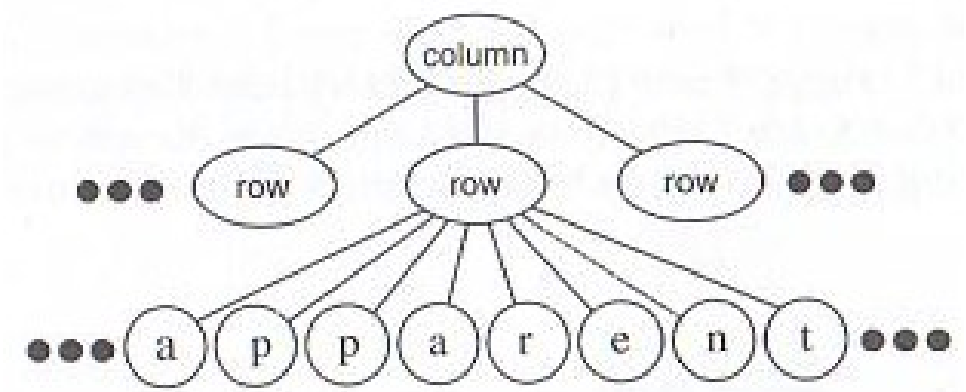
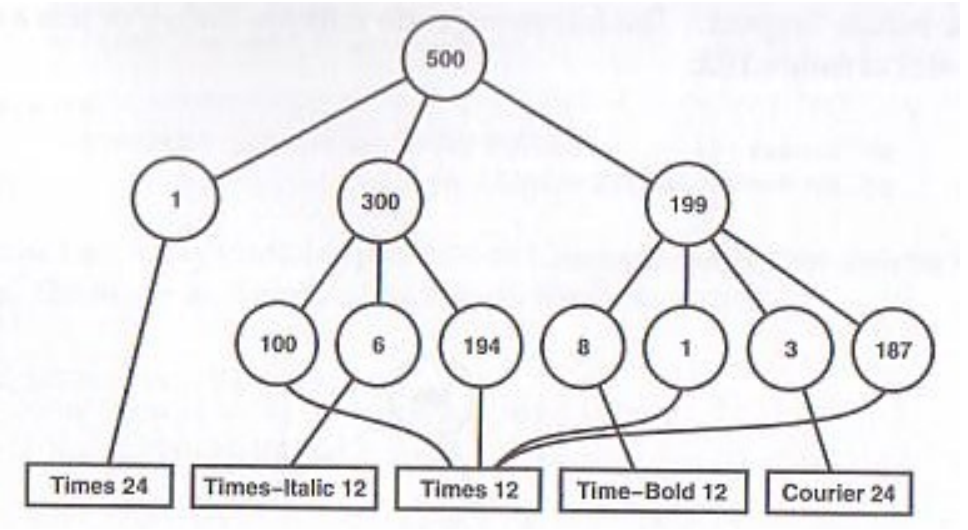
Plan for Today

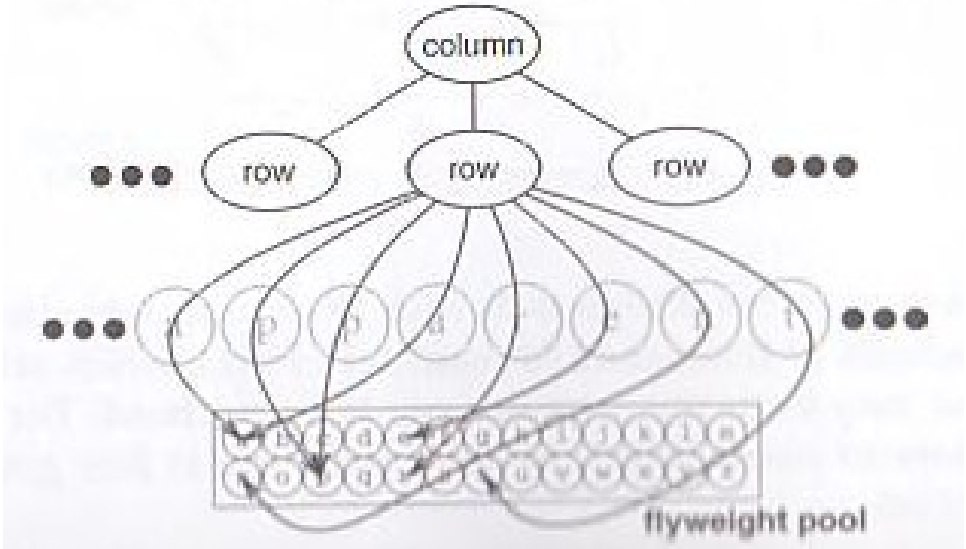
- Prototype pattern
- Memento pattern
- Iterator pattern

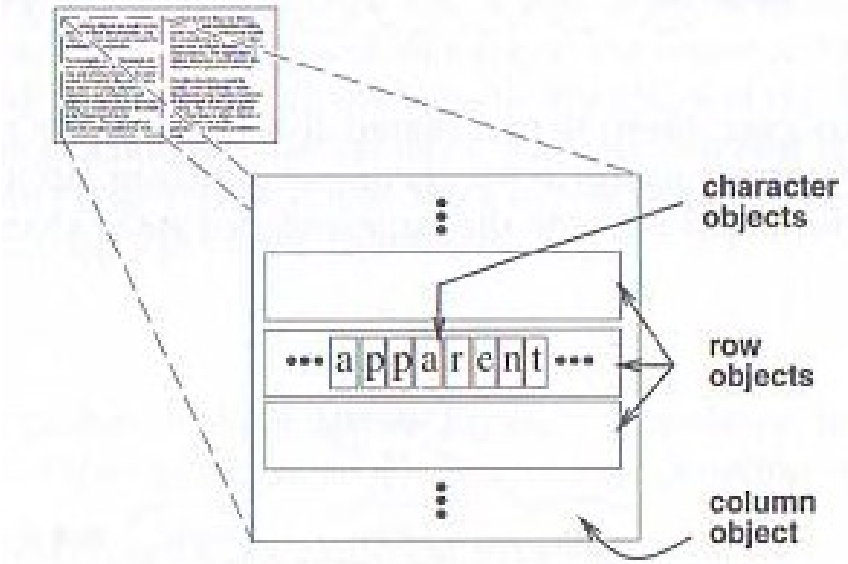
Lecture 49 – 1 December

Task 7 Questions?







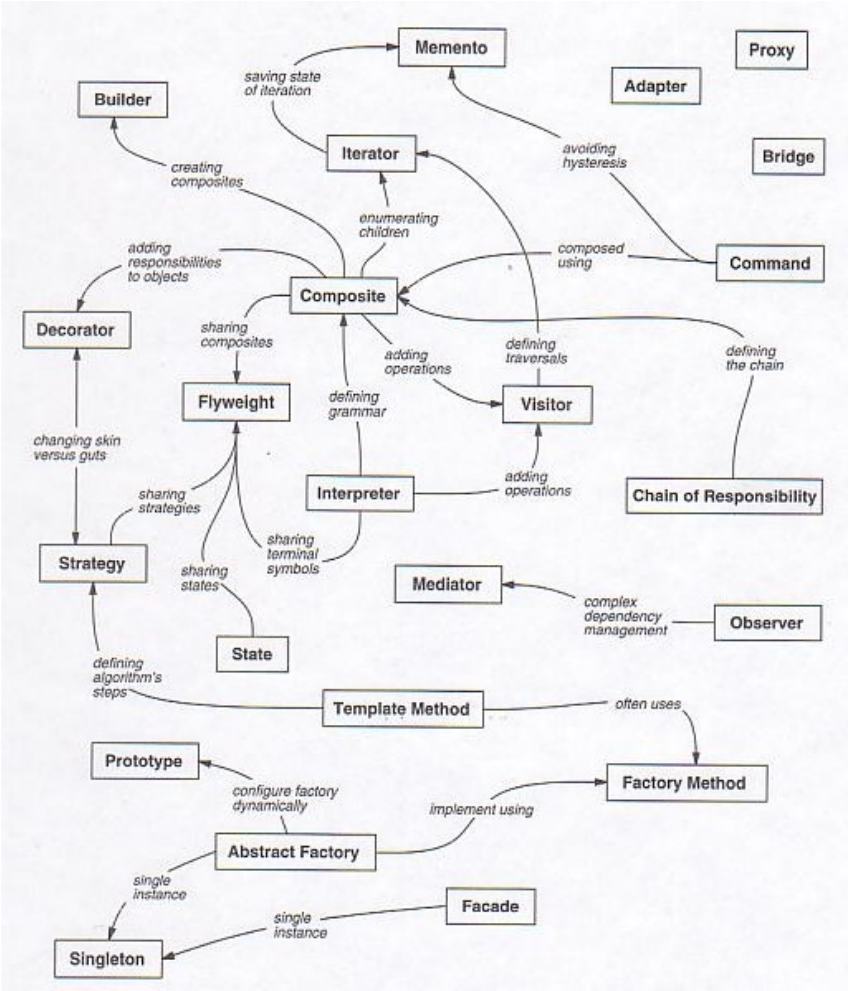


Introduction · Objectives

- To understand concepts and techniques of "advanced programming"
- To recognize inherent patterns in every problem
- To learn to assess, choose, apply, and evaluate critically
  - problem space and solution space
  - design patterns and antipatterns

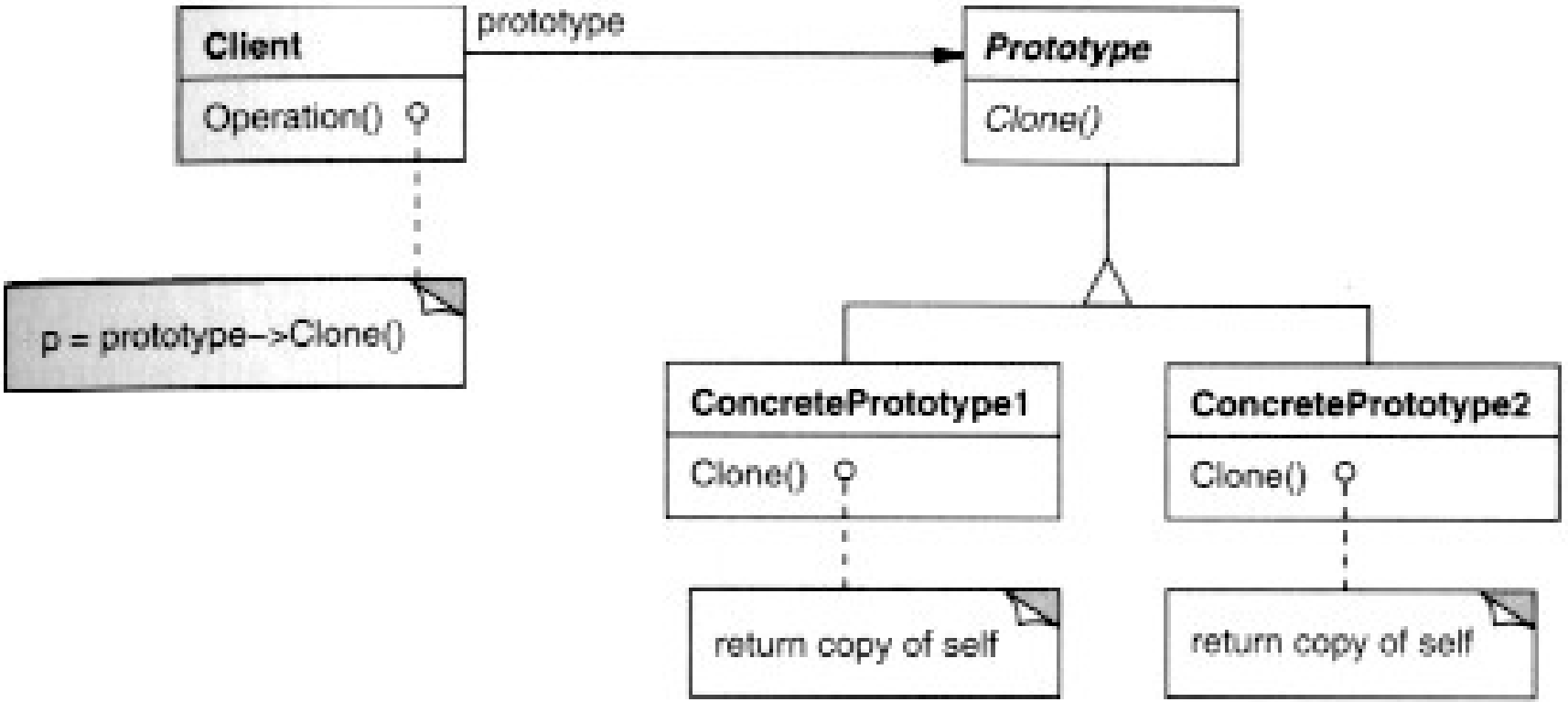
THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		



Prototype Pattern

- Specifies kinds of objects to create using prototypical instance and creates new objects by copying it

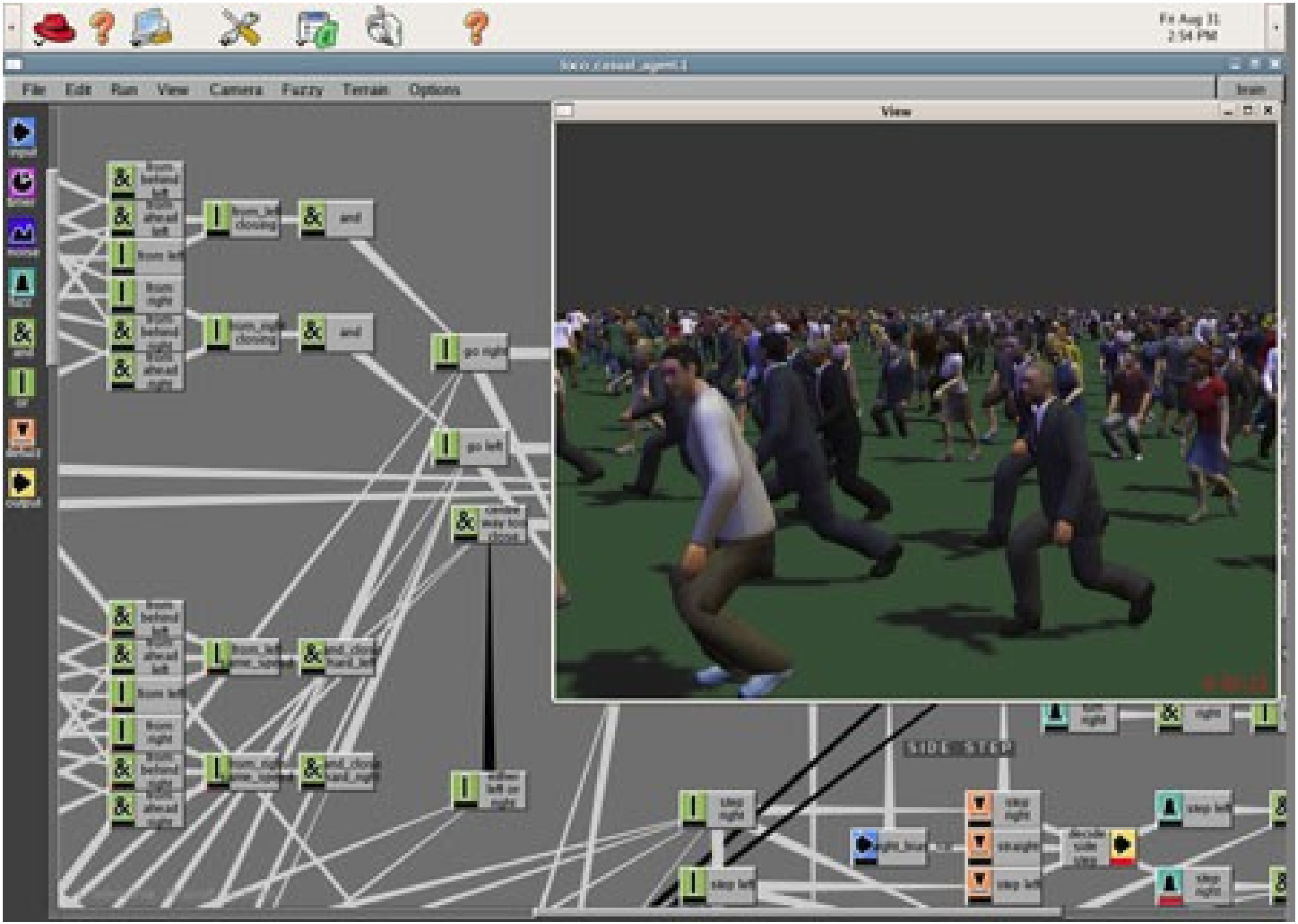


Prototype Pattern

- Rubber-stamping prototypical instances to populate set of identical objects
- Registry-based
  - centralized registry of prototypes, one each
  - request copy (unique instance) of any prototype
    - differs from flyweight in unique instanceness
  - delivered by copying existing prototype, not by instantiating new one
    - saves on construction costs
  - dynamic registry population at runtime
    - plug-and-play rubber stamps

Prototype Pattern

Prototype Pattern



Prototype Pattern

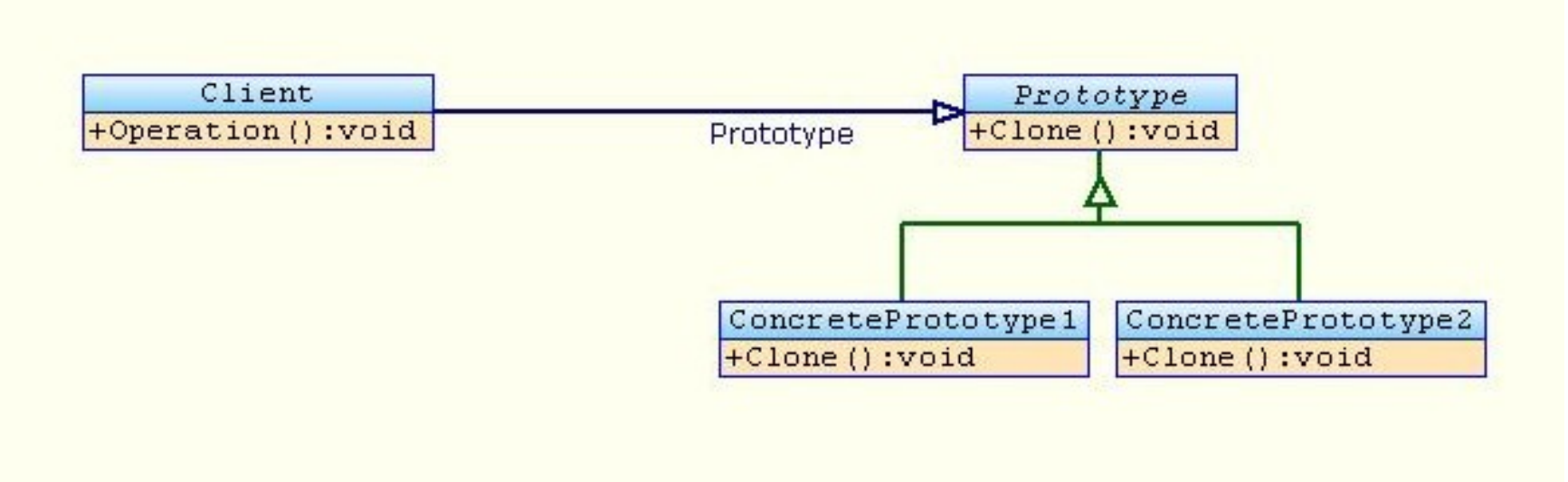
Monsters

```
public abstract class A_Monster { shared monster stuff }
public class WerewolfMonster extends A_Monster { werewolf stuff }
public class VampireMonster  extends A_Monster { vampire stuff  }
```

Monster Registry

```
public class MonsterRegistry { // possibly singleton, but then could not be subclassed
  private Map<String,A_Monster> _registry = new HashMap<String,A_Monster>();

  public MonsterRegistry() {
    _registry.put("werewolf", new WerewolfMonster());
    _registry.put("vampire",  new VampireMonster());
  }

  public A_Monster getMonster(String type) {
    A_Monster monster = _registry.get(type).clone();
    // do any setup here
    return monster;
  }
}
```

Prototype Pattern



source: wikipedia

[Review] : Class : Features : clone()

- Self-replicating mechanism
  - create unique, independent shallow copy
    - *shallow* clone
      - just the object
      - for example: `ArrayList`
    - *deep* clone
      - the object and any embedded objects
      - for example: `ArrayList`

?

```
public class Widget implements Cloneable
{
  private String _name;

  private Date _date = new Date();

  public Widget(String name)
  {
    _name = name;
  }

  public Object clone()
  {
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      ...
    }
  }
}
```

```
Widget w1 = new Widget("billy");
Widget w2 = (Widget) w1.clone();

System.out.println(w1);   // billy : Tues 29 Dec 2005 08:00.00 MDT
System.out.println(w2);   // billy : Tues 29 Dec 2005 08:00.01 MDT
```

Prototype Pattern : Cloning

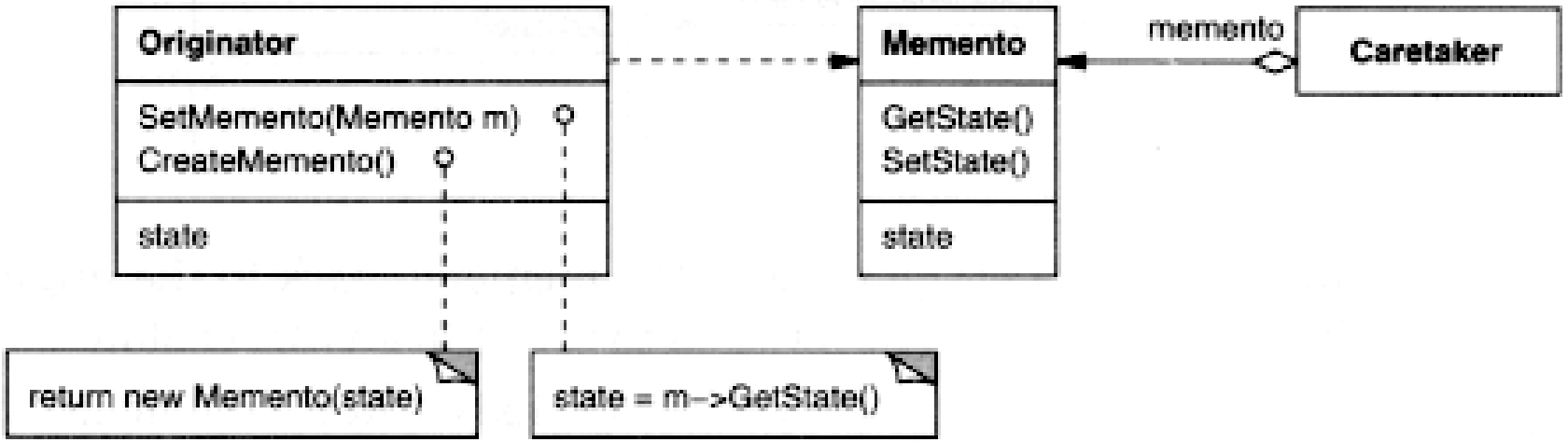
- Shallow clone
  - copy object
  - reuse references (pointers) within object
  - `clone()` method
- Deep clone
  - copy object
  - recursively clone all references within object
  - serialization approach:

consider Composite pattern

```
import java.io.*;
public abstract class A_Monster implements Serializable {
   shared monster stuff
}
```

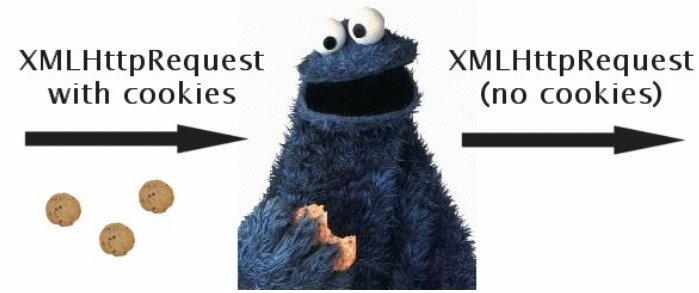
Memento Pattern

- Captures and externalizes internal state of object without violating encapsulation so it can be restored to same state later



Memento Pattern

- Variation on Visitor pattern
  - "self-visitor"
    - get own state and store (or allow external entity to store without inspection)
    - nobody else can use state (opposite of Visitor pattern): does not expose innards
  - externalizes internal state for later recreation
    - do/undo
    - store/reload; save game; cookies



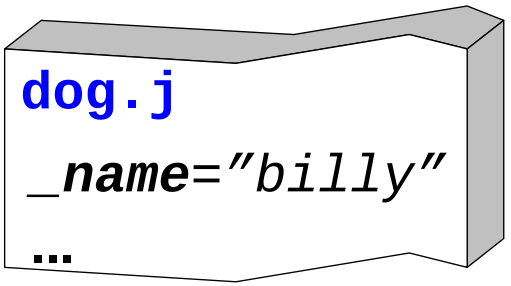
- Undo strategies
  - Command pattern reverses action
  - Memento pattern recalls past snapshot
- Store strategy
  - storage formalism independent of implement formalism
- Be careful about retaining mementos
  - garbage collection inhibited

[Review] : Class : Input / Output : Serialization

- Writes data to binary file
  - persistent stored
  - `transient` not stored

```
import java.io.*;

public class Widget implements Serializable
{
  private String _name;

  private transient Date _date = new Date();

  public Widget(String name)
  {
    _name = name;
  }
}
```

```
Widget w = new Widget(“billy”);
try
{
  ObjectOutputStream out =
    new ObjectOutputStream(new FileOutputStream("dog.j"));
  out.writeObject(w);
}
catch (IOException exception)
{
  ...
}
```

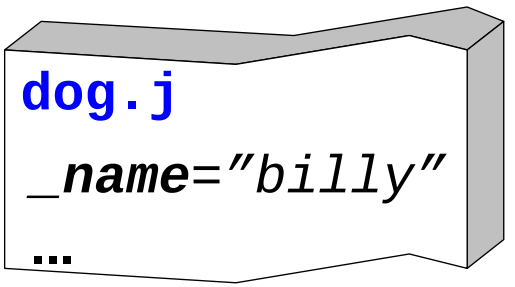


[Review] : Class : Input/Output : Deserialization

- Reads data from binary file
  - persistent read
  - `transient` reinitialized

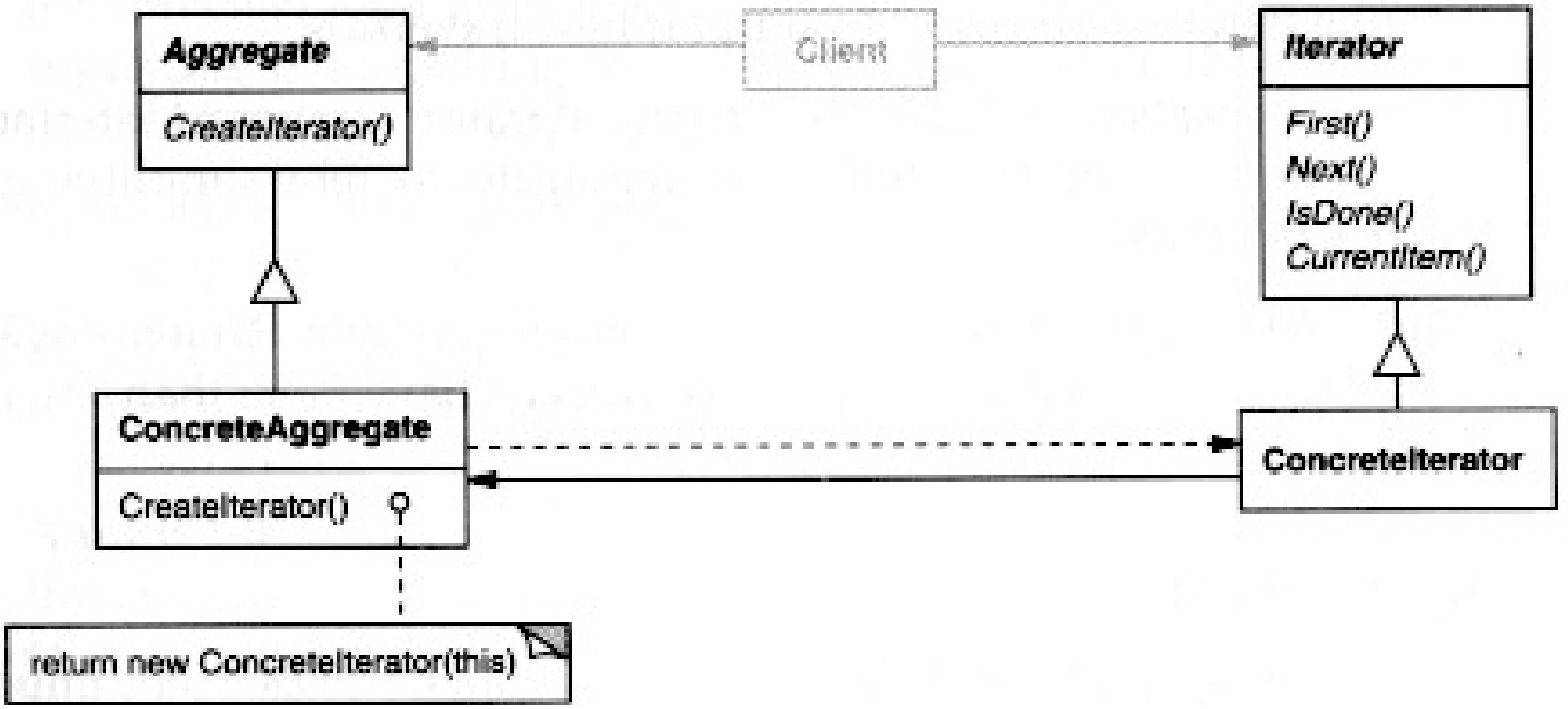
```
import java.io.*;

public class Widget implements Serializable
{
  private String _name;

  private transient Date _date = new Date();

  public Widget(String name)
  {
    _name = name;
  }
}
```

```
try
{
  ObjectInputStream in =
    new ObjectInputStream(new FileInputStream("dog.j"));

  Widget w = (Widget) in.readObject();
}
catch (IOException exception)
{
  ...
}
```



Iterator Pattern

- Provides means to access elements of aggregate object sequentially without exposing underlying representation



Iterator Pattern

- Properties
  - provides way to access elements of aggregate object sequentially without exposing underlying representation
  - supports multiple traversals of aggregate objects
  - provides uniform interface for traversing different aggregate structures
- Variants
  - forward/reverse
  - immutable/mutable
  - unfiltered/filtered
- Interface
  - `java.util.Iterator`

```
List<String> a = new ArrayList<String>();

a.add("dog");
a.add("cat");

// direct approach
for (String entry : a)
{
  System.out.println(entry);
}

// indirect (iterator) approach
for (Iterator i = a.iterator(); i.hasNext() ; )
{
  System.out.println(i.next());
}
```

Iterator Pattern

- Example of forward/reverse iterator

```
public class ForwardReverseIterator {

  private final List<String> _list;
  private int                _index = 0;

  public ForwardReverseIterator(List<String> list) {
    _list = list;
  }

  public String next() {
    return _list.get(_index++);
  }

  public String previous() {
    return _list.get(_index--);
  }

  public boolean hasNext() {
    return (_index < _list.size());
  }

  public boolean hasPrevious() {
    return (_index > 0);
  }
}
```

Iterator Pattern

- Example of mutable iterator

```
public class MutableIterator {

  private final List<String> _list;
  private int                _index = 0;

  public MutableIterator(List<String> list) {
    _list = list;
  }

  public String next() {
    return _list.get(_index++);
  }

  public boolean hasNext() {
    return (_index < _list.size());
  }

  public void remove() {
    _list.remove(_index);
  }
}
```

Iterator Pattern

- Example of filtered iterator

```
public class FilteredIterator {

  private final List<String> _list;
  private final String       _query;
  private int                _index = 0;

  public FilteredIterator(List<String> list, String query) {
    _list  = list;
    _query = query;
  }

  public String next() {
    // return next _list entry containing _query
  }

  public boolean hasNext() {
    // return whether there is a next _list entry containing _query
  }
}
```