Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Programming-2 CS213

Abstractions

Dr. Amin Allam

1 Abstract classes

In the Game and Hero example, the Hero class acts as a placeholder, that is, we never create objects of type Hero. Although creating objects of type Hero is syntactically valid, it is semantically meaningless since it does not represent a particular hero that can be used in the game. We use a Hero* only to hold the address of an object from a derived class such as SuperMan, but we never used it to hold the address of an object of type Hero.

It is possible to put some semantics in class Hero to have the definition of some kind of default hero, and create objects of it to be actually used in the game. However, whenever we need to modify or enhance class Hero, we will encounter the problems we discussed in our second attempt in the polymorphism lecture (basically we will need to change the definition of class Game to contain a pointer to the enhanced Hero type). Therefore, if we need a default hero, it is usually better to derive it from class Hero as well.

In this case, to prevent the programmer from creating objects of type Hero by mistake, and to make the program more readable, we should provide at least one pure virtual function in the class. A pure virtual function does not have an implementation and has the characters =0 after its declaration. A class containing a pure virtual function is called abstract class and can not be instantiated (we can not create objects of its type). A class which does not contain any pure virtual function is called a concrete class and can be instantiated.

```
class Hero            // Abstract class
{
protected:
    int armor;
public:
    virtual void Fire() {armor--;}
    virtual void Jump() {}
    virtual void Draw()=0;      // Pure virtual function, does not have a definition
    virtual ~Hero() {}
};
```

2 Interfaces

An abstract class may contain some data members (such as int armor) and some minimal function definitions which are very difficult to change (such as Fire() function above).

However, since data and function definitions are usually subject to change, it is better to avoid them completely by making all functions pure virtual. An interface is a class which does not contain any data members, and all its functions are pure virtual except the constructors which are not virtual and the destructor which is virtual but not pure.

```
class Hero           // Interface
{
public:
    Hero() {}
    virtual void Fire()=0;
    virtual void Jump()=0;
    virtual void Draw()=0;
    virtual ~Hero() {}
};
```

The above discussion leads to the following principle:

3 The dependency inversion principle

The dependency inversion principle is one of the SOLID principles stated as follows:

"High level modules should not depend upon low level modules. Both should depend upon abstractions." $\Rightarrow$ "Program to an interface, not an implementation."

The high level class `Game` depends on abstraction of Hero (the `Hero` interface), instead of the heroes themselves (such as `class SuperMan`).

Another example, consider a high level module `Copy()` depends on two low level modules: `ReadCharFromKeyboard()` and `WriteCharToPrinter()`.

```
void Copy()
{
    while(true)
    {char c=ReadCharFromKeyboard(); WriteCharToPrinter(c);}
}
```

Suppose we need to do the same behavior, but to copy from a file to screen, instead of copying from keyboard to printer. We must define another function that has very similar behavior of `Copy()` but depends on different low level modules:

```
void Copy2()
{
    while(true)
    {char c=ReadCharFromFile(); WriteCharToScreen(c);}
}
```

This caused duplicating the code of the original `Copy()` and replacing the inside function calls by new function calls. We can avoid this code duplication by making the `Copy()` function depends upon interfaces instead of low level function calls as follows:

```
class Reader
{
public:
    virtual char ReadChar()=0;
    virtual ~Reader() {}
};
class KeyboardReader : public Reader
{
public: char ReadChar() {/*Read char from keyboard*/}
};
class FileReader : public Reader
{
public: char ReadChar() {/*Read char from file*/}
};

class Writer
{
public:
    virtual void WriteChar(char c)=0;
    virtual ~Writer() {}
};
class PrinterWriter : public Writer
{
public: void WriteChar(char c) {/*Write char to printer*/}
};
class ScreenWriter : public Writer
{
public: void WriteChar(char c) {/*Write char to screen*/}
};

void Copy(Reader* r, Writer* w)
{
    while(true) {char c=r->ReadChar(); w->WriteChar(c);}
}
void TestCopy()
{
    Reader* r=new KeyboardReader; Writer* w=new PrinterWriter;
    Copy(r, w);
    delete rk; delete wp;
    Reader* rf=new FileReader; Writer* ws=new ScreenWriter;
    Copy(rf, ws);
    delete rf; delete ws;
}
```

Alternatively, the function `Copy()` can be defines as follows:

```
void Copy(Reader& r, Writer& w)
{
    while(true) {char c=r.ReadChar(); w.WriteChar(c);}
}
void TestCopy()
{
    KeyboardReader rk; PrinterWriter wp;
    Copy(rk, wp);
    FileReader rf; ScreenWriter ws;
    Copy(rf, ws);
}
```

It is also possible to use pointers to functions instead of interfaces as follows (which can be done in the C language as well as C++):

```
char ReadCharFromKeyboard() {/*Read char from keyboard*/}
char ReadCharFromFile() {/*Read char from file*/}
void WriteCharToPrinter(char c) {/*Write char to printer*/}
void WriteCharToScreen(char c) {/*Write char to screen*/}

void Copy(char (*ReadChar)(), void (*WriteChar)(char))
{
    while(true) {char c=ReadChar(); WriteChar(c);}
}
void TestCopy()
{
    Copy(ReadCharFromKeyboard, WriteCharToPrinter);
    Copy(ReadCharFromFile, WriteCharToScreen);
}
```

4 The single responsibility principle

The single responsibility principle is one of the SOLID principles stated as follows:

"A module should have only one reason to change."

The `Copy()` function definition needs to change only if the behavior (or logic) of copying changes. It does not need to change if the reader (source) or writer (destination) changes.

Similarly, the definition of `class Game` needs to change only if the behavior (or logic) of playing the game changes. It does not need to change if the Hero changes. The game can support a new Hero easily by just deriving it from `class Hero` and passing a pointer to its object to the `Game` constructor.

5 The interface segregation principle

The interface segregation principle is one of the SOLID principles stated as follows:

"A module should not depend upon an interface which does not use."

Given the declaration of the following interfaces:

```
class Button
{
public:
    virtual void Click()=0;
    virtual void DrawNormal()=0;
    virtual void DrawClicked()=0;
    virtual ~Button() {}
};

class Animation
{
public:
    virtual void Animate()=0;
    virtual ~Animation() {}
};
```

`class Button` represents clickable buttons used on graphical user interfaces, while `class Animation` represents an area on screen where animation is played. Both interfaces are useful and independently used by other modules.

Suppose that we need to create a button which contains animation. A first attempt is to redefine `class Button` as follows:

```
class Button : public Animation
{
public:
    virtual void Click()=0;
    virtual void DrawNormal()=0;
    virtual void DrawClicked()=0;
    virtual ~Button() {}
};
```

The above approach contains a significant drawback. `class Button` now depends upon `class Animation` regardless of whether the button actually has animation. All modules that need non-animated buttons are forced to depend upon animated buttons.
This reduces the readability of the programs, introduces additional complexity by incorporating unused interfaces in applications which do not need them, and gives more chances to misuse `class Button` if some user calls `Animate()` on a non-animated button.

A better solution is not to change the original `Button` or `Animation` interfaces, and declare a new interface as follows:

```
class AnimatedButton : public Button, public Animation
{
public:
    virtual ~AnimatedButton() {}
};
```

This approach has the following problem: it introduces a dependency between Button and Animation which should not exist. Suppose we have two concrete classes SkyAmination and WaterAnimation derived from Animation. There is no way to use these concrete classes in class AnimatedButton

A better solution is to use aggregation as the following:

```
class AnimatedButton : public Button
{
protected:
    Animation* animation;
public:
    AnimatedButton(Animation* a) {animation=a;}
    virtual void Animate() {animation->Animate();}
    virtual ~AnimatedButton() {}
};
```

We can pass any concrete class derived from Animation to the constructor of AnimatedButton.

Note that class AnimatedButton is still an abstract class since it inherits non-implemented pure virtual functions from class Button. This solution has the advantage of being close to the is-a and has-a rule: An animated button is a button, so it is derived from class Button. An animated button has an animation, so it aggregates an Animation object and delegates the animation task to it.

Although class AnimatedButton is an abstract class (not interface) because it contains a member variable and an implemented method, it has the same properties of interfaces because it does not contain state (the member variable is actually a stateless interface) and the only implemented function delegates its task to an interface.

Aggregation is usually preferred to inheritance, since it gives more flexibility to change the type of aggregated objects (as we did in Game and Hero), keeps each class encapsulated and focused on one task, and keeps class hierarchies simple. This leads to the following principle:

“Prefer object aggregation over class inheritance.”

Now, should AnimatedButton aggregate Button as well? This has the advantage of using any concrete class derived from class Button, such as WoodButton and MetalicButton. However, since class AnimatedButton is no more inherited from class Button, it can not be used in place of Button pointers or references in modules which already use them.