. Use Prototype(117).

Prototype(117) (1 of 3)

- Intent:
  - It uses an object to create other objects, by copying a prototypical object.
- Motivation:
  - For Lexi, we want to ask a Command object to make a copy of itself, so the copy can be put in the command-history data structure.
- Applicability:
  - You want to instantiate an object of a class that is not known until run-time.
  - You don't want the inconvenience of a factory.
  - You want a pre-initialized object.
- Structure:

  pub/patterns/Prototype.pdf

Prototype(117) (2 of 3)

- Participants:
  - Prototype (Command)
    - It is the interface for all ConcretePrototype objects, with a `clone` method.
  - ConcretePrototype (SetFontSizeCommand)
    - It implememements the interface of Prototype.
  - Client (CommandHistory)
    - It asks a ConcretePrototype object to clone itself.

Prototype(117) (3 of 3)

- Consequences:
  - It is similar to AbstractFactory(87).
  - Products can be added and removed at run time, via dynamic loading, where you cannot refer to the constructor statically.
  - Cloning a prototype is like instantiating a class.
  - It uses fewer subclasses than FactoryMethod(107).
- Implementation:
  - You can maintain a "registry" of prototypes, keyed on some reasonable type of value (e.g., String).
  - The `clone` method can perform a deep or shallow copy, but beware of cycles.
  - You might also need methods to reinitialize a cloned object.

ChainOfResponsibility(223) (1 of 3)

- Intent:
  - It decouples the sender of a request from its receiver, by passing the request down a chain of potential receivers, until a receiver handles it.
- Motivation:
  - For Lexi, we want to ask a Glyph object if it overlaps a particular x, y location.
  - It can reply with "yes" or "no," or it can ask a child Glyph object to reply.
- Applicability:
  - You want more than one object to consider replying to a request.
  - You don't know, in advance, which object will reply.
- Structure:

pub/patterns/ChainOfResponsibility.pdf

ChainOfResponsibility(223) (2 of 3)

- Participants:
  - Handler (Glyph)
    - It declares an interface for handling requests (e.g., a `find` method).
    - It optionally implements the successor link. I did this in a subclass, named CompositeGlyph.
  - ConcreteHandler (Glyph, Button)
    - It handles requests for which it is responsible.
    - It can access its successor (e.g., children).
    - If it can handle the request, it does so. Otherwise, it forwards the request to a successor.
  - Client (Window)
    - It initiates the request to a ConcreteHandler on the chain.

ChainOfResponsibility(223) (3 of 3)

- Consequences:
  - It reduces coupling.
  - It adds flexibility.
  - It allows for no object to handle the request.
- Implementation:
  - The successor chain may already exist (e.g., Glyph children) or it may require new links.
  - If new links are required they can be in the Handler class, which can also provide a default handling method, which just forwards the request to the successor. A "responsible" object can override the method.
  - A request can be represented by: different methods, codes (e.g., integers or strings), or objects.

Spell checking and hyphenation (1 of 7)

- We want to perform various kinds of Glyph analysis. For example: spell checking, hyphenation, searching, word counting, numeric summing, and grammar checking.
- We don't want to change Glyph and its subclasses to add a new kind of analysis.
- We will decouple the gathering and analysis parts of the problem.
- Our data is scattered across a large data structure.
- Different analyses require different traversals (e.g., forward search versus reverse search).
- We'll probably need preorder, inorder, and postorder traversals.

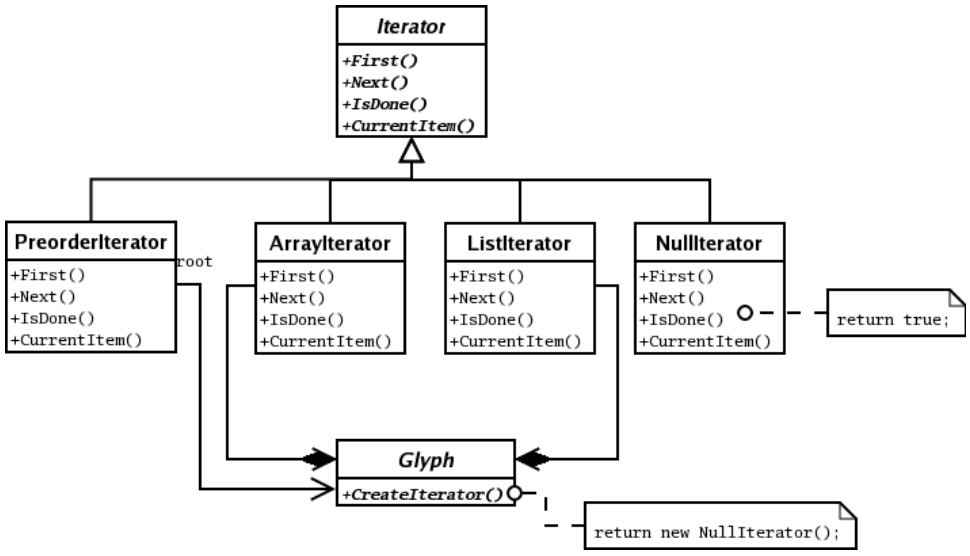
Spell checking and hyphenation (2 of 7)

- Currently, `Glyph.child` takes an integer index to select the child to return. This makes sense for a Vector, but less sense for a linked structure. Our Glyph interface should not be biased toward a particular representation. Indeed, different Glyph subclasses might use different data structures.
- We could change Glyph's interface to support these operations:

  pub/ch2/iter.java
- This helps, but:
  - We can't add traversals without changing Glyph classes.
  - We can't have concurrent traversals.
  - We can't traverse different object structures.

Spell checking and hyphenation (3 of 7)

- We will, again, encapsulate the concept that varies: access and traversal mechanisms. Our objects will be called *iterators*:



- Iterator subclasses hold a reference to the structure they traverse. Each also has state to identify the "current" item (e.g., an array index or link). The darker arrowhead denotes multiplicity.

Spell checking and hyphenation (4 of 7)

- By default, `createIterator` returns a new NullIterator object, which is always "done." Glyph subclasses override this method appropriately.
- We can use this interface as follows:

  pub/ch2/iterator.java
- Notice that this allows concurrent traversals.
- Here's an example of overriding `createIterator`:

  pub/ch2/CompositeGlyph.java
- Iterators for preorder, inorder, and postorder traversal are implemented in terms of particular iterators.

Spell checking and hyphenation (5 of 7)

- Consider `PreorderIterator`:
  - `first` needs to:
    - Create an iterator for the root (e.g., `VectorIterator.createIterator`).
    - Call `first` on it (e.g., `VectorIterator.first`). This is not recursion.
    - Unless it's done, push it onto a stack.
  - `currentItem` needs to:
    - Return the result of calling `currentItem` for the iterator at the top of the stack (e.g., `VectorIterator.currentItem`). This is not recursion.

Spell checking and hyphenation (6 of 7)

- `PreorderIterator` (continued):
  - `next` is tricky, and our textbook is wrong:
    * Create an iterator for the item from the iterator on top of the stack (e.g., `VectorIterator.createIterator`).
    * Call `first` on it (e.g., `VectorIterator.first`).
    * Push it onto the stack.
    * While the stack is not empty and the top iterator is done:
      · Pop the stack.
      · If the stack is not empty and the top iterator is not done: call `next` on the top iterator (e.g., `VectorIterator.next`). This is not recursion.

Spell checking and hyphenation (7 of 7)

- Some subclasses of Iterator (e.g., PreorderIterator) might contain references to Glyph objects:
  - If so, they are not reusable.
  - Our textbook says:

    There's no reason we can't parameterize a class like PreorderIterator by the type of object in the structure.
  - It's actually a bit tricky. The homework assignment has a link to a document that provides more information.
  - I want the graduate students to do this in their homework.
  - This will allow us to use preorder, inorder, and postorder traversals on other trees (e.g., parse trees).
- Of course, this is an example of the Iterator(257) pattern.

Iterator(257) (1 of 6)

- Aliases: It is also known as Cursor or Enumerator.
- Motivation:
  - For Lexi, we want to produce the children of a Glyph.
  - We want to hide the representation of a sequence of children.
  - We want different traversals (e.g., preorder, inorder, and postorder).
  - We want to allow multiple interleaved traversals (e.g., a search during a spellcheck)
  - We want reusability (i.e., we're lazy).

Iterator(257) (2 of 6)

- Applicability:
  - You want to access an aggregate's content without exposing representation.
  - You want concurrent traversals.
  - You want polymorphic iteration.
- Structure:

  pub/patterns/Iterator.pdf
- Participants:
  - Iterator (Iterator)
    - It is the interface for access and traversal.
  - ConcreteIterator (VectorIterator)
    - It implements Iterator for a particular aggregate.
    - It records the current position in an aggregate.

Iterator(257) (3 of 6)

- Participants (continued):
  - Aggregate (Glyph)
    - It is the interface for creating an Iterator object.
  - ConcreteAggregate (CompositeGlyph)
    - It implements the interface for creating an Iterator object.
- Consequences:
  - It supports different traversals of a single data structure.
  - It simplifies the Aggregate (e.g., Glyph) interface.
  - It supports concurrent traversal, because each ConcreteIterator maintains state.

Iterator(257) (4 of 6)

- Implementation:
  - There are external and internal iterators:
    * With an external iterator, the client performs an action on each element. For example:

```
for (i.init(); !i.done(); i.next())
  ···i.item()···
```

    * With an internal iterator, the client tells the iterator what to do with each element. For example:

      pub/ch2/iit.scm

      pub/ch2/iit.pl

    * External iterators are more flexible, but internal iterators are more convenient.

Iterator(257) (5 of 6)

- Implementation (continued):
  - The client can perform the traversal, using the iterator object to store state. This is called a *cursor*, a simple example of the Memento(283) pattern.
  - Robust iterators allow elements to be added or removed during traversal.
  - Of course, you can provide additional methods (e.g., `prev`) or fewer methods (e.g., merge `next` and `done`).
  - Polymorphic iterators can be produced by a factory method, but this technique has disadvantages. For Lexi, I used Java's generic interfaces.

Iterator(257) (6 of 6)

- Implementation (continued):
  - Iterators for structures like Composite(163) can be difficult to implement (e.g., preorder is tricky). You can maintain a collection (e.g., a stack) of node iterators.
  - A leaf node's `createIterator` method can return an instance of NullIterator, which is always "done."

Traversal versus traversal actions (1 of 2)

- The Iterator(257) pattern allows us to traverse a Glyph structure.
- Now, we need to gather Glyph data for analysis.
- We could put our analysis logic in our iterators, but that prevents us from reusing an iterator for different kinds of analysis.
- We could put our analysis logic in our Glyph classes. Different analyses involve different Glyph subclasses, which we could accomplish with polymorphism (i.e., overriding "default" methods in subclasses). However, new analyses would require changes to our Glyph classes.

Traversal versus traversal actions (2 of 2)

- As usual, we solve the design problem by encapsulating the concept that varies: the analysis algorithm.
- Each Glyph-analysis algorithm is implemented by a separate class.
- All Glyph-analysis classes implement an interface specifying Glyph-analysis methods: one method for each kind of Glyph (e.g., Character and Row). Overloading allows the method names to be the same.
- Now, we add a single method to the Glyph class hierarchy, allowing a Glyph to "accept" an analysis object.
- A Glyph object "accepts" an analysis object by invoking an analysis method on itself:

  pub/ch2/Character.java
  pub/ch2/Words.java

Visitor and subclasses

- Our Glyph-analysis classes are called visitors, subclasses of GlyphVisitor.
- A Glyph object "accepts" visitors, via its `accept` method.
- An object of a subclass of GlyphVisitor defines methods for visiting Glyph objects. It can distinguish between Glyph subclasses, via polymorphism. It can accumulate Glyph data in instance variables.

  `pub/ch2/GlyphVisitor.java`
- Of course, this is the Visitor(331) pattern.

Visitor(331) (1 of 7)

- Intent:
  - It encapsulates an operation or algorithm to be performed on an object structure (e.g., from Composite(163) or Interpreter(243)).
  - It allows a new operation or algorithm to be added without changing the class of objects being operated upon.
- Motivation:
  - For Lexi, we want to spellcheck a Glyph structure.

Visitor(331) (2 of 7)

- Applicability:
  - You want to perform an operation on each element of a structure of differing concrete subclasses.
  - You want related operations together, in one class, not spread across the structure's subclasses.
  - You want to separate operations from the object they operate on.
  - The classes defining the objects operated upon change rarely, but new operations are often required.
- Structure:

  pub/patterns/Visitor.pdf

Visitor(331) (3 of 7)

- Participants:
  - Visitor (GlyphVisitor)
    - It declares an overloaded `visit` method, for each concrete subclass of Element (e.g., Character).
  - ConcreteVisitor (WordsVisitor)
    - It implements each method declared by Visitor. Many might do the same thing or nothing. Collectively, the methods implement the algorithm.
    - An object of this class accumulates state during a traversal.
  - Element (Glyph)
    - It declares an `accept` method that takes a Visitor object as an argument.
  - ConcreteElement (Character)
    - It implements or overrides `accept` from its superclass.

Visitor(331) (4 of 7)

- Participants (continued):
  - ObjectStructure (Glyph)
    - It is the root of the structure.
    - It can enumerate its elements.
    - It is often a Composite(163).
- Consequences:
  - Adding new operations is easy.
  - Related methods are consolidated, away from the class of the object operated upon.
  - Adding a new ConcreteElement class is hard: every Visitor class must change.
  - While an iterator can only traverse objects with a common superclass, a visitor has no such restriction.
  - A visitor can accumulate state.

Visitor(331) (5 of 7)

- Consequences (continued):
  - This pattern breaks encapsulation. If a Glyph-analysis algorithm is inside a Glyph class, it can access private variables. With this pattern, the Glyph-subclass interface must provide public methods to allow the visitor to perform its job (e.g., Character must provide a `getchar` method).

Visitor(331) (6 of 7)

- Implementation:
  - We are effectively adding methods to subclasses of Element (e.g., Character) without changing the subclasses, using a technique called *double dispatch*. Usually, with *single dispatch*, the method to call is determined by the method signature and the object upon which it is invoked. With double dispatch, the particular Visitor object also helps determine which method is called.

Visitor(331) (7 of 7)

- Implementation (continued):
  - The traversal code can go in one of three places:
    * The Element (e.g., Row): A parent invokes `accept` on each of its children. This is the simple, but least flexible way.
    * An Iterator (e.g., PreorderIterator): This is much more flexible.
    * A Visitor: This is only necessary for very complex traversals. It causes each ConcreteVisitor to contain duplicate traversal code.

Chapter 2, revisited (1 of 5)

We wanted patterns to address the "Common Causes of Redesign." Let's see if they do:

- You create an object by explicitly specifying its implementation.
  We avoided sprinkling:

```
···=new RedLabel();
```

  throughout Lexi by using AbstractFactory(87), FactoryMethod(107), and Singleton(127):

```
WidgetFactory widgetFactory=
    WidgetFactory.instance(···);
⋮
···=widgetFactory.createLabel(···);
```

Chapter 2, revisited (2 of 5)

- You call a method by explicitly specifying its name.
  We avoided having Lexi's Window abstraction refer to particular click/keystroke operations by using Command(233) and Prototype(117):

```
public void key(char c) {
    execute(_keymap.get(c));
}
```

- You depend on a hardware or software platform.
  We avoided sprinkling references to SwingWindow methods throughout Lexi by using Bridge(151):

```
window.drawCharacter(_c,···);
```

Chapter 2, revisited (3 of 5)

- You depend on an object's implementation.
  We prevented Lexi from knowing about SwingWindow's implementation by using Bridge(151) and AbstractFactory(87):

```
Window w=new ApplicationWindow(···);
```

- You depend on a particular algorithm.
  We prevented parts of Lexi from knowing which algorithms we used by using Iterator(257), Strategy(315), and Visitor(331):

```
public abstract class Composition
     extends CompositeGlyph {
⋮
    _compositor.compose();
```

Chapter 2, revisited (4 of 5)

- You allow tight coupling.
  We designed loosely coupled classes by using AbstractFactory(87), Bridge(151), Command(233), and ChainOfResponsibility(223). See above.
- You customize by subclassing.
  We used object composition, rather than subclassing, in Bridge(151), Composite(163), Decorator(175), and Strategy(315):

```
public abstract class Composition
    extends CompositeGlyph {
⋮
    private Compositor _compositor;
```

Chapter 2, revisited (5 of 5)

- You need to change a class that you can't (e.g., you don't have its source code or you can't risk affecting current clients). We avoided changing Glyph subclasses by using Decorator(175) to add embellishments and Visitor(331) to add analysis algorithms.

Model/View/Controller (MVC) and Observer(293)

- The MVC "pattern" is very common, but not really a fundamental pattern.
- It's essentially a macro pattern, composed of at least Observer(293). It can also employ Singleton(127), Composite(163), and Strategy(315), as well as other patterns.
- It's mentioned in Section 1.2 of our textbook.
- First, we'll look at Observer(293), then see its use in MVC.

Observer(293) (1 of 7)

- Intent:
  - It defines a one-to-many dependency between objects.
  - When the one changes state, it notifies its dependents.
- Aliases: It is also known as Dependents or Publish-Subscribe.
- Motivation:
  - For a graphical application (e.g., Lexi), we want to coordinate the interactions of several dependent objects or subsystems:
    * the internal state (i.e., data representation)
    * one or more external (e.g., visual output) representations
    * one or more input mechanisms (e.g., keyboard and/or mouse)

Observer(293) (2 of 7)

- Motivation (continued):
  - For example, a keystroke changes the state and its external representation.
  - We want to avoid polling and tight coupling.
- Applicability:
  - An abstraction has two aspects, which you want to separate, but one object depends on the other.
  - A run-time change to one object requires changing other objects, but you don't know how many there are.
  - The objects should be loosely coupled.
- Structure:

  pub/patterns/Observer.pdf

Observer(293) (3 of 7)

- Participants:
  - Subject
    - It knows its one or more Observers.
    - It provides an interface for allowing Observers to be added or removed.
  - Observer
    - It declares an `update` method that the Subject can call to notify an Observer of changes.
  - ConcreteSubject (internal state)
    - It stores state.
    - It notifies Observers of a change of state.

Observer(293) (4 of 7)

- Participants (continued):
  - ConcreteObserver (visual representation)
    - It maintain a reference to its Subject.
    - It may store part of the Subject's state, but watch for consistency.
    - It implements the `update` method, to react to Subject state changes.
- Collaborations:
  - A ConcreteSubject notifies ConcreteObservers.
  - A ConcreteObserver queries the ConcreteSubject for state information.

Observer(293) (5 of 7)

- Consequences:
  - Subjects and Observers can be varied independently.
  - Their coupling is abstract and minimal.
  - Communication is broadcast. Observers can ignore it.
  - Many updates may be unnecessary.
  - Updates can get stuck in a "loop."
- Implementation:
  - The set of `observers` can be indexed, and shared between Subjects.
  - If there are multiple Subjects, `Observer.update()` can be passed a reference to the notifying Subject, for identification.

Observer(293) (6 of 7)

- Implementation (continued):
  - Who calls `Subject.notify()`? The Subject can, so Observers don't forget. An Observer can, which might be more efficient, because the Observer understands the change(s).
  - The Subject should ensure its state is self consistent, before notifying Observers.
  - Additional information can be passed to `Observer.update()`, so Observers have some idea of what changed.
  - The Subject can maintain multiple sets of Observers, for different sorts of changes.

Observer(293) (7 of 7)

- Implementation (continued):
  - If there are multiple Subjects, their changes might need to be atomic, or at least coordinated. A "change-manager" object can do this.
  - The Subject and Observer interfaces can be combined, perhaps avoiding the need for multiple inheritance.

MVC (1 of 3)

- In its simplest form, MVC is just Observer(293) with one Subject and two Observers:
  - Model is the Subject
  - View is an "output" Observer
  - Controller is an "input" Observer
- From Wikipedia:

  pub/etc/mvc.pdf
- I've written a miminal Java program to demonstrate MVC.
  - The Model is a string.
  - The View is a window showing the string.
  - The Controller is the keyboard, for changing the string.
- Here's the main class:

  pub/mvc/MVC.java

MVC (2 of 3)

- In pattern patois, passing and storing the subsystem references is called *dependency injection*, implementing *inversion of control*.
- It's a little weird because the Java API's windows and keyboards are coupled.
- Subject is like our textbook's:

  pub/mvc/Subject.java

  except:
  - There's an initial call to `Observer.update()`.
  - Method `notify()` is renamed.
  - The class is actually concrete.
- Observer is simple:

  pub/mvc/Observer.java
- The state is just a string and an insertion point:

  pub/mvc/State.java

MVC (3 of 3)

- Model uses Singleton(127):
  pub/mvc/Model.java
- The View is a Java API `JFrame`, containing a `JLabel`, containing the Model's string:
  pub/mvc/View.java
- A Java API `JLabel` doesn't respond to keystrokes, like a `JTextField`. However, my Controller class listens for `JFrame` keystrokes, and updates the Model. The Model then notifies the View to change the label's content.
- Notice that the Controller's change causes the controller to be notified, too:
  pub/mvc/Controller.java

Another application domain: scanning and parsing

- I want to demonstrate using OO design patterns for a Scanning/Parsing Framework.
- We'll see familiar patterns (e.g., Composite(163)) and new patterns (e.g., Adapter(139)).

Ported from Icon to Java

- Initially, I implemented the application in Icon:
  - Values have types, but variables are untyped.
  - It has no OO features.
  - It has generators, a good control abstraction.
  - It has containers (e.g., sets, lists, and tables).
- Here's an example of an Icon generator:
  `pub/etc/fib.icn`

What the framework can do (1 of 3)

- I've ported it to Java, using OO features, design patterns, and generics.
- Here's what an application using the framework can do:
  - It scans and parses a grammar-input file, producing a parse tree. This is a recursive-descent parser. The content of the file describes a grammar.
  - It walks the parse tree, producing a grammar.

What the framework can do (2 of 3)

- what it can do (continued)
  - It processes the grammar, producing:
    - symbols
    - nonterminal symbols
    - terminal symbols
    - first: Symbol $\rightarrow$ Terminals
    - follow: Symbol $\rightarrow$ Terminals
    - an LL parsing table: Nonterminal Terminal $\rightarrow$ Rule
    - an LR parsing table:
      - action: State Symbol $\rightarrow$ Action
      - goto: State Symbol $\rightarrow$ State
  - It uses the LL parsing table to parse an input file.
  - It uses the LR parsing table to parse an input file.

What the framework can do (3 of 3)

- I realize you might be rusty on, or have not seen, some of these concepts. I'll review them as needed.
- The behavior described above isn't the framework. It's an application that uses parts of the framework.
- I tried to guess which parts of the application are reusable, and thus part of the framework, but until other applications are developed, I won't know for sure.

Grammar-input file

- The grammar-input file to the application contains a textual representation of a grammar:

  pub/parse/TEST/grammar.grammar
- Notes:
  - The metasymbols are `:`, `|`, and `;`.
  - The symbols are everything else, separated by whitespace.
  - The nonterminals are symbols that appear to the left of a `:`. The terminals are the other symbols.
  - Rules are terminated by a `;`. There are twelve rules.
  - The whole thing is a grammar.
- It is a grammar specifying grammars.

The main program

- Now, let's look at the "main" program, to see how the behavior is implemented:
  `pub/parse/Main.java`
- Does `main` reveal that *any* patterns are being used? Yes, the `Args.instance` method does suggest the Singleton(127) pattern:
  - There is one instance of the command-line and default argument values.
  - The static method provides global access to these values.
- Before we look at Args, why didn't I follow the pattern more closely?

Command-line arguments (1 of 7)

- The Singleton participant of the Singleton(127) pattern is Args.
- Almost every program needs command-line arguments, so we want a flexible and reusable subsystem.
- Here are some requirements:
  - We've already seen: global access, default values, and command-line values.
  - Nickname arguments are convenient (e.g., `-g` and `--grammar`).
  - Some arguments have non-String values. For example, an argument's value might be an existing file, which should be opened and ready to read.

Command-line arguments (2 of 7)

- Requirements (continued):
  - A good "usage" message helps users. For example:

    pub/parse/usage
  - We should be able to add new arguments, without changing the reusable classes. In this application, we only need to change class Main.
  - We don't have to change, for example, Error:

    pub/parse/Error.java

Command-line arguments (3 of 7)

- The argument subsystem comprises these classes:

  `pub/parse/Args.java`

  `pub/parse/Arg.java`
- I used Command(233) for arguments:
  - The Command participant is Arg.
  - The ConcreteCommand participant is demonstrated by ArgFileRead, and others:

    `pub/parse/ArgFileRead.java`
  - I did not employ undo/redo behavior.

Command-line arguments (4 of 7)

- Arguments are stored in an ArgMap:
  - This is like Lexi's KeyMap. A key is a String (e.g., `--grammar` or `-g`). A data value is a reference to an Arg object.
  - Here's the code:

    pub/parse/ArgMap.java
  - Here's a sample call sequence:

```
Main.main()
  Args.add()
    Arg.set()              // for nicknames
      ArgMap.put()
        Map.put()          // Java's API
```

  - ArgMap provides an iterator, by exposing its representation's iterator (i.e., the Map's iterator). I've found this technique to be so handy that it's almost a Java-API-specific pattern.

Command-line arguments (5 of 7)

- ArgMap (continued):
  - Now, we can see how the usage message is generated:

```
Main.main()
  ArgUsage.init()
    Args.toString()
      Arg.toString()
```

The Map is transformed to a sorted set, which is transformed to a big String:

pub/parse/ArgUsage.java

Command-line arguments (6 of 7)

- The last argument-related feature I want to show is how an argument's value can be of a type other than String. (e.g., FileRead).
- When you request the value of an argument, you pass the name of the argument (e.g., `--grammar`) to `Args.get`, which looks-up the name in the ArgMap, and calls an overridden `Arg.get`. For example:

  pub/parse/ArgFileRead.java
- An `Arg.get` method can perform an arbitrary amount of work, but it returns an ArgWrap object:

  pub/parse/ArgWrap.java

Command-line arguments (7 of 7)

- An ArgWrap object is effectively a free union, implemented as a structure:
  - There is one field for each type of Arg (Java does not have unions, like C or C++).
  - Exactly one of these fields is "active."
  - The client must know which field is active, so it can call the right extraction method.
  - It can be subclassed.
- Is this technique a pattern?

  `pub/parse/Main.java`

The FileRead Abstraction (1 of 3)

- Java's file-oriented abstractions are quite complicated.
- Icon's files are simple. Here's a cat program, in Icon:

```
procedure main()
    while write(read())
end
```

- I used the Adapter(139) pattern to provide a simple interface to Java's API:

  pub/parse/FileRead.java

- The `init` method is for when you need to read through the file more than once.
- I suppose I could have used Iterator(257).

The FileRead Abstraction (2 of 3)

- Java provides (at least) two kinds of line-oriented file: BufferedReader and Scanner.
- We can use FactoryMethod(107), and we could use Singleton(127).
- The Creator participant is:

  `pub/parse/JavaFile.java`
- The two ConcreteCreator participants are:

  `pub/parse/JavaBufferedReader.java`

  `pub/parse/JavaScanner.java`
- We could have other FileRead implementations.
- We could also have several FileWrite implementations.

The FileRead Abstraction (3 of 3)

- Should we have used AbstractFactory(87) and/or Bridge(151)?
- AbstractFactory(87) is applicable when we have a family of products, rather than just one.
- Bridge(151) is more applicable when we want to allow interfaces to vary, rather than when we are are changing the interface to an existing class.

Adapter(139) (1 of 3)

- Intent:
  - It provides a client with a desired interface, to a class with a different interface.
- Aliases: It is also known as Wrapper.
- Motivation:
  - For the scanner/parser framework, we want a simple interface to Java's file API.
- This pattern has two forms:
  - In the object form, the Adapter has-a Adaptee.
  - In the class form, the Adapter is-a Adaptee.
  - In both cases, Adapter provides the interface wanted by the client.
  - We'll consider the object form.

Adapter(139) (2 of 3)

- Applicability:
  - You want to use a class, but its interface isn't what you need.
  - You want to create a reusable client.
  - You want to adapt several existing subclasses, so you adapt their superclass' interface.
- Structure:

  pub/patterns/Adapter.pdf
- Participants:
  - Target (FileRead)
    - It declares the domain-specific (e.g., lexical analysis) interface the client uses.
  - Client (Lexer)
    - It employs the Target interface.
  - Adaptee (BufferedReader)
    - It declares an existing (e.g., Java API) interface.

Adapter(139) (3 of 3)

- Participants (continued):
  - Adapter (JavaBufferedReader)
    - It adapts Adaptee to the Target interface, via object composition.
- Consequences:
  - The Adapter can work with many Adaptees (i.e., its subclasses).
  - The Adapter can add functionality.
  - The Adapter cannot (directly) override Adaptee behavior, like it can with the class form of this pattern.
  - An Adapter is *not* an Adaptee object. You can't treat it like an Adaptee.
- Implementation:
  - This is usually straightforward.

Grammar analysis (1 of 7)

- You should/might remember (some of) this from your programming-languages class.
- Examples of simple analysis are computing the set of terminal symbols and the set of nonterminal symbols of a grammar.
- As you might expect, a Symbol object has-a String member: the lexeme read by the lexer.
- A Symbols object has-a Set<Symbol> member.
- We'll see the code later, but we can extend Symbols and iterate over the Rule objects in a Grammar object:

  `pub/parse/AllSymbols.java`

  `pub/parse/NonterminalSymbols.java`

  `pub/parse/TerminalSymbols.java`

Grammar analysis (2 of 7)

- Examples of more complex grammar analysis are computing the FIRST and FOLLOW functions. Recall that both these functions take a symbol as an argument (i.e., a Symbol object) and return a set of symbols (i.e., a Symbols object).
- Since FIRST and FOLLOW are so similar, they should extend a common superclass:
  - What should that class be?
  - Can we make it reusable?
  - What is a good implementation?
- A function can be modeled as a map from argument types (i.e., its domain) to a return type (i.e., its range).
- For the superclass of FIRST and FOLLOW, we want to map from a single argument type (e.g., Symbol) to the type that is the powerset of the argument type (e.g., Symbols). For lack of a better name, I call the superclass SetMap.

Grammar analysis (3 of 7)

- Now, we can subclass SetMap:

```
public class First extends
    SetMap<Symbol,Symbols> {

    public First(Grammar grammar,
                 Symbols terminals) {
⋮
public class Follow extends
    SetMap<Symbol,Symbols> {

    public Follow(Grammar grammar,
                  Symbols symbols,
                  Symbols nonterminals,
                  First first) {
⋮
```

Grammar analysis (4 of 7)

- From a high level, SetMap is fairly straightforward:

```
import java.util.*;

public class SetMap<Key,Data> {

    private Map<Key,Data> _map;

    public SetMap(···) {
        _map=new TreeMap<Key,Data>();
⋮
```

- However, we have several problems:
  - Key objects must have a:

    ```
    int compareTo(Key key)
    ```

    method, so the map won't contain duplicate keys.

Grammar analysis (5 of 7)

- Problems (continued):
  - We'll need to construct new Data objects, but this is illegal in Java:

```
Data data=new Data();
```

    because Data is a generic parameter.
  - Data objects must have a:

```
void add(Key key)
```

    method, so we can maintain the map.

Grammar analysis (6 of 7)

- Problems (continued):
  - Data objects must also have:

```
int size()
Iterator<Key> iterator()
```

    methods, so we can do this:

```
public boolean
    add(Key key, Data data) {
  int size=size(key);
  for (Key k: data)
    put(key,k);
  return size!=size(key);
}
```

    The return value determines whether `this` for `key` was changed by the method. I suppose we could have used `get` to see if each `k` was already in `this` for `key`.

Grammar analysis (7 of 7)

- Fortunately, Java has a mechanism for placing such constraints on generic parameters:

```
import java.util.*;

public class SetMap<
  Key extends Comparable<Key>,
  Data extends Iterable<Key>
              & SetMapData<Key,Data>> {
⋮
```

- We also require Data to provide a way to create new Data objects.

  pub/parse/SetMapData.java

Scanning

- The scanner, called from Main, is very simple:
  - Tokens are separated by whitespace.
  - There are several kinds of token:
    * literals (e.g., `:` or `;`)
    * keywords (e.g., `":"`, `";"`, or `"for"`)
    * numbers (e.g., `123`)
    * identifiers (anything else)
- Why didn't I use Iterator(257)? A scanner doesn't create an iterator; it is an iterator. If anything, a String or File should create an iterator.

Parsing (1 of 2)

- This is a simple recursive-descent parser, designed according to the Interpreter(243), Flyweight(195), and Visitor(331) patterns.
- The idea is to have an abstract class declare a `parse` method:

```
public abstract class Node
    implements TreeIteratee<Node> {

    public abstract Node parse()
      throws ParseException;
⋮
```

- In addition, there is a class for every symbol in the grammar: terminals and nonterminals.

Parsing (2 of 2)

- Each `parse` implementation corresponds to the usual recursive-descent parsing procedure:
  - It has a Node member for each child in the parse tree.
  - It returns `this` as the parse-tree node. For example:

    pub/parse/NodeNext.java
  - The `eat` method is the interface to the scanner.
  - Parsing begins when `main` calls the constructor of:

    pub/parse/NodeRoot.java
  - All nodes have access to the scanner.

Interpreter(243) (1 of 5)

- Intent:
  - It defines a representation for a grammar, along with an interpreter that uses the representation to interpret sentences in the grammar's language.
- Motivation:
  - For the scanner/parser framework, we want a simple recursive-descent parser that builds parse trees.
  - We want a class for each grammar symbol.
  - The textbook's example demonstrates interpretation of sentences. We just want to parse them.

Interpreter(243) (2 of 5)

- Applicability:
  - You want to parse sentences in a language with a simple grammar. If it's not simple, use Bison.
  - You aren't too concerned about performance.
- Structure:

  pub/patterns/Interpreter.pdf
- Participants:
  - AbstractExpression (Node)
    - It declares a method common to all nodes in the parse tree (e.g., `parse`).
  - TerminalExpression (NodeSymbol)
    - It implements the method for a terminal symbol.
    - An instance is required for every terminal symbol in a sentence.

Interpreter(243) (3 of 5)

- Participants (continued):
  - NonterminalExpression (NodeNext)
    - It implements the method by calling the same-named method on objects corresponding to symbols on its productions' RHS's. Children are stored in instance variables. For example:

```
// A : X "+" Y ;
x=new X().parse();
eat("+");
y=new Y().parse();
```

Interpreter(243) (4 of 5)

- Participants (continued):
  - NonterminalExpression (continued)
    - For alternatives, it can "sniff" tokens and use exceptions. For example:

```
// A : X ;
//   | X A ;
x=new X().parse();
try {
  a=new A().parse();
} catch (ParseException e) {}
return this;
```

      This also demonstrates recursion.
  - Context (Lexer)
    - It is data that is global to the interpreter.
  - Client (NodeRoot)
    - It starts the parser.
    - It ensures that the parser finishes appropriately.

Interpreter(243) (5 of 5)

- Consequences:
  - Changing or extending the grammar is easy.
  - Implementing the classes is fairly mechanical.
  - Complex grammars are difficult to maintain.
  - Adding new forms of interpretation is easy: use Visitor(331).
- Implementation:
  - The textbook does not show how to build the parse tree, like my example does.
  - Type checking, code generation, and pretty printing are good Visitor(331) applications.
  - You can share tokens, with Flyweight(195)

An object for every symbol?

- The scanner/parser framework uses Flyweight(195) to avoid creating a new Symbol object if a similar Symbol object has already been created:

  `pub/parse/NodeSymbol.java`

Flyweight(195) (1 of 5)

- Intent:
  - It allows a large number of fine-grained objects to be represented by a small number of shared objects.
- Motivation:
  - For the scanner/parser framework, we want to represent a symbol as an object, but we don't want to create separate objects for "identical" symbols.
  - For Lexi, "identical" characters can be represented by a single shared object.
  - The similarity between two "identical" objects is provided by instance variables. This is called the *intrinsic state*.

Flyweight(195) (2 of 5)

- Motivation (continued):
  - The difference between two "identical" objects is provided by method arguments. This is called the *extrinsic state*.
  - For Lexi characters, the intrinsic state is the ASCII code, and the extrinsic state might be the font, size, and bounds.
- Applicability:
  - You want to represent many objects, by few objects, to save space.
  - Most state can be made extrinsic, resulting in a small set of shared objects.
  - Object identity isn't important.
- Structure:

  pub/patterns/Flyweight.pdf

Flyweight(195) (3 of 5)

- Participants:
  - Flyweight (Node)
    - It declares an interface to pass extrinsic state to flyweight methods.
  - ConcreteFlyweight (NodeSymbol)
    - It implements the Flyweight interface and stores intrinsic state.
  - UnsharedConcreteFlyweight (other Node subclasses)
    - Some objects of type Flyweight need not be shared. For example, if Glyph is the Flyweight interface, Row might be unshared.
  - FlyweightFactory (NodeSymbol)
    - It creates and manages Flyweight objects.
    - It enforces sharing of objects.

Flyweight(195) (4 of 5)

- Participants (continued):
  - Client (NodeSymbols)
    - It holds references to Flyweight objects.
    - It computes or stores extrinsic state.
- Collaborations:
  - A client passes extrinsic state to a Flyweight object, which holds intrinsic state.
  - A client must use the FlyweightFactory, or sharing will be subverted.
- Consequences:
  - It may be a time/space tradeoff.
  - If Composite(163) uses Flyweight(195) for leaf nodes, a leaf cannot store a reference to its parent.

Flyweight(195) (5 of 5)

- Implementation:
  - It removes extrinsic state from an object.
  - Shared objects are stored in some sort of table. Since there a few shared objects, reference counting or garbage collection (of them) isn't really necessary.
  - Lexi's ConcreteCompositor objects could be flyweights.

Code generation: from parse tree to grammar (1 of 3)

- We'll use Iterator(257) to traverse each parse-tree node.
- We'll use PostOrderIterator to traverse a whole parse tree.
- We'll use Visitor(331) to gather grammar parts and compose them into a grammar.
- A parse-tree node has zero or more children. References to them are stored in separate member variables, rather than in a Vector as with Lexi. In practice, the maximum number of children is small. So, for purposes of iteration, we can put them in a Vector, and reuse our VectorIterator class.
- The abstract Node class implements `ci` methods, to create Vector-based iterators. It also declares `createIterator` and `accept`:

  pub/parse/Node.java

Code generation: from parse tree to grammar (2 of 3)

- Concrete subclasses of Node implement `createIterator` and `accept`:
  pub/parse/NodeFirst.java
- The Visitor interface is trivial:
  pub/parse/NodeVisitor.java
- The Visitor implementation is straightforward:
  - The constructor is passed a new Grammar object, so the client has a reference to the resulting grammar.
  - As we visit various parts of the parse tree, we:
    * Add rules to the grammar.
    * Add symbols to the current rule's RHS.

    pub/parse/NodeToGrammarVisitor.java

Code generation: from parse tree to grammar (3 of 3)

- Finally, the traversal/visit process is started by the Grammar constructor:

```
private Vector<Rule> _grammar;

public Grammar() {
  _grammar=new Vector<Rule>();
}

public Grammar(Node root) {
  this();
  NodeVisitor v=
    new NodeToGrammarVisitor(this);
  TreeIterator<Node> i=
    new PostorderIterator<Node>(root);
  for (i.init(); !i.done(); i.next())
      i.item().accept(v);
}
```