Copyright Warning

ViSLAB
High-Performance Visualization, Communication, Computing

Object Oriented Design

Week 10-1

- Generation :
  - Factory Method
  - Abstract Factory
- Structural:
  - Composite
- Behaviour:
  - Template Method
  - Command

Factory Method

- Abstract (Super)class
  - instantiate an object of a concrete class (subclass)

- Example:

```
DataObject *dObject1 = new FileDataObject();

DataObject *dObject1 = new URLDataObject();
```

Factory Method

- Why?
  - During the development....use temp. class
  - later...want to replace it with a concrete class

```
DataObject *dObject1 = new FileDataObject();
```

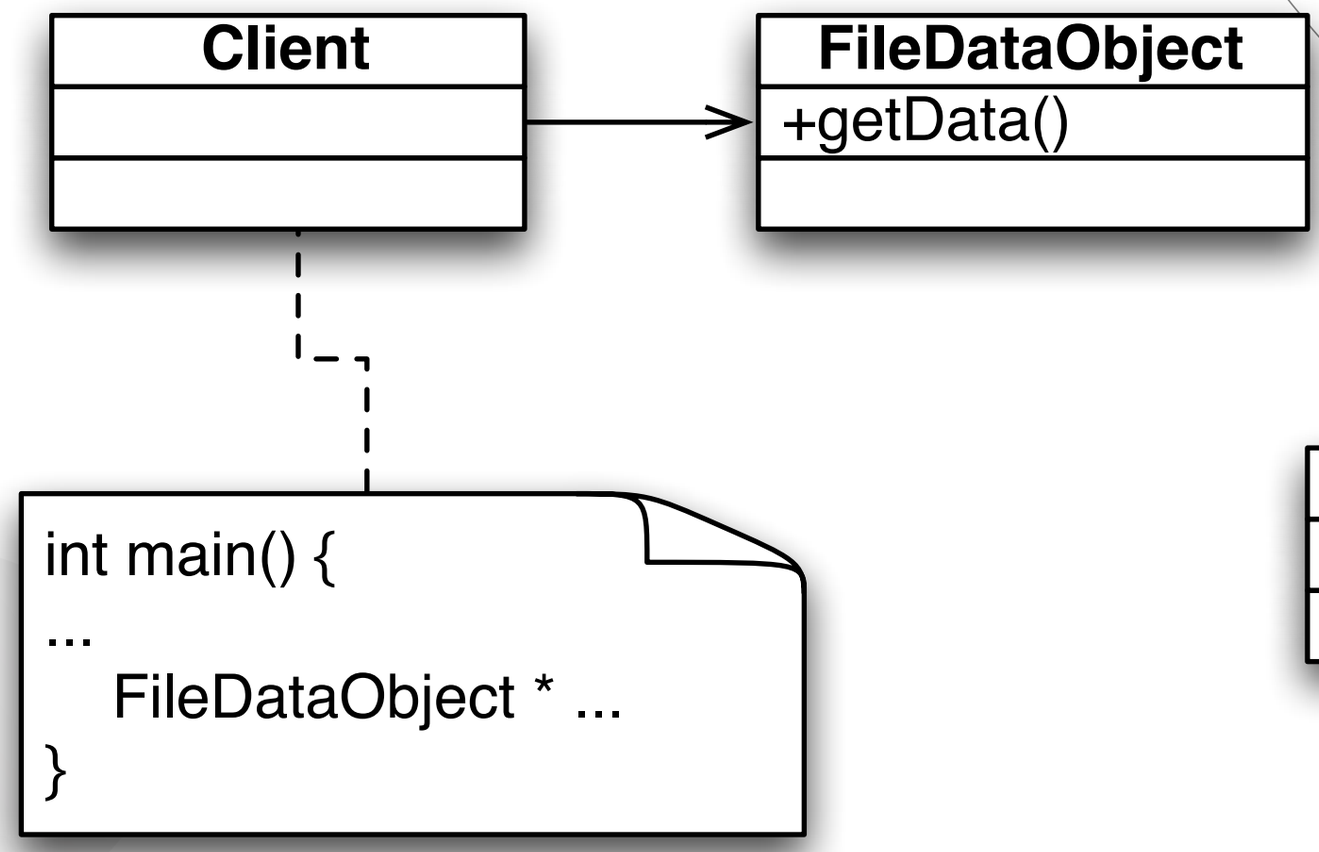
```
DataObject *dObject1 = new URLDataObject();
```

- Factory Method Pattern
  - generate an instance through Abstract (Super) class's method.

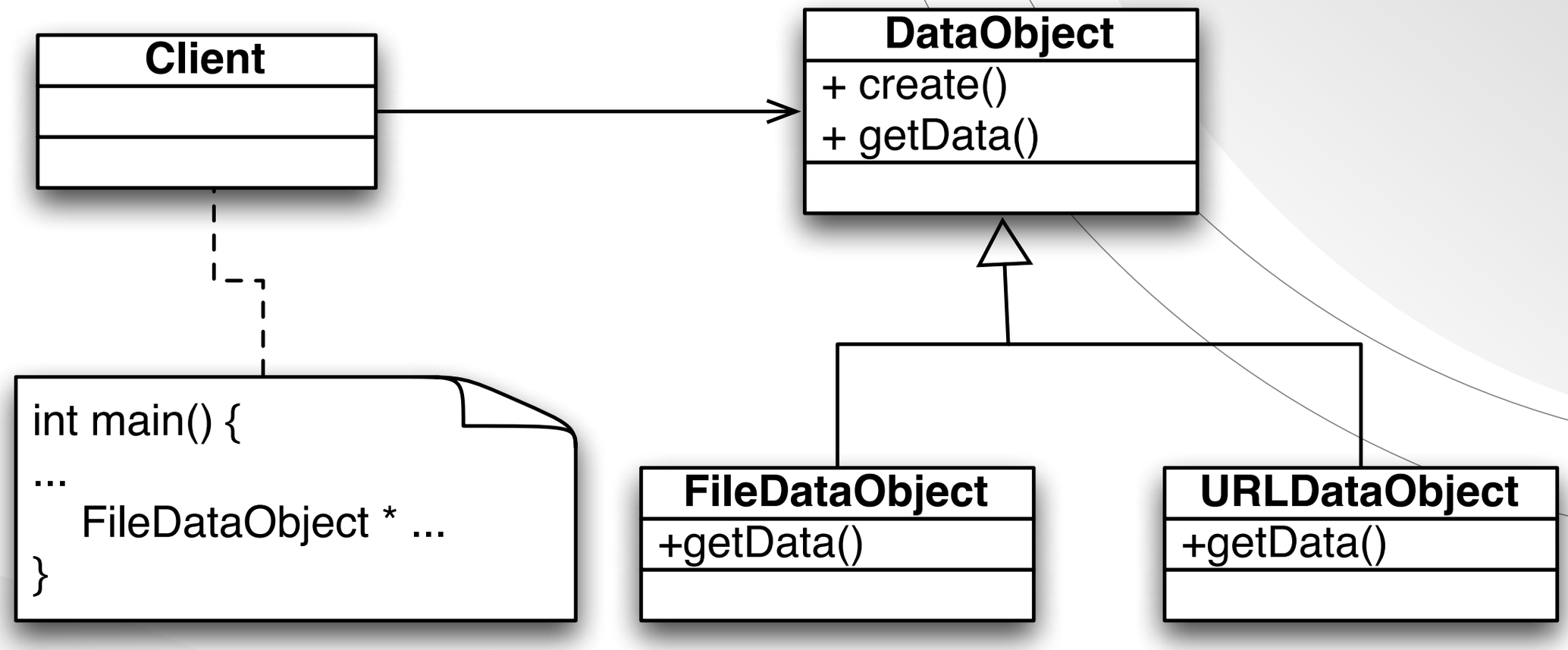
Factory Method

```
class FileDataObject {
public:
    FileDataObject() {
        // read data from a file and store in myData;
    }
    Datum getData(int idx) {
        // retrieve a Datum from myData using idx
    }
private:
    map<int, Datum> myData;
};

int main() {
    ...
    FileDataObject *dataObject = new FileDataObject();
    Datum *data = dataObject->getData(1);
    ...
}
```

Factory Method



Factory Method




```
class DataObject {
public:
   static DataObject* create() {
      return new FileDataObject();
   }
   virtual Datum getData(int idx) = 0;
};
```

```
int main() {
    ...
    FileDataObject *dataObject = new FileDataObject();
    Datum *data = dataObject->getData(1);
    ...
}

int main() {
    ...
    DataObject *dataObject = new DataObject::create();
    Datum *data = dataObject->getData(1);
    ...
}
```

- Merit
  - no need to change code using the DataObject classes

Abstract Factory

- A class provide functions to create
  - related objects without calling individual's constructor

- Example:
  - GUI components:
    - Window
    - Button
    - Textfield

Abstract Factory

- Example:
  - GUI components:
    - Window, Button, Textfield, etc.
  - GUI using
    - Windows GUI
    - OS X
    - GNOME, etc.

Abstract Factory

- Example:
  - GUI using
    - Windows GUI

      `WinWindow`
      `WinButton`
      `WinTextField`

    - OS X

      `MacWindow`
      `MacButton`
      `MacTextField`

Abstract Factory

- Example:
  - WinButton

```
class WinButton {
    ...
    void setStatus(int s) {
        winButton.showMenu(s);
        ...
        return;
    }
    ...
};
```

Abstract Factory

```
class MyGUIFactory {
public:
   WinWindow* createWinWindow() {
      return new WinWindow();
   }
   WinButton* createWinButton() {
      return new WinButton();
   }
   MacWindow* createMacWindow() {
      return new MacWindow();
   }
   MacButton* createMacButton() {
      return new MacButton();
   }
};
```

Abstract Factory

```
int main() {
MyGUIFactory myGUIfactory;
WinWindow* pWin_window;
WinButton* pWin_button;
MacWindow* pMac_window;
MacWindow* pMac_button;

int gui_type = 0; // 0: windows, 1: mac style

   switch (gui_type) {
   case 0:
      pWin_window = myGUIfactory.createWinWindow();
      pWin_button = myGUIfactory.createWinButton();
   case 1:
      pMac_window = myGUIfactory.createMacWindow();
      pMac_button = myGUIfactory.createMacButton();
   }
}
```

Abstract Factory

- Applying the pattern:

```
class MyWindow {
public:
    ...
};

class MyButton {
public:
    ...
};
```

```
class WinWindow : public MyWindow {
public:
    ...
};

class WinButton : public MyButton {
public:
    ...
};

class MacWindow : public MyWindow {
public:
    ...
};

class MacButton : public MyButton {
public:
    ...
};
```

Abstract Factory

```
class MyGUIFactory {
public:
    virtual MyWindow* createWindow() = 0;
    virtual MyButton* createButton() = 0;
};

class WinGUIFactory : public MyGUIFactory {
public:
    MyWindow* createWindow() {
        return new WinWindow();
    }
    MyButton* createButton() {
        return new WinButton();
    }
};

class MacGUIFactory : public MyGUIFactory {
public:
    MyWindow* createWindow() {
        return new MacWindow();
    }
    MyButton* createButton() {
        return new MacButton();
    }
};
```

Abstract Factory

```
class MyGUIFactory {
public:
    virtual MyWindow* createWindow() = 0;
    virtual MyButton* createButton() = 0;
};

class WinGUIFactory : public MyGUIFactory {
public:
    MyWindow* createWindow() {
        return new WinWindow();
    }
    MyButton* createButton() {
        return new WinButton();
    }
};

class MacGUIFactory : public MyGUIFactory {
public:
    MyWindow* createWindow() {
        return new MacWindow();
    }
    MyButton* createButton() {
        return new MacButton();
    }
};
```

```
int main() {
    MyGUIFactory pGUIfactory;
    MyWindow* pWin;
    MyButton* pButton;

    pGUIfactory = new WinGUIFactory();

    pWin = pGUIfactory->createWindow();
    pButton = pGUIfactory->createButton();
    ...
}
```

Abstract Factory

- Merit
  - class user does not need to know the details of concrete classes.
  - can group relevant classes and their generation
  - easy to switch the groups

Composite

- easy to deal with a tree-like structure
- Example:

```
class SceneManager {
protected:
   list<Scene> m_SceneArray;

public:
   void addScene(Scene *);
   void removeScene(Scene *);
};
```

- change Scenes
- need manager to manage SceneManagers?

Composite

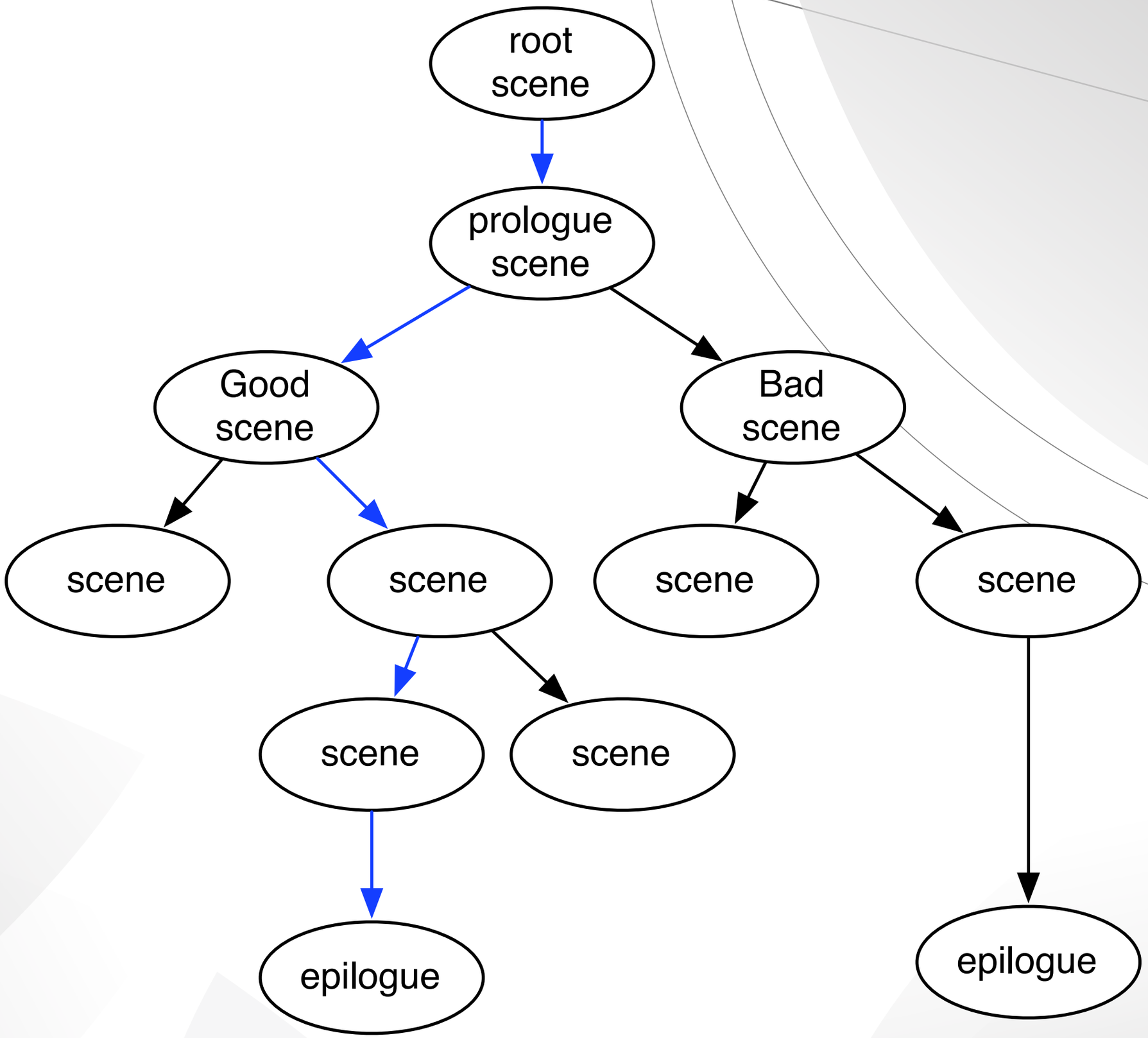
- SceneManager != Scene

- Composite Pattern
  - SceneManager == Scene
  - SceneManager contains Sciene

Composite

- SceneManager == Scene

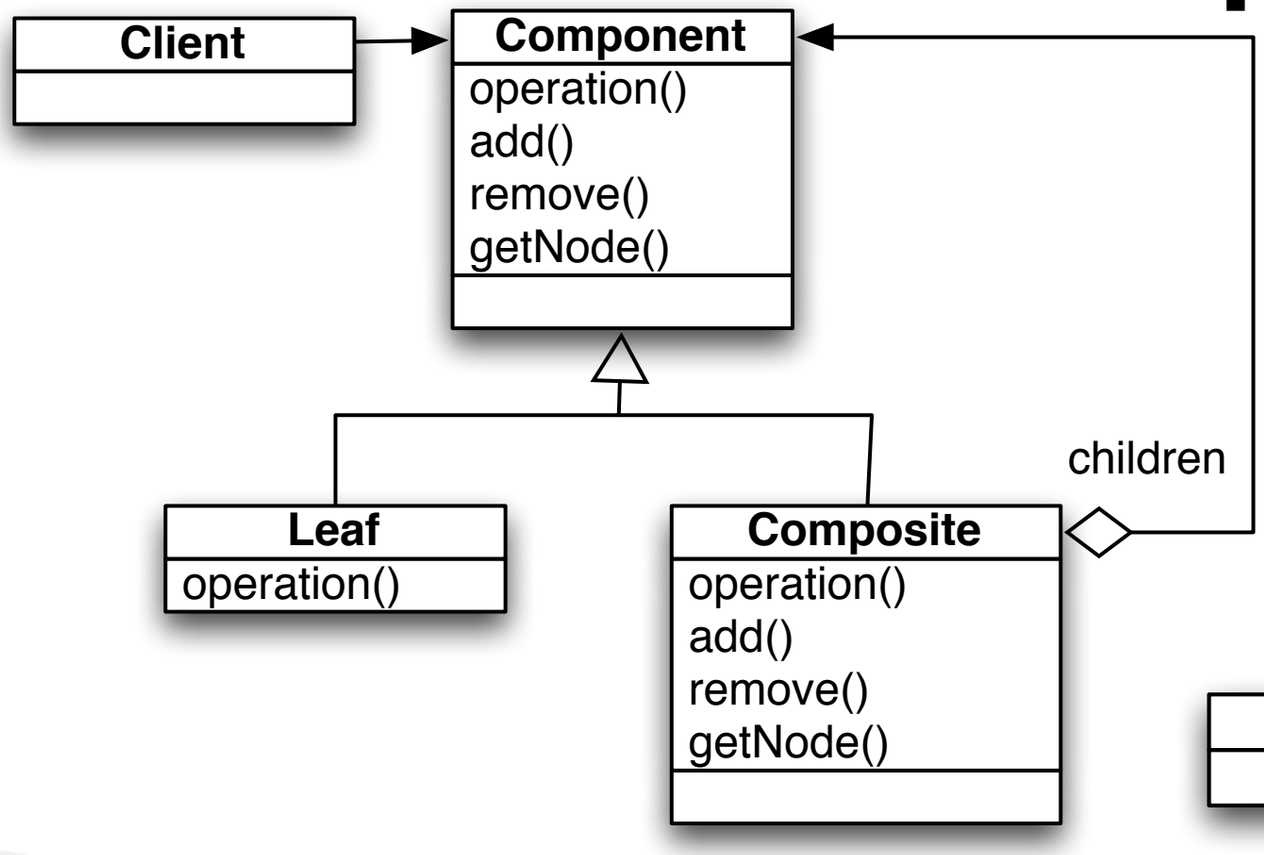
```
class Scene {
private:
   list<Scene*>_SceneArray;
protected:
   Scene* _pParent;
public:
   virtual void addScene(Scene*);
   virtual void removeScene(Scene*);
   virtual Scene* getChild(int idx) {
      if (_SceneArray.size() < idx)
         return NULL;
      return _SceneArray[idx];
   }

   virtual void begin(); // start the scene
};
```

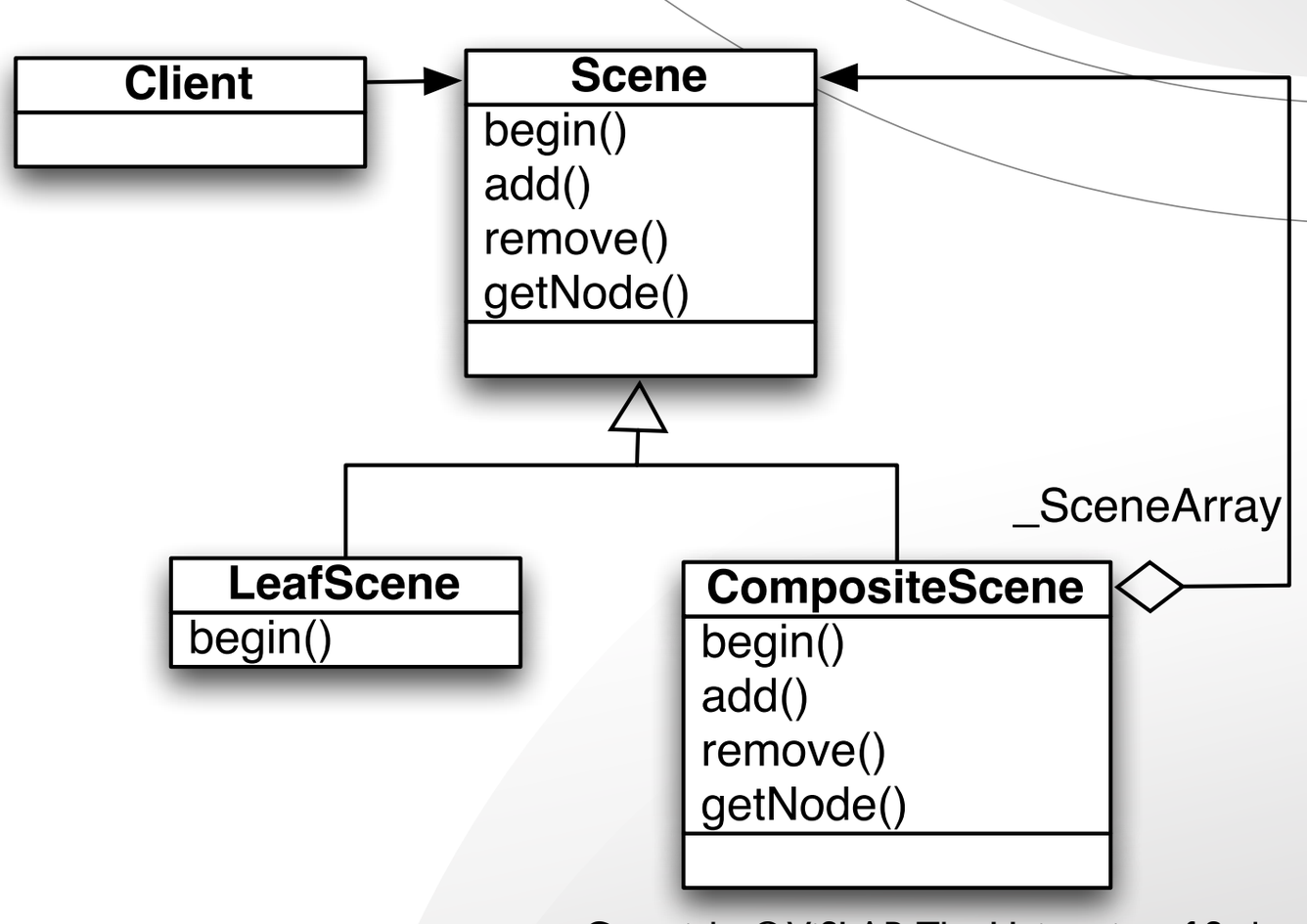


Composite

- SceneManager == Scene
- Scene
- node
- leaf node (no children)

Composite





Composite

- Merit:
- common class/functions for
  - node and container
- easy grouping
- easy addition of new classes

Template Method

- allow you to customise
  - each task
  - in a series of tasks
- Example:
  - Sound player in a Game Program
    - select a music file
    - analyse the file formant
    - launch a monitoring thread
    - play the music

Template Method

- Example:
  - Sound player in a Game Program

```
class SoundPlayer {
private:
    char* _pSoundData; // binary data
    int _size;
    char* _filename;

    virtual bool openFile(char* file);
    virtual bool analyseFile();
    virtual bool launchObserverThread();
    virtual bool playMusic();
public:
    bool play(char *file) {
        if (openFile(file)) return false;
        if (analyseFile())  return false;
        if (launchObserverThread()) return false;
        if (playMusic())    return false;
    }
};
```

Template Method

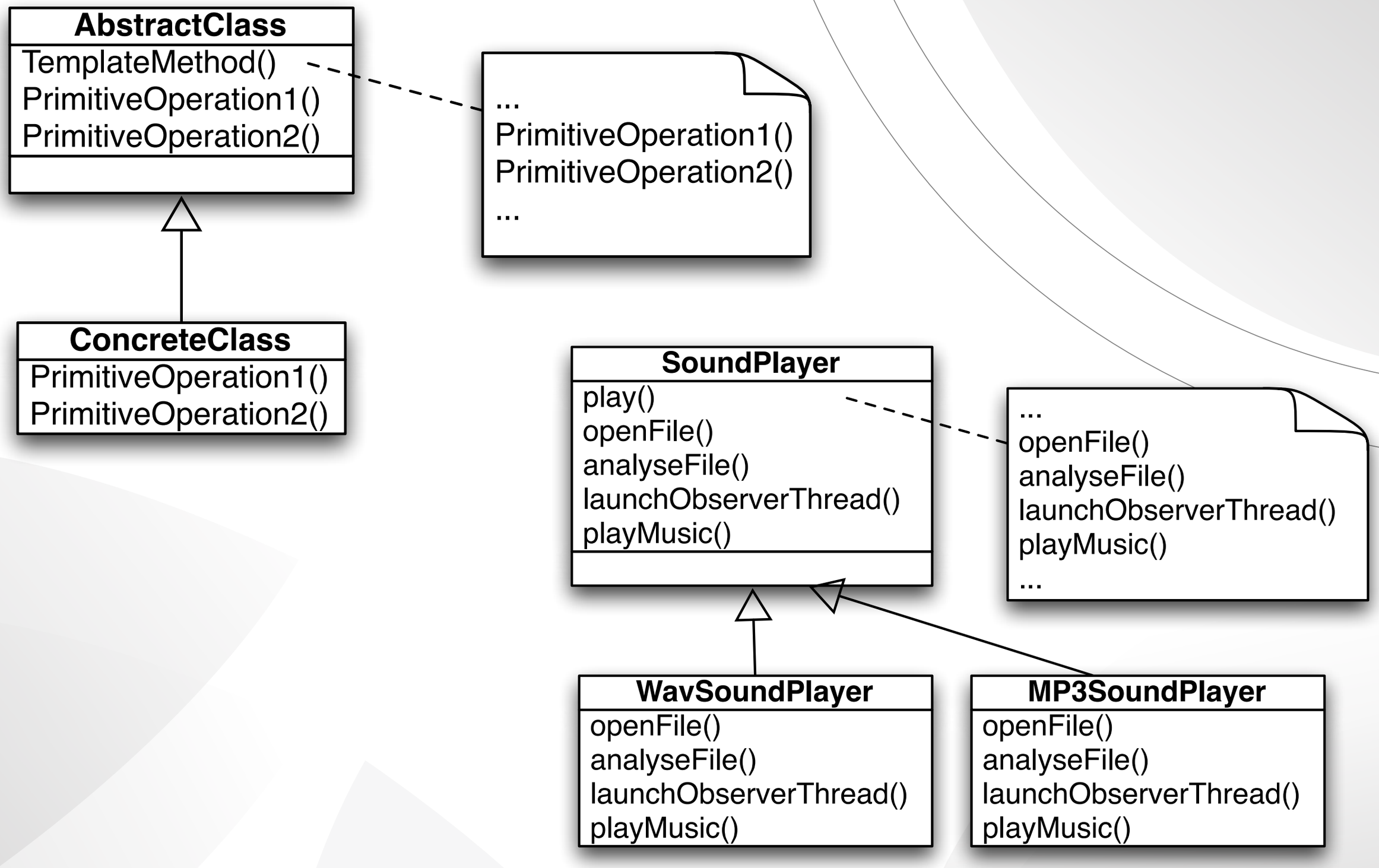
Template Method

- Example:
  - SoundPlayer and its subclass WavPlayer, MP3Player can implement their own operations (openFile, analyseFile, launchObserverThread, playMusic)

```
class SoundPlayer {
private:
...
    virtual bool openFile(char* file);
    virtual bool analyseFile();
    virtual bool launchObserverThread();
    virtual bool playMusic();
public:
    bool play(char *file) {
        if (openFile(file)) return false;
        if (analyseFile())  return false;
        if (launchObserverThread()) return false;
        if (playMusic())    return false;
    }
};
```

Template Method

Template Method



Template Method

- Merit:
- being able to define clear sequence of tasks
- clear customisation point

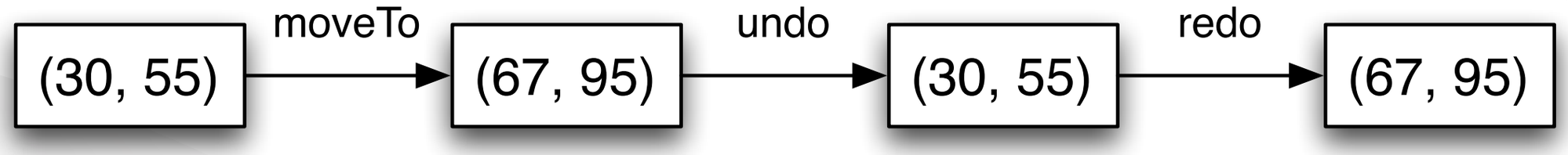
Command

- task == command
- regardless of tasks
  - when you know the timing of execution
  - when you need to control the timing of execution

- Example:
  - Simple Graphics (drawing) system

Command

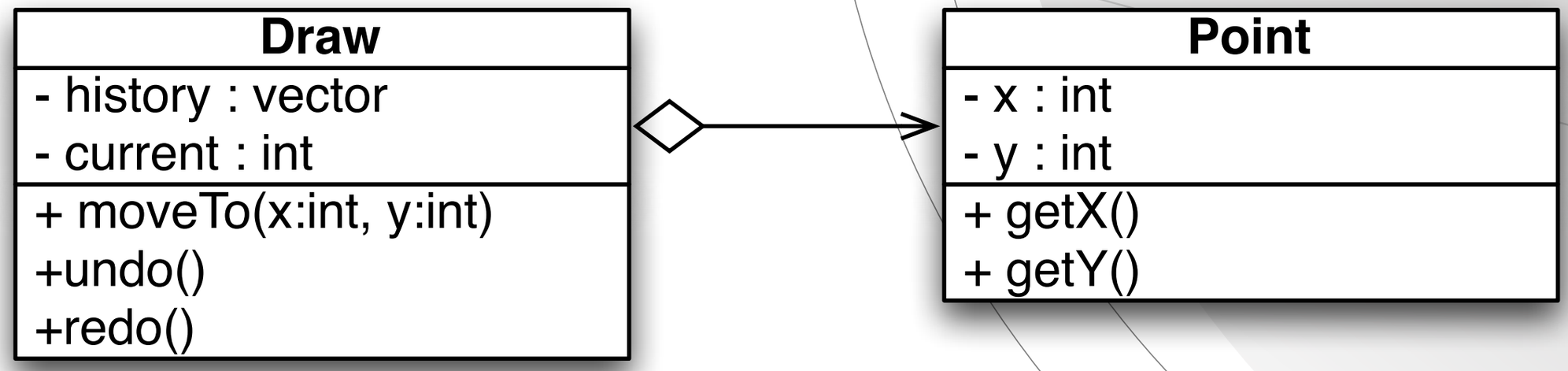
- Example:
  - Simple Graphics (drawing) system



- move a pen-tip to draw an image.

```
class Point {
    int x, y;
public:
    Point() : x(0), y(0) {}
    Point(int x, int y) :
x(x), y(y){}
    int getX() {return x;}
    int getY() {return y;}
};
```

```
class Draw {
    vector<Point*>history;
    int current;
public:
    Draw() : current(0) {
        history.push_back(new Point());
    }
    ~Draw() {
        vector<Point*>::iterator it = history.begin();
        while (it != history.end()) {
            delete (*it);
            it++;
        }
        history.clear();
    }
    void moveTo(int x, int y) {
        if (reduable()) {
            vector<Point*>::iterator it = history.begin();
            it += current +1;
            history.erase(it, history.end());
        }
        history.push_back(new Point(x, y));
        current++;
    }
    bool reduable() {
        return (current < (int)history.size() - 1) ? true : false;
    }
    void undo() { if (current > 0) current--;}
    void redo() { if (reduable()) current++;}
};
```

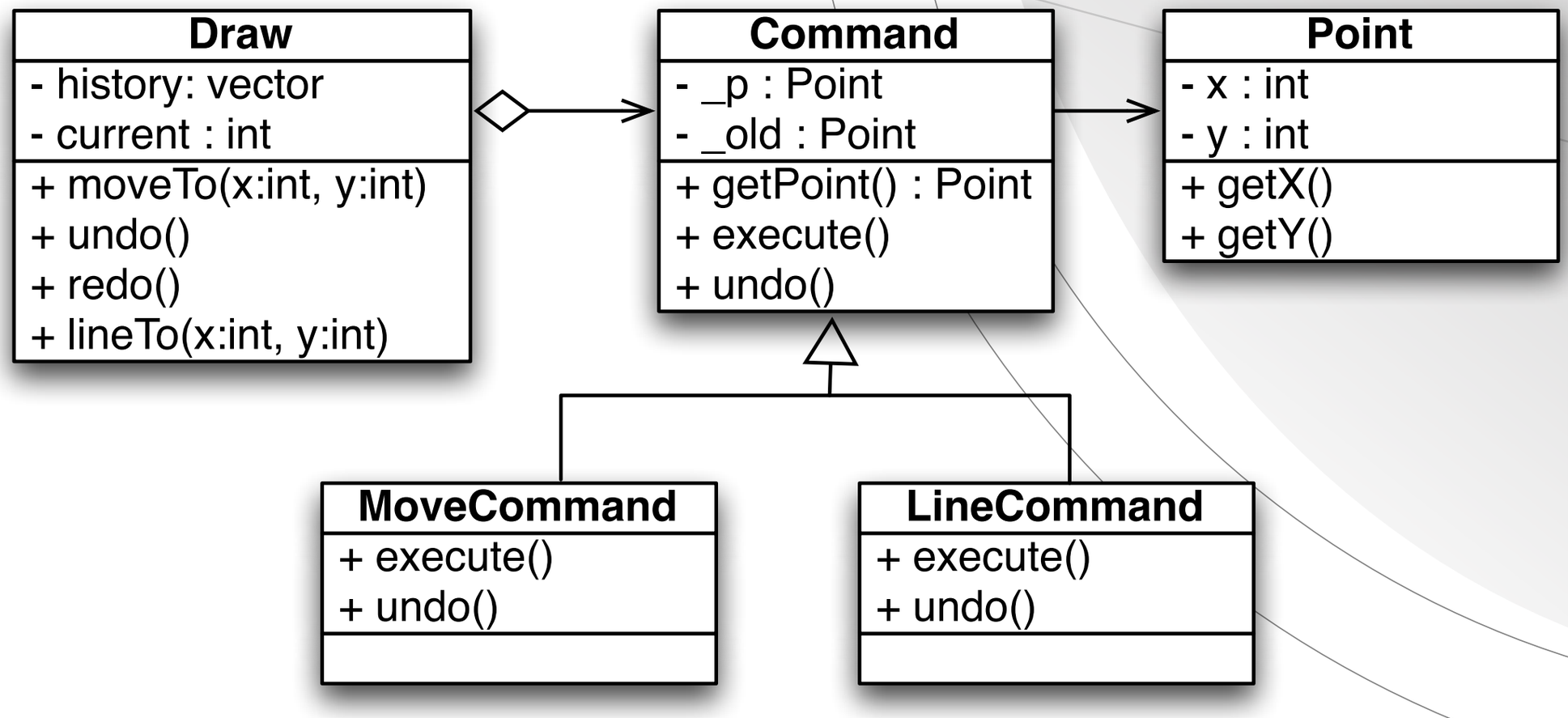


- Apply “Command” pattern
  - moveTo : MoveCommand
  - undo/redo : manipulation of index of the current command in the list
  - (new) drawLine : LineCommand

```
class Draw {
    vector<Command*>history;
    int current;
    void addHistory(Command* cmd) {
        if (redoable()) {
            vector<Command*>::iterator it = history.begin();
            it += current + 1;
            history.erase(it, history.end());
        }
        history.push_back(command);
        current++;
        command->execute();
    }
public:
    Draw() : current(0) {
        history.push_back(new MoveCommand(new Point(0,0),
                                          new Point(0,0)));
}
~Draw() {
    vector<Command*>::iterator it = history.begin();
    while (it != history.end()) {
        delete (*it);
        it++;
    }
    history.clear();
}
void moveTo(int x, int y) {
    addHistory(new MoveCommand(new Point(x, y),
                               getCurrentCommand()->getPoint()));
}
```

```
bool redoable() {
    return (current < (int)history.size() - 1) ? true : false;
}
void undo() {
    if (current > 0) {
        getCurrentCommand()->undo();
        current--;
    }
}
void redo() {
    if (reduable()) {
        current++;
        getCurrentCommand()->execute();
    }
}
Command* getCurrentCommand() {
    vector<Command*>::iterator it = history.begin();
    it += current;
    return (*it);
}

void lineTo(int x, int y) {
    addHistory(new LineCommand(new Point(x, y), getCurrentCommand()->getPoint()));
}
};
```

```
class Command {
    Point _p;
    Point _old;
public:
    Command(Point* p, Point* old) : _p(p), _old(old) {}
    ~Command() { delete _p;}
    virtual void execute() = 0;
    virtual void undo() = 0;
    virtual Point* getPoint() {return _p;}
};

class MoveCommand : public Command {
public:
    MoveCommand(Point * p, Point* old) : Command(p, old) {}
    void execute() {/* move the cursor to _p*/}
    void undo() { /* move _old to _p */}
};

class LineCommand : public Command {
public:
    LineCommand(Point* p, Point* old) : Command(point, old) {}
    void execute() { /* draw a line from _old to _p*/}
    void undo() {/* delete a line from _old to _p */}
};
```



- Merit:
  - easy to combine different commands
  - easy to add new commands
  - support a queue
  - support undo