Design Patterns

Outline

- Purpose
- Useful Definitions
- Pattern Overview

Purpose

To provide programmers with already documented solutions to common problems.

Gives the programmers a common language.

COMPOSITION?

Patterns != Frameworks

- A framework provides actual code. You use patterns in a framework to create the code.
- If someone gives you a pattern you’ll get a list of diagrams, it’s a concept. A framework consists of actual code. (Remove)

Useful Definitions

- Object – A package for both data and procedures (methods, functions) that operate on that data
- Class – Definition of an object implementation
- Encapsulation – Abstract away implementation details of a given object
- Interface – All of the method signatures of a given object

Useful Definitions (cont’d)

- Inheritance – Sub-classing one object to another so it can inherit some properties of its parent while creating more specific details for itself
    - Good: Subclasses are nice. A simple concept and easy to use.
    - Bad: Static, tied to it. When you change one thing you might have to change lots of classes. Inheritance is determined at compile time, while aggregation is determined at run time.
- Dynamic Binding – The run-time association of a request to an object and one of its operations (methods)
- Polymorphism – The ability to substitute one object for another without having to change any implementation details

Useful Definitions (cont'd)

- Instantiation – The act of creating an object (a.k.a. an *instance* of a class)
- Abstract class – A class whose main purpose is to define a common interface for its subclasses
- Abstract operation – A declaration of a method with no implementation details
- Concrete classes – A class that contains implementation details.
- Override – Allowing a subclass to handle its method calls on its own by changing the implementation of its parent

Useful Definitions (cont'd)

- Aggregation – One object owns or is responsible for another object. The second object is a part of the first. Both objects have identical lifespans
- Aquaintance – One object knows of another, so it can make method calls to it, however, neither object's lifespan is dependent on the other's.

Pattern Overview

- State
- Template
- Composite
- Command
- Strategy
- Mediator

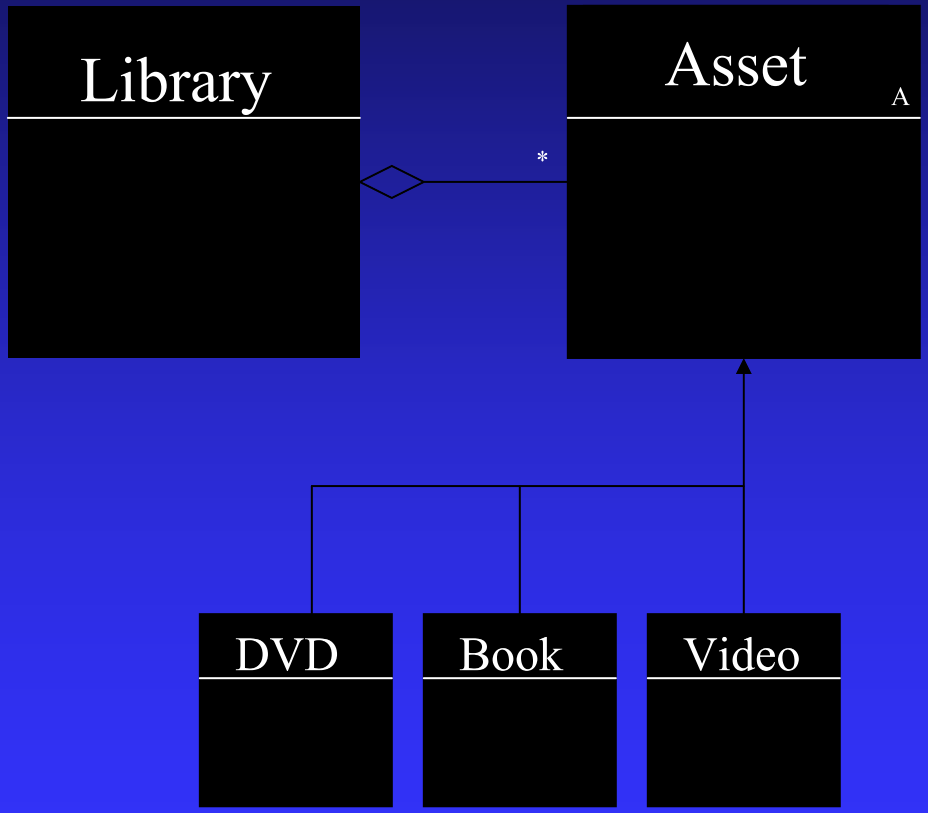
State Pattern

- Intent:
  - Provide the ability for an object to change its behavior in response to internal state changes.

State Pattern

Library

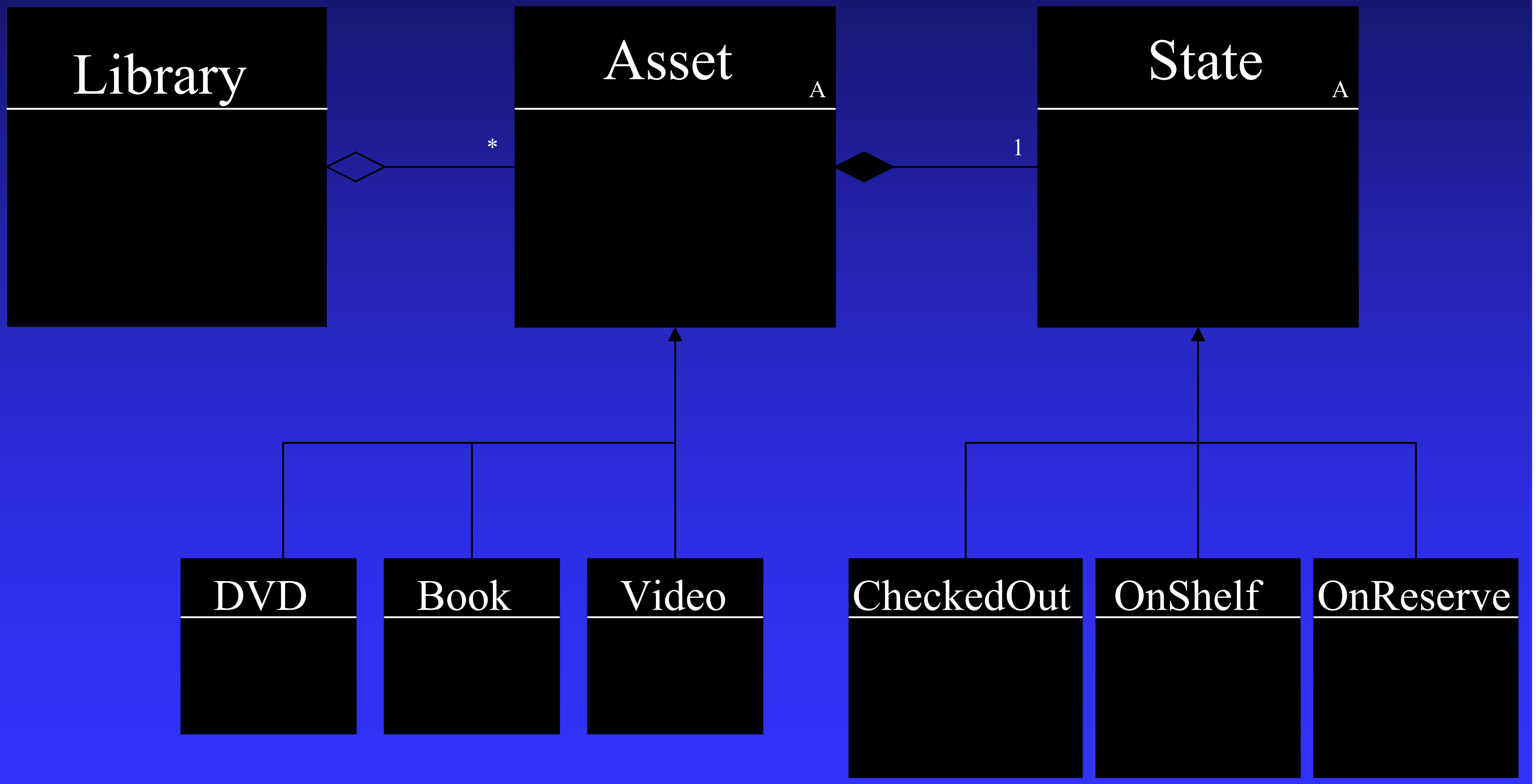
State Pattern



◇ = acquaintance

A = abstract

State Pattern



Library

*

Asset A

1

State A

DVD

Book

Video

CheckedOut

OnShelf

OnReserve

◇ = acquaintance ◆ = aggregation A = abstract

State Pattern

```
public class Library {
        private List assets = new List<Asset>();
        …
}

abstract public class Asset {
        State state = new onShelf();

        protected boolean checkOut() {
                if(state.checkOut() = = true) {
                    changeState(new CheckedOut());
                    return true;
                }
                else
                    return false;
        }
        protected boolean putOnShelf() { … }
        protected boolean putOnReserve() { … }
        protected void changeState(State newState) {
                state = newState;
        }
}
```

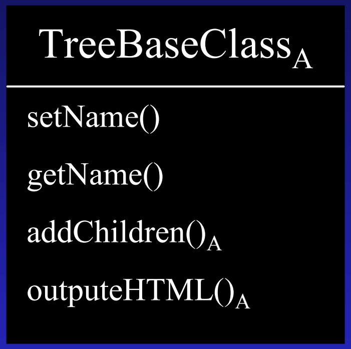
State Pattern

```
abstract public class State {
    protected boolean checkOut() { return false; }
    protected boolean putOnShelf() { return false; }
    protected boolean putOnReserve() { return false; }
}
public class CheckedOut extends State{
    private boolean putOnShelf() { return true; }
}
public class OnShelf extends State {
    private boolean checkOut() { return true; }
    private boolean putOnReserve() { return true; }
}
public class onReserve extends State {
    private boolean checkOut() { return true; }
}
```

Template Method

- Intent:
  - Create a skeleton for an algorithm, while allowing subclasses to redefine certain steps.

Template Method



TreeBaseClass$_A$

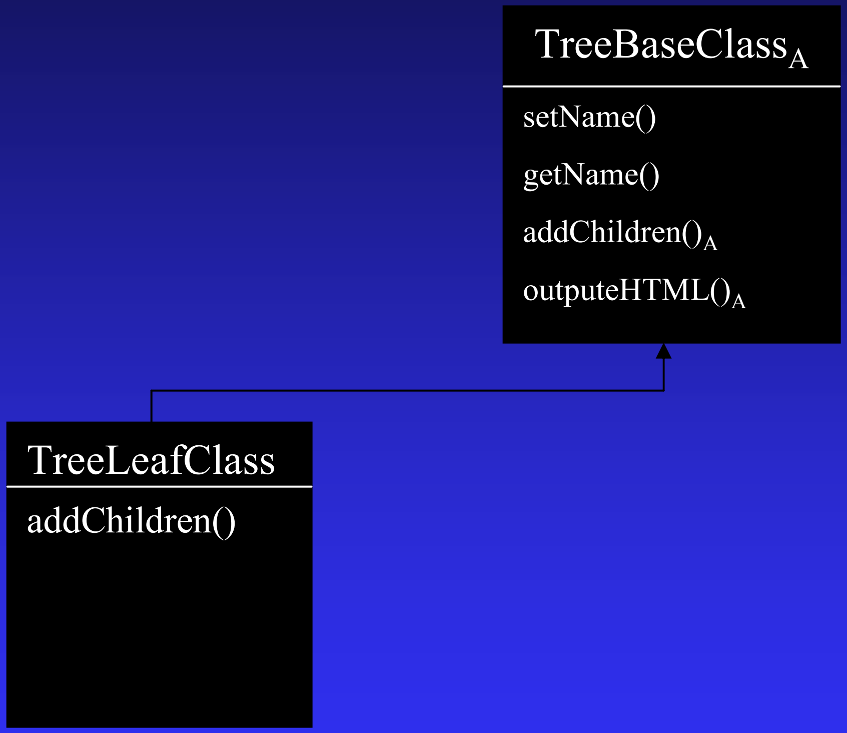
setName()

getName()

addChildren()$_A$

outputeHTML()$_A$

Template Method



TreeBaseClass$_A$

setName()

getName()

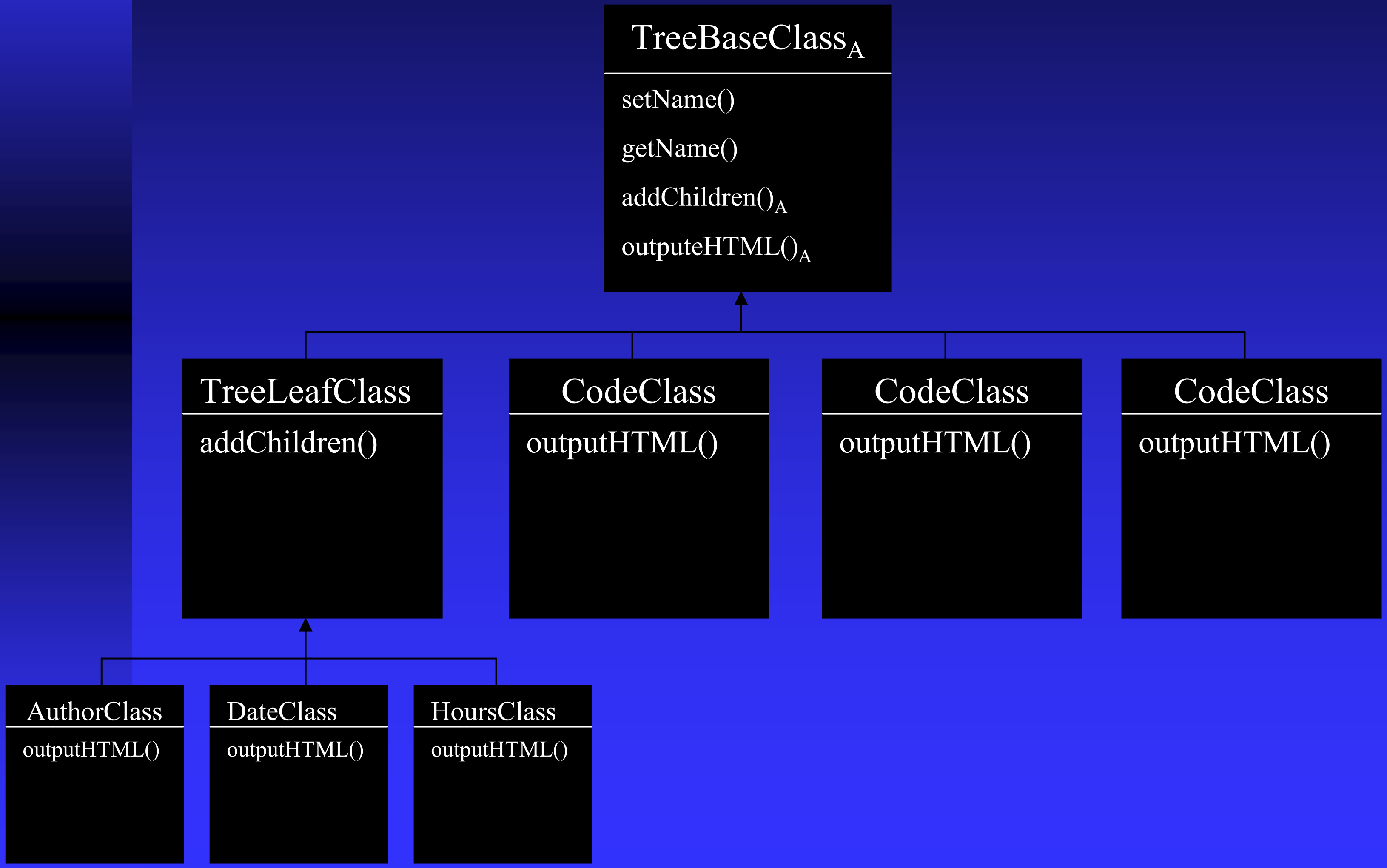
addChildren()$_A$

outputeHTML()$_A$

TreeLeafClass

addChildren()

Template Method



TreeBaseClass$_A$

setName()

getName()

addChildren()$_A$

outputeHTML()$_A$

TreeLeafClass

addChildren()

CodeClass

outputHTML()

CodeClass

outputHTML()

CodeClass

outputHTML()

AuthorClass

outputHTML()

DateClass

outputHTML()

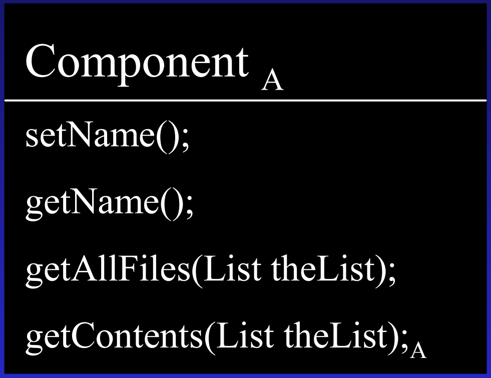
HoursClass

outputHTML()

Composite Pattern

- Compose an object into a tree structure. Let clients treat individual objects and compositions of objects as the same thing.

Composite Pattern



Component $_A$

setName();

getName();

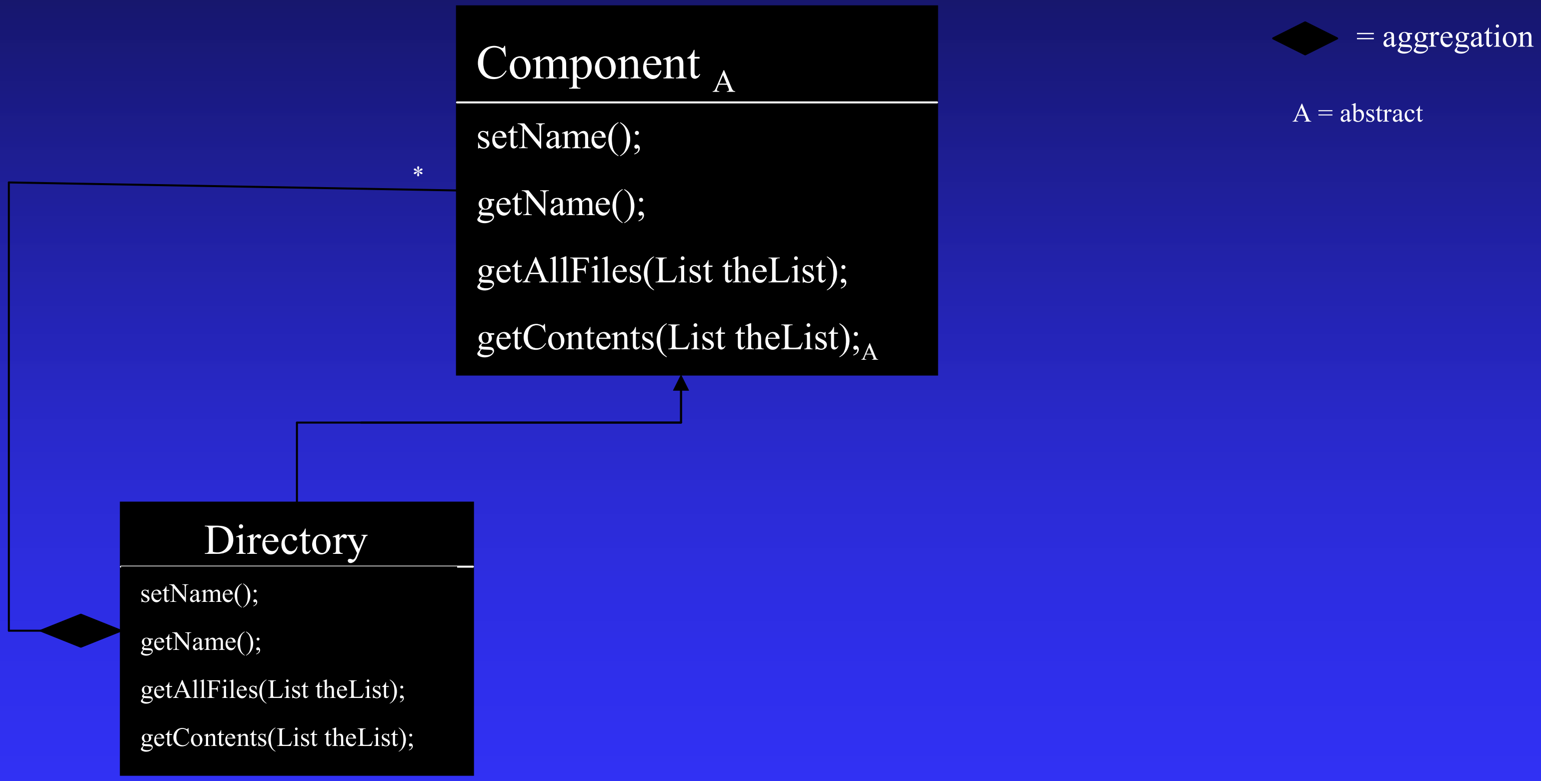
getAllFiles(List theList);

getContents(List theList);$_A$

◆ = aggregation

A = abstract

Composite Pattern



Component $_{A}$

setName();

getName();

getAllFiles(List theList);

getContents(List theList);$_{A}$

*

Directory

setName();

getName();

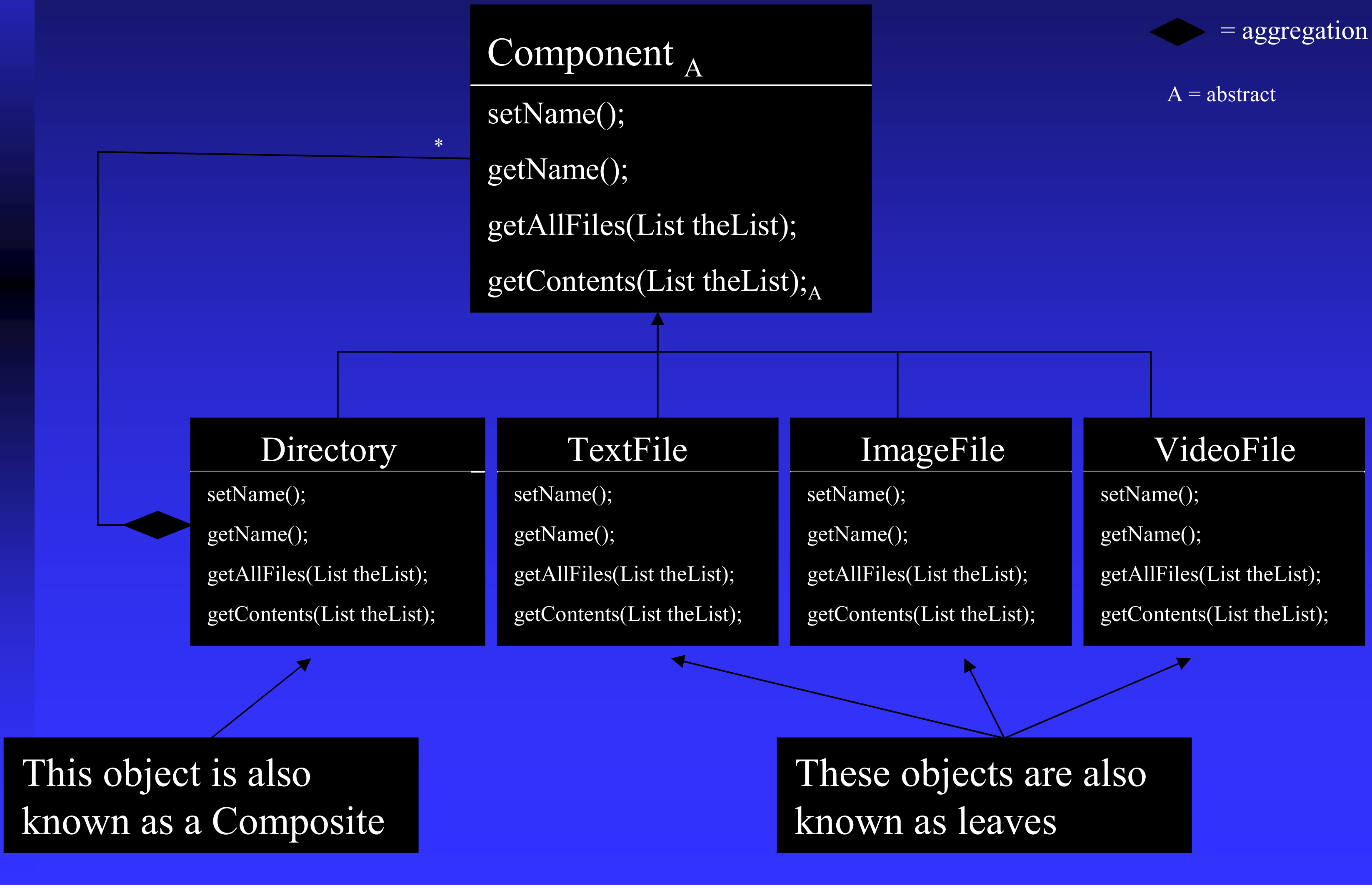
getAllFiles(List theList);

getContents(List theList);

◆ = aggregation

A = abstract

Composite Pattern



Component $_A$

setName();

getName();

getAllFiles(List theList);

getContents(List theList); $_A$

*

Directory

setName();

getName();

getAllFiles(List theList);

getContents(List theList);

TextFile

setName();

getName();

getAllFiles(List theList);

getContents(List theList);

ImageFile

setName();

getName();

getAllFiles(List theList);

getContents(List theList);

VideoFile

setName();

getName();

getAllFiles(List theList);

getContents(List theList);

◆ = aggregation

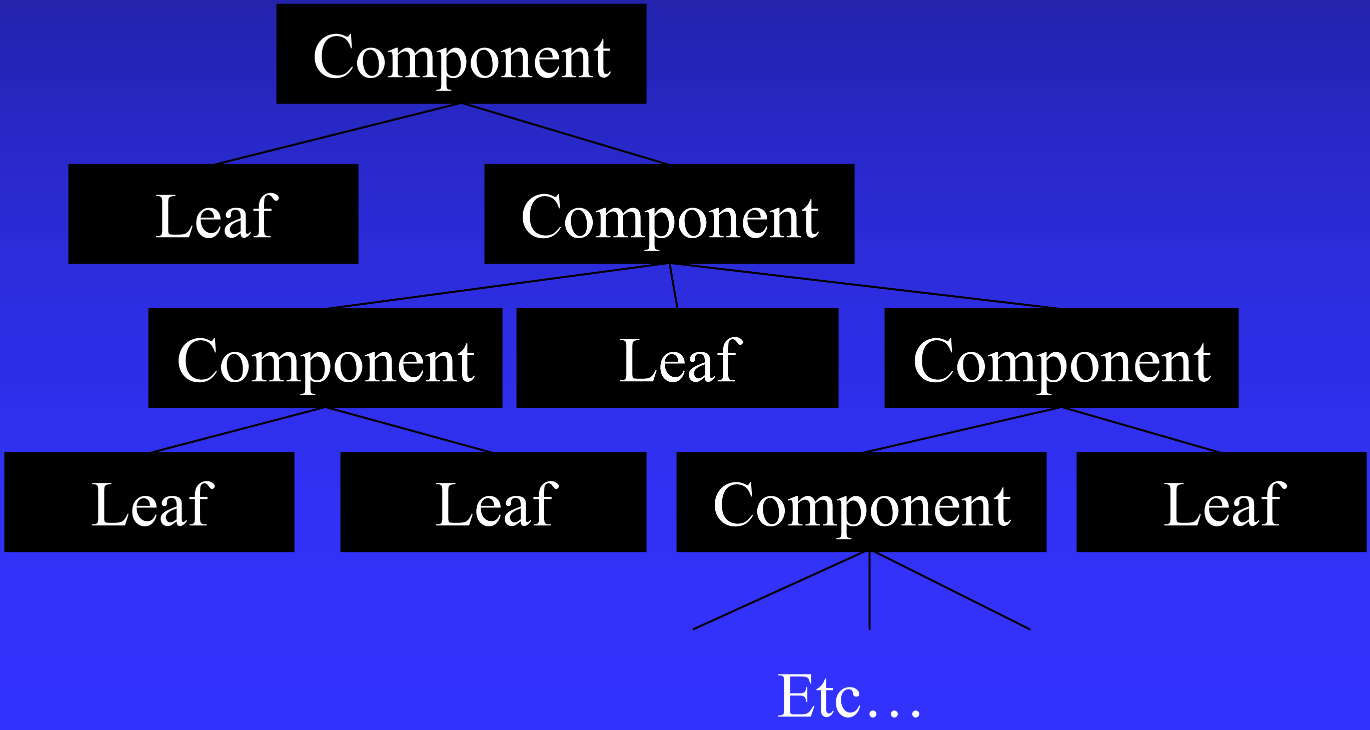
A = abstract

This object is also known as a Composite

These objects are also known as leaves

Composite Pattern

- The component pattern will result in a tree structure.



Component

Leaf Component

Component Leaf Component

Leaf Leaf Component Leaf

Etc…

Component Pattern

```
abstract public class Component {                           // The component is the abstract
      String myName;                                        // class that all other elements will
                                                            // extend
      private void setName(String theName) {
           myName = theName;
      }

      private String getName() {
          return myName;
      }

      private void getAllFiles(List theList) {                // This method will loop through
          for all children {                                  // all of the component's children
                theList.append(child)                         // and add them to the list of files
          }
      }

      abstract private void getContents(List theList);
}
```

Composite Pattern

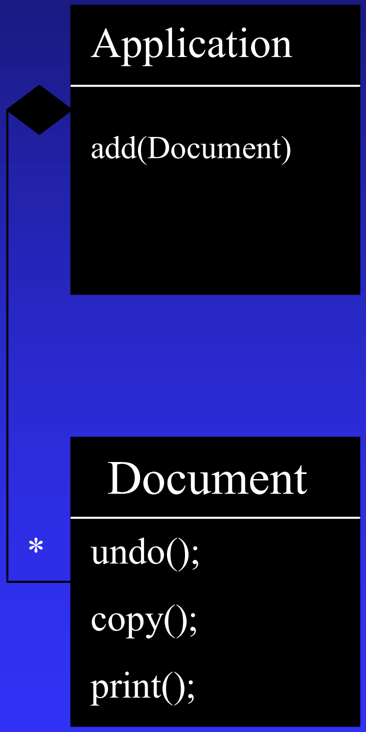
```
public class Directory extends Component {          // The directory is a component
    private void getContents(List theList) {        // Get the contents of this directory
        for all children {
            child.getContents(theList);
        }
    }
}
public class TextFile extends Component {           // This is a leaf element
    private void getContents(List theList) {
        theList.append(this);                       // The leaf node adds itself to the
    }                                               // list its parent’s contents list
}
```

Command Pattern

- Intent:

  Encapsulate a request as an object. This allows action to occur without knowing exactly what request is being made.

Command Pattern



Application

add(Document)

Document

undo();

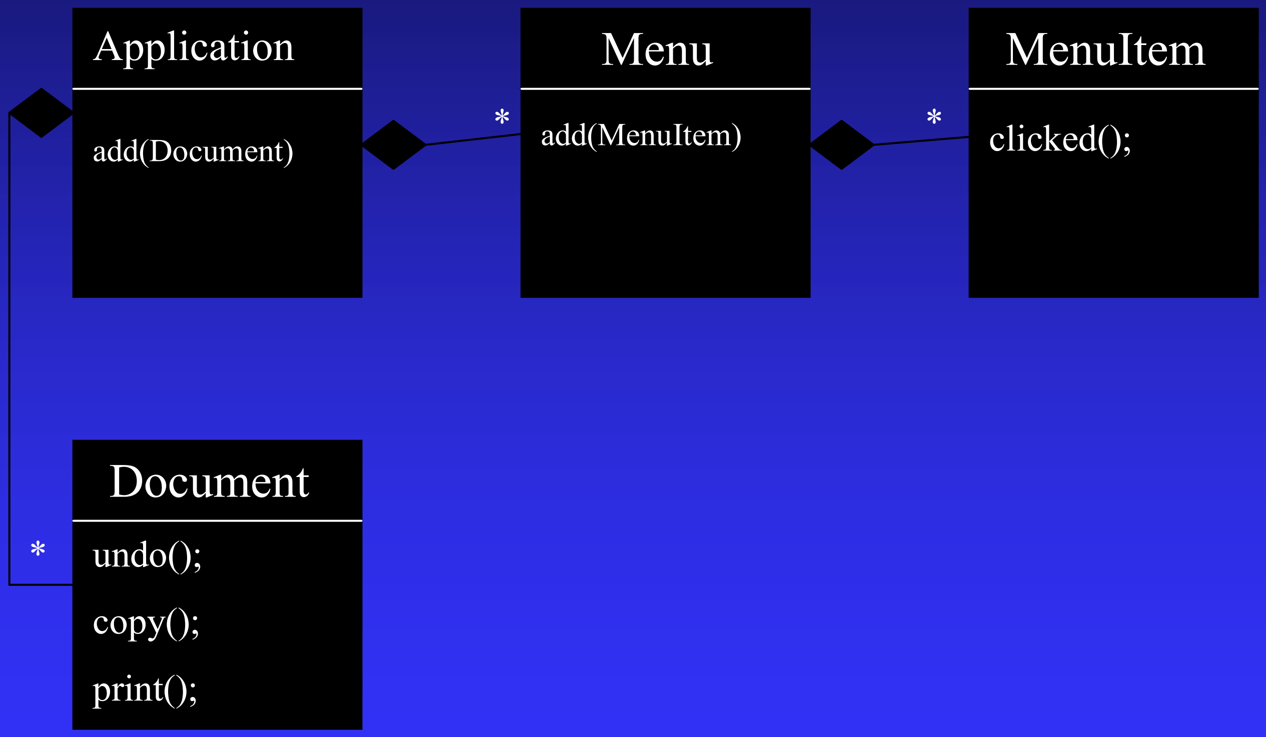
copy();

print();

*

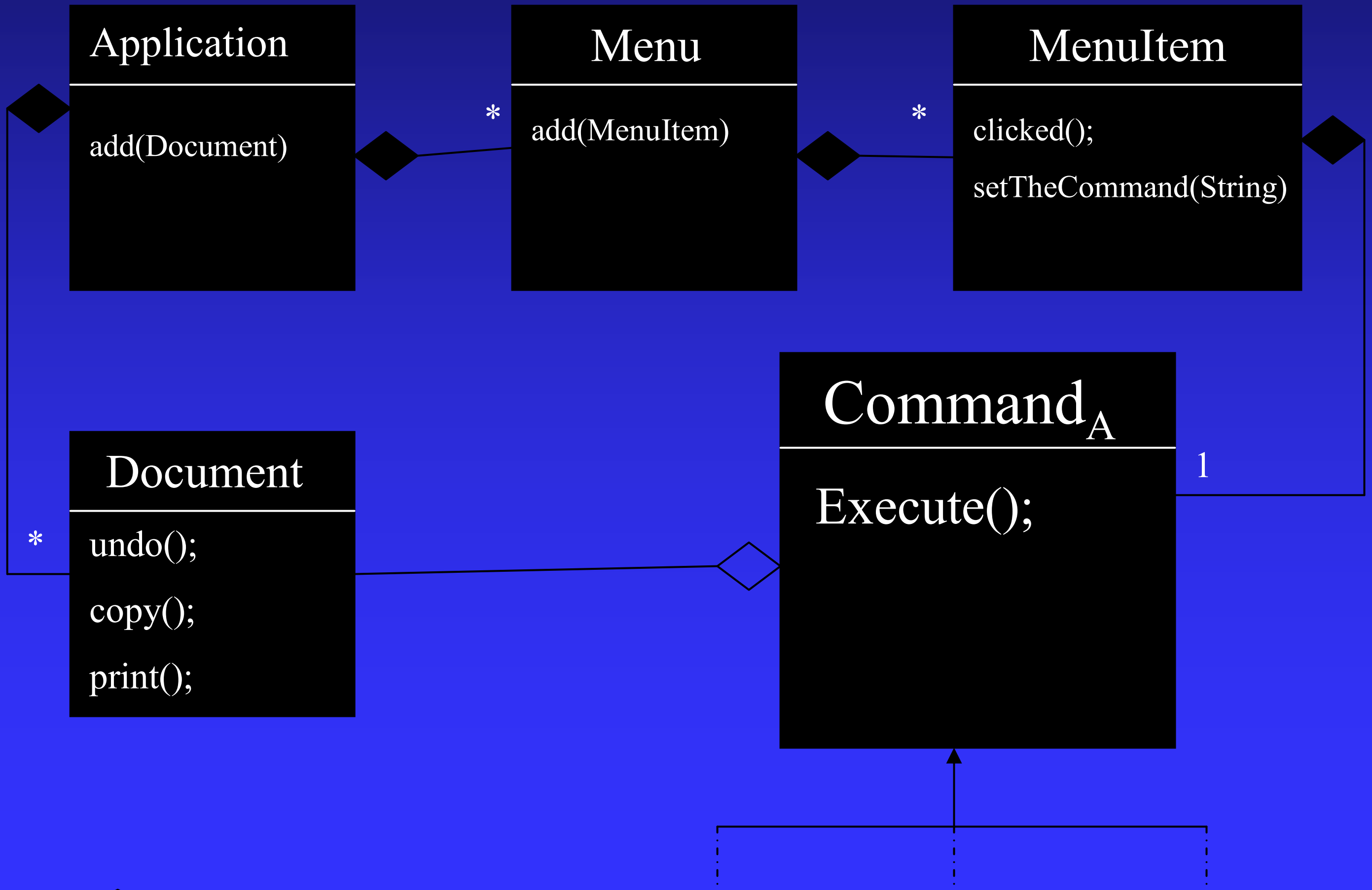
◇ = acquaintance ◆ = aggregation

Command Pattern



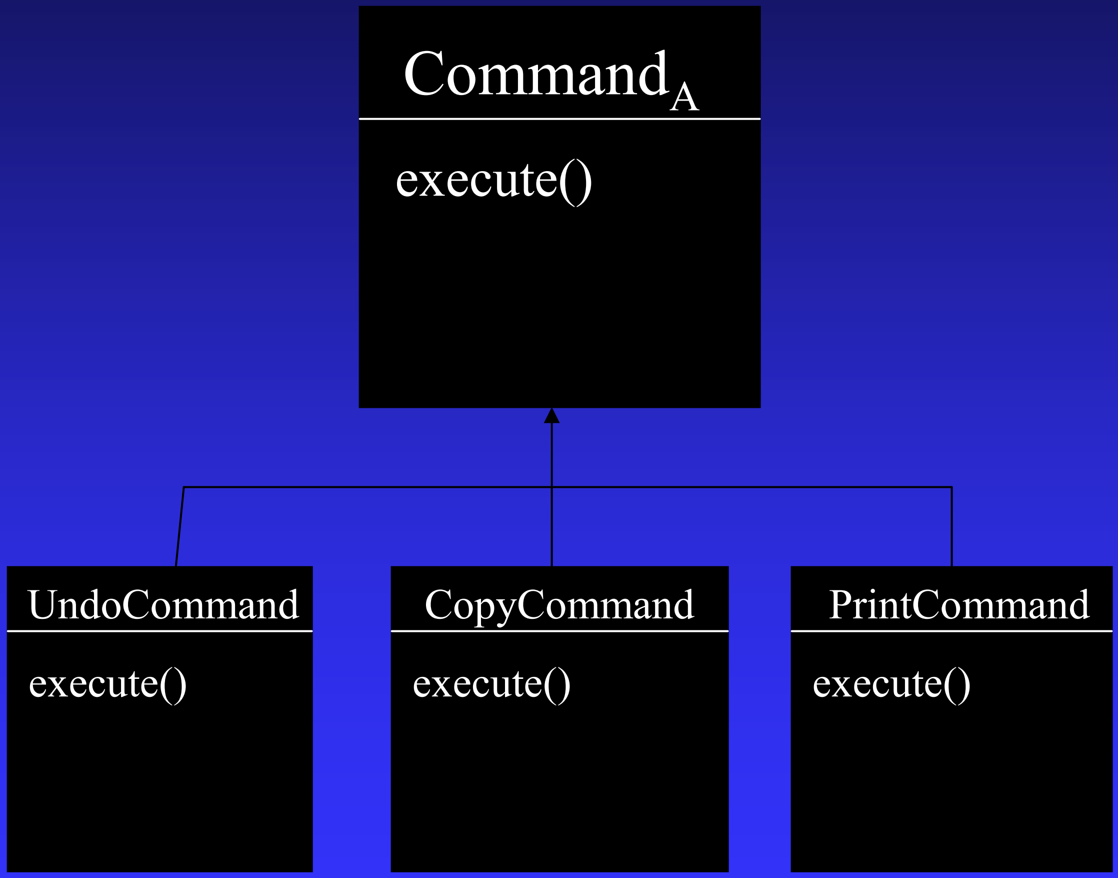
◇ = acquaintance ◆ = aggregation

Command Pattern



◇ = acquaintance ◆ = aggregation A = abstract

Command Pattern



Command$_A$

execute()

UndoCommand

execute()

CopyCommand

execute()

PrintCommand

execute()

A = abstract

Command Pattern

```
public class MenuItem {
    public Command command;

    public void Clicked() {
        command.execute();                        // Simply call execute and
    }                                             // the type of the command
                                         // determines what exactly occurs

    public void setTheCommand(String theCommand) {
        command = theCommand;
    }
}
```

Command Pattern

```
abstract public class Command {
      abstract private void execute();
}

public class UndoCommand {
      document.undo();
}

public class CopyCommand {
      document.copy();
}

public class PrintCommand {
      document.print();
}
```

Strategy Pattern

- Intent:

  Encapsulate a family of algorithms and make them interchangeable. This allows the algorithm to very independently from the clients that will be using it

Mediator

- Intent:

  Define an object that encapsulates how a set of objects interact.

One Final Example

```
// Template
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        super.execute(mapping, form, request, response);
        baseForm.setCommand();   // Factory
        prepareAction(request, baseForm);
        isFormValid = ValidationValidator.isFormValidCritereon(baseForm, getSearchCriterionValidations(baseForm));
        performAction(request, baseForm);
        return baseForm.getCommand().searchActionForward(mapping, baseForm);
}
// Command
Protected void performAction(HttpServletRequest request, BaseForm baseForm) throws Exception {
        this.baseForm.getCommand().performAction(this.baseForm, façade, this, request);
}
//Façade
Public void performAction(BaseForm baseForm, WebFacade facade, BaseAction baseAction, HttpSerbletRequest request)
        throws Exception {
        this.checkForData(baseForm.getHandlesToAction());
        facade.holdSettlements(baseForm.getHandlesToAction());
}
```