Comp-303 : Programming Techniques
Lecture 20

Alexandre Denault
Computer Science
McGill University
Winter 2004

Last lecture . . .

- Facade allows us to hide a complex subsystem, thus reduce coupling and make that subsystem easier to use.
- Adapter allows us to change the interface of a class, thus allowing us to that class in our hieratic (polymorphic behavior).

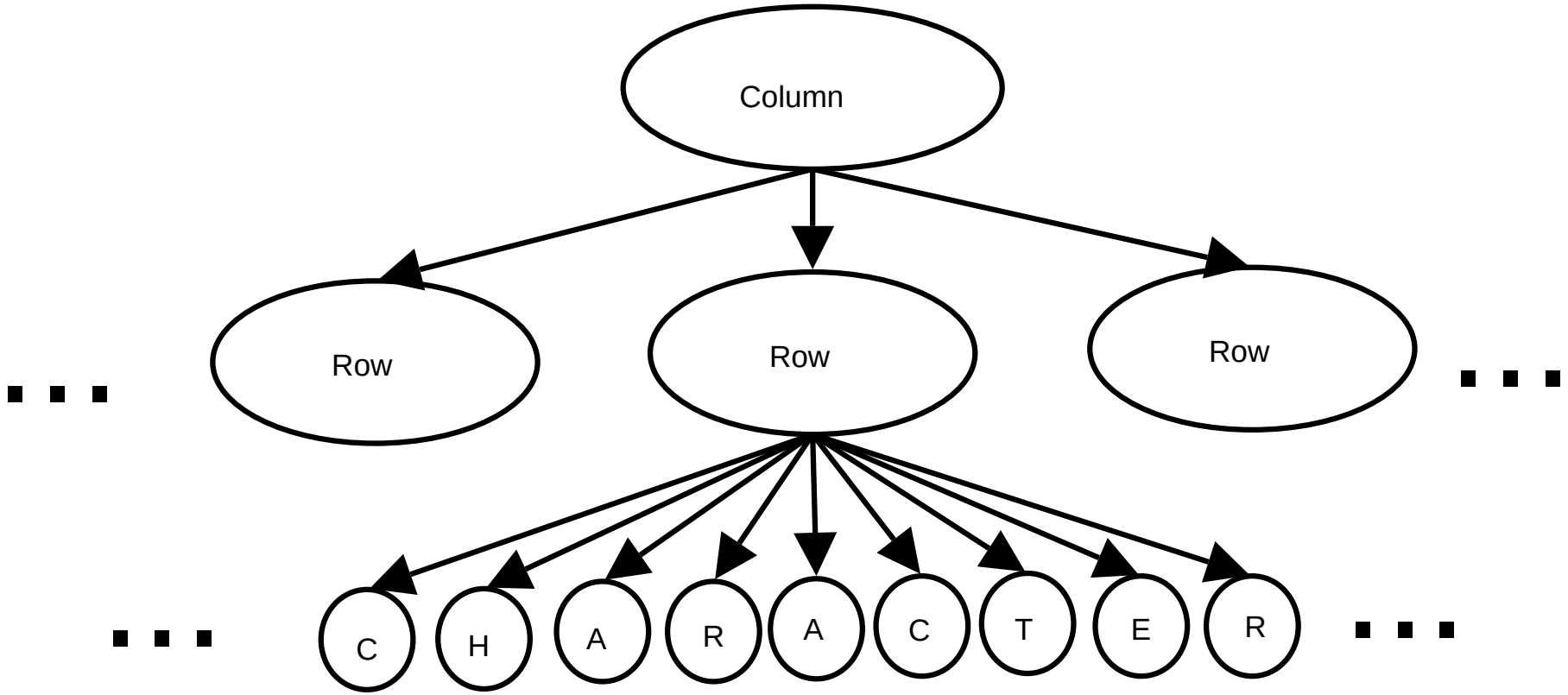
Typical Situation

- Today, our example starts with a word processing application.
- For added flexibility, you would like each letter in the document to be stored as an object.
- However, representing each character as an object would create to many object (and add a lot of overhead on the system).
- How can we have the flexibility of having each letter an object without the overhead?

Initial Application Architecture

- The text in our word processor application is stored in three classes: *Column*, *Row* and *Letter*.
- Each document has at least one *Column* object.
- A *Column* has one or more *Row*.
- A *Row* has one or more *Letter*.

Too many objects!



Flyweight

- *Pattern Name* : Flyweight
- *Classification* : Structural
- *Intent* : Use sharing to support large numbers of fine-grained objects efficiently.

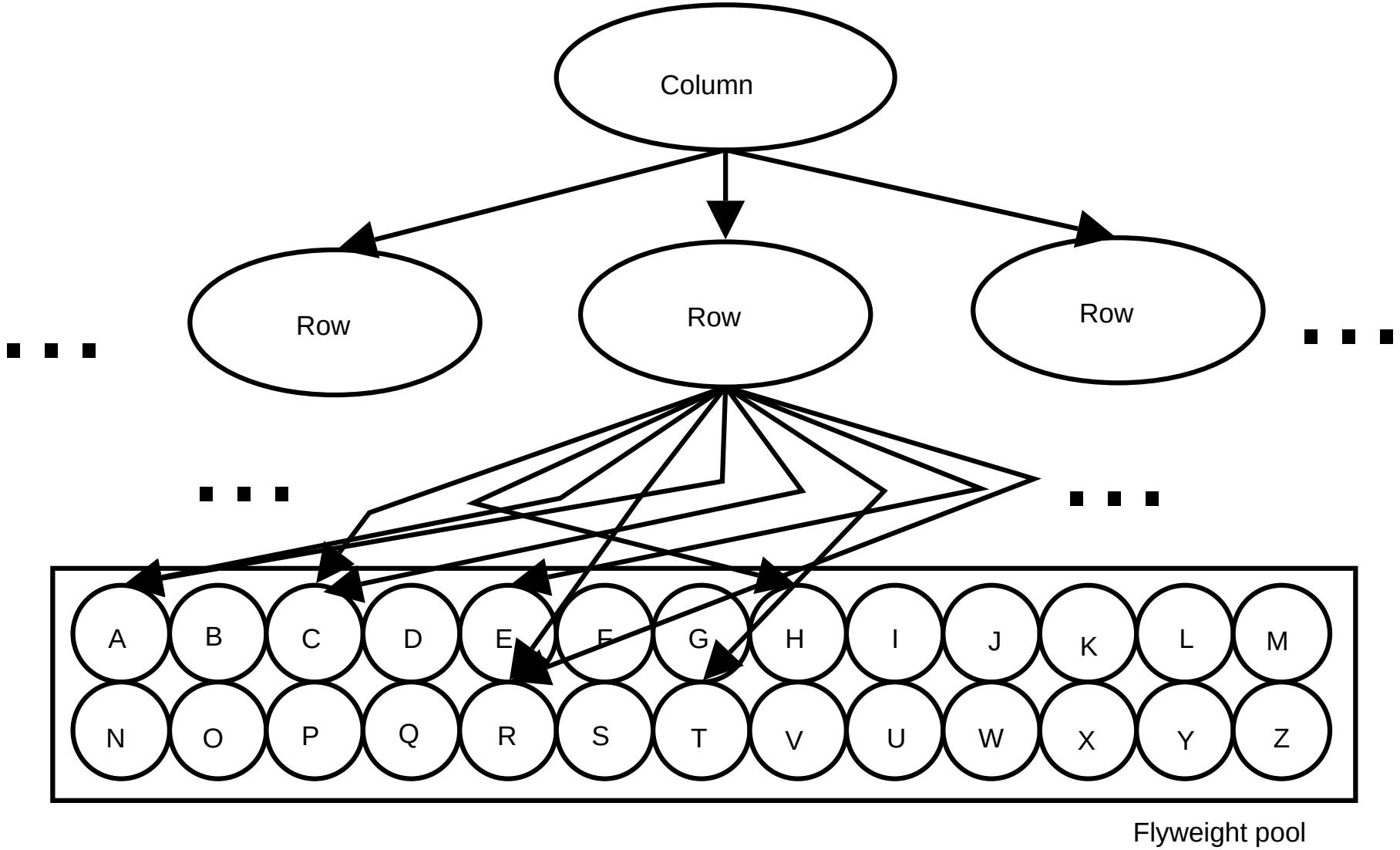
Motivation

- Using the FlyWeight object, we can reuse letter objects.
- The trick is to separate the intrinsic state from the extrinsic state of each object:
  - Intrinsic state: information that's independent of the flyweight's context. This information should be store inside the flyweight.
  - Extrinsic state: information that depends and varies with the flyweight's context (thus shouldn't be shared). This information should be stored outside the flyweight.
- In our example, the position, the size and the font of the character is store in the extrinsic state.
- Thus, we are able to share *Letter* objects.

Motivation (cont.)

- Instead of using a *Row* for each line of text, we use a *Row* to delimit text with a specific style. Thus, the font, style and size is store in the row.
- Only 256 characters of a given font style need to be created.
- The application keeps track of a flyweight pool, thus allowing it to reuse existing flyweights.
- Thus the number of flyweight we need is determined by the number of styles in our document, not the document size.

Motivation (cont.)

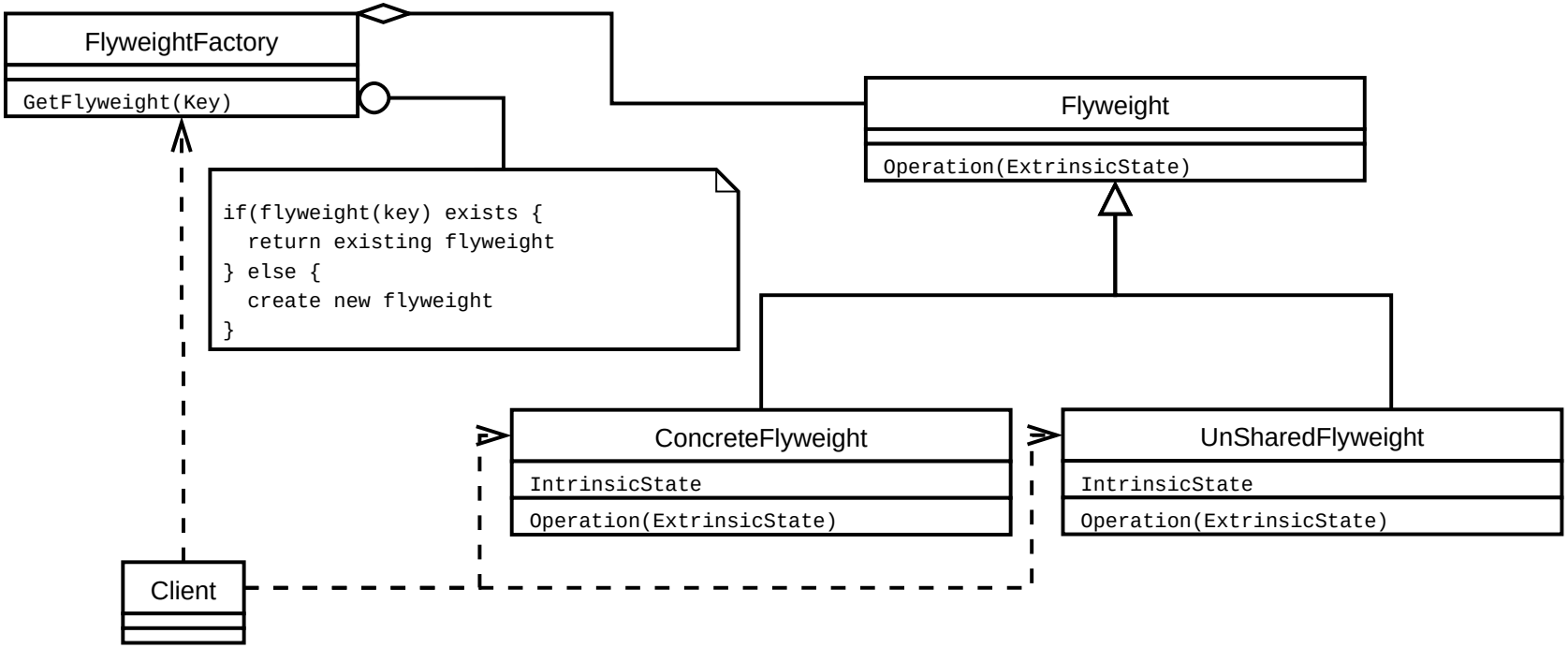


Applicability

The Flyweight pattern's effectiveness depends heavily on how and where it's used. Flyweight should only be used when all the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Objects can be replaced by few shared objects once the extrinsic state is removed.
- The application doesn't depend on object identity.

Structure


FlyweightFactory
GetFlyweight(Key)
Flyweight
Operation(ExtrinsicState)
if(flyweight(key) exists {
return existing flyweight
} else {
create new flyweight
}
ConcreteFlyweight
IntrinsicState
Operation(ExtrinsicState)
UnSharedFlyweight
IntrinsicState
Operation(ExtrinsicState)
Client

Participants

- Flyweight : declares the interface that flyweights use to act on extrinsic state.
- ConcreteFlyweight : implements the Flyweight and adds storage for the intrinsic state (which should be sharable).
- UnsharedConcreteFlyweight: not all Flyweight need to be shared.
- FlyweightFactory: creates and manages flyweight objects.
- Client: stores the extrinsic state of the flyweights.

Collaborations

- The Client passes the extrinsic state to the flyweight when invoking operations.
- The Client should never create a Flyweight directly. It should always use the FlyweightFactory to create new objects.

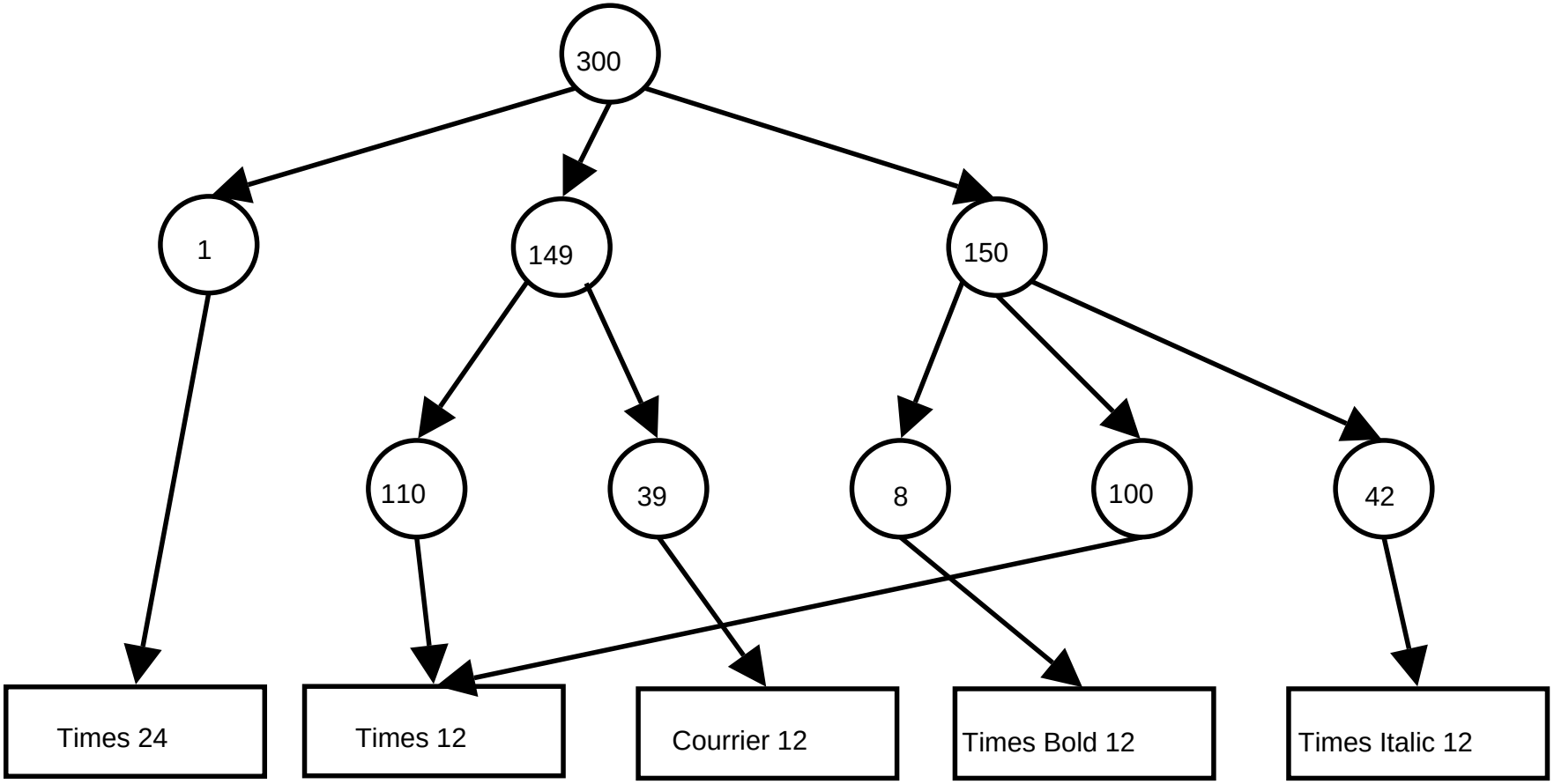
Consequences

- By increasing the amount of data that is shared, we can increase our savings in space (memory, disk, etc).
- This saving is heavily influenced by the number of flyweight we need to create.
- However, the use of Flyweights may introduce some performance overhead.
- Since some data is stored outside the object, there are some performance penalties (retrieving that data).

Implementation

- The first step when building a flyweight is to decide what data should be intrinsic and what data should be extrinsic.
- This decision is critical since it will heavily influence the flyweight's performance.
- The next step is to design the FlyweightFactory.

BTree


300
1
149
150
110
39
8
100
42
Times 24
Times 12
Courrier 12
Times Bold 12
Times Italic 12

Example

```
public abstract class Glyph {

  public abstract void draw(Context contextValue);

  public abstract void insert(Glyph glyphValue, int i);
  public abstract void remove(int i);

  public abstract void SetFont(Font fontValue);
  public abstract Font GetFont();
}
```

Example

```
public class GlyphContext extends Glyph {

  Glyph[] glyphList;
  Font   currentFont;
  int  index

  GlyphContext() {
   \\ Create an empty context }

  public abstract void draw(Context contextValue) {
   \\ draw all glyphs in glyphList at position contextValue }

  public abstract void insert(Glyph glyphValue, int i) {
   \\ adds the glyph to the context at position i}

  public abstract void remove(int iPosition) {
    \\ removes the context a position i from the context  }
}
```

Example

```
public class Letter extends Glyph {

  char  currentChar;
  Font   currentFont;

  Letter(char charValue, Font newFont) {
    \\ Create the letter charValue. }

  public abstract void draw(Context contextValue) {
   \\ draw the letters at position contextValue  }

  public abstract void insert(Glyph glyphValue, int i) {
   \\ throws an exception }

  public abstract void remove(int iPosition) {
    \\ throws an exception }
}
```

Example

```
public class FlyweightFactory {

  Glyph[] poolLetters;

  public Glyph Create(char charValue, Font newFont) {
   \\ check pool to see if letter (same char and font) already exist
   \\   if so, returns existing letter
   \\   else creates and returns new letter
  }
}
```

Known Uses

- Word Processors
- Guis (widgets)

Related Patterns

- Composite
- State
- Strategy
- *Factory*
- *Singleton*

Project Submission

- On April 8th (before the midterm), each team must handin a CDROM with the following items:
  - Source code for each modules (well commented)
  - Compiled code (class files)
  - Javadoc
  - A Readme file explaining
    - the content of the CDRom
    - instruction on how to compile the program
    - instruction on how to run the program
  - Any other documentation you might find important.
- The source for each module should be clearly separated (different directories).
- You can handin the CDRom the day of the interview, but you will suffer a small late penalty (1 point).

Project Submission

- Don't forget to handing any paper documentation you might also have.

Project Interview

- Project Interview will be held (tentivaly) Thursday, April 15th and Friday, April 16th from 10h00 to 18h00.
- You team needs to sign up for a block of time. You can sign up by sending me an email. First come, first served.
- Total time required for interview will be 30 minutes + 15 minutes per team member.
- Add an extra 15 minutes if you want to demonstrate your program in Trottier.
- If you want to show me something that uses network, you need to use Trottier (don't have space in my office for multiple laptops).
- I reserve the right to move the time slots a little.

Project Interview (cont.)

Features evaluated during the interview include (but are not limited to . . .

- Quality of documentation
- Complexity of Project
- Stability/Reliability of Project
- Ease of Use
- Modularity (Amount of coupling between modules)
- Ability to explain and justify design choices
- Abstraction (Quality of O.O. coding)
- Style of coding (repetition, indentation, ease of searching)

Summary

- Flyweights allows us to deal with situation where a large number of objects could be required.
- There is still a lot of work to do on the project and not a lot of time to do it.

Tool of the day: SableCC

- SableCC is an object-oriented framework that generates compilers (and interpreters) in the Java programming language.
- Think of it as an object-oriented Java version of flex and bison.
- First, the framework uses object-oriented techniques to automatically build a strictly typed abstract syntax tree.
- Second, the framework generates tree-walker classes using an extended version of the visitor design pattern.
- More information on SableCC is available at:

  `http://www.sablecc.org/`

References

- These slides are inspired (i.e. copied) from these three books.
  - Design Patterns, Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; Addison Wesley; 1995
  - Java Design Patterns, a Tutorial; James W. Cooper Addison Wesly; 2000
  - Design Patterns Explained, A new Perspective on Object Oriented Design; Alan Shalloway, James R. Trott; Addison Wesley; 2002