Object-Oriented Progrmming

Mikesc Huang

What is OO ?

What is OO?

- 有人說 OO 是 “The combination of data and function”
- 以這些人的觀點來看，o.f() 與 f(o) 是不同的，但光以這句話來看，o.f() 與 f(o) 是相同的，有矛盾
- 在 1966 年第一個 OO 語言被發明之前，就已經有人把 data structure 當作參數傳入 function，但卻沒有被稱作 OO，表示這句話無法明確定義 OO

What is OO?

- 又有人說 OO 是 “A way to model the real world”
- 雖然 OO 的概念比較貼近真實世界是沒錯，但這樣的說法太模糊，我們還是不知道什麼是 OO

What is OO?

- 而有些人用這三個字來解釋 OO
  - Encapsulation
  - Inheritance
  - Polymorphism
- 也就是我們常聽到的封裝，繼承，多型
- 我們可以說 OO 是封裝，繼承，多型的綜合體，或是說至少一個 OO 語言必須支援封裝，繼承，多型

Encapsulation
封裝

Encapsulation

- 封裝是以 data 為核心，將相關的 data 放在一起，把會用到這些 data 的 function 也放進來，等於將 data 與 function 放在一起
- 但其他 class 或 module 看不到這些 data，且只有部分 function 可以使用
- 舉個最常見的例子，private data member 必須透過 public member function 才能存取

Encapsulation

- 不是只有 OO 才有封裝，其實 C 就有了

point.h

```
struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);
```

- 程式開發者可以使用 makePoint(), distance()，但卻無法存取 Point 的 member，也無法得知 function 的實作內容

Encapsulation

point.c

```
#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Point {
double x,y;
};

struct Point* makepoint(double x, double y) {
struct Point* p = malloc(sizeof(struct Point));
p->x = x;
p->y = y;
return p;
}

double distance(struct Point* p1, struct Point* p2) {
double dx = p1->x - p2->x;
double dy = p1->y - p2->y;
return sqrt(dx*dx+dy*dy);
}
```

Encapsulation

- C 程式開發者很習慣這樣的用法，在 header 宣告 data structure 跟 function，然後 implementation 是在另外一個檔案
- C 程式開發者從來都不會去存取 implementation 中的 data

Encapsulation

- 然而 C++ 卻破壞了 C 的完美封裝
- C++ member variable 必須宣告在 header

point.h

```
class Point {
public:
Point(double x, double y);
double distance(const Point& p) const;

private:
double x;
double y;
};
```

Encapsulation

point.cc

```
#include "point.h"
#include <math.h>

Point::Point(double x, double y)
: x(x), y(y)
{}

double Point::distance(const Point& p) const {
double dx = x-p.x;
double dy = y-p.y;
return sqrt(dx*dx + dy*dy);
}
```

Encapsulation

- 現在程式開發者會知道 member 有 x 跟 y
- 雖然 compiler 會限制程式開發者對 member 的存取,
  但程式開法者依然知道有這些 member

Encapsulation

- Java/C# 並沒有將 header/implementation分開，更破壞了封裝，這類的語言是無法將 class 的宣告跟定義分開
- 由於以上這些原因，實在很難接受 OO 語言是 strong encapsulation
- 實際上，很多 OO 語言只有部分或甚至沒有封裝的概念
- 結論就是 OO 讓原本 C 那種完美的封裝變得不完美了

Inheritance
繼承

Inheritance

- 其實在 OO 語言出現之前就已經有繼承的概念
- 以下舉個例子說明

Inheritance

-

namedPoint.h

```
struct NamedPoint;

struct NamedPoint* makeNamedPoint(double x, double y, char* name);
void setName(struct NamedPoint* np, char* name);
char* getName(struct NamedPoint* np);
```

Inheritance

- namedPoint.c

```
#include "namedPoint.h"
#include <stdlib.h>

struct NamedPoint {
double x,y;
char* name;
};

struct NamedPoint* makeNamedPoint(double x, double y, char* name) {
struct NamedPoint* p = malloc(sizeof(struct NamedPoint));
p->x = x;
p->y = y;
p->name = name;
return p;
}

void setName(struct NamedPoint* np, char* name) {
np->name = name;
}

char* getName(struct NamedPoint* np) {
return np->name;
}
```

Inheritance

main.c

```
#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) {
struct NamedPoint* origin = makeNamedPoint(0.0, 0.0, "origin");
struct NamedPoint* upperRight = makeNamedPoint (1.0, 1.0, "upperRight");
printf("distance=%f\n",
distance(
(struct Point*) origin,
(struct Point*) upperRight));
}
```

Inheritance

- NamedPoint 可以視為 Point 所衍生出來的，因為 NamedPoint 是 Point 的 superset，而且 member 順序都是相同的
- 這是 OO 出現之前，程式開發者常用的技巧
- C++ 的單一繼承就是用這種技巧

Inheritance

- 雖然我們可以說在 OO 之前就有這種繼承方式，但又沒有真正的繼承來的方便
- 多重繼承很難用這種技巧來達到
- 結論就是 OO 並沒有提供我們新的繼承概念，但它確實讓 data structure 的衍生更容易

Polymorphism
多型

Polymorphism

- 其實在 OO 語言出現之前就已經有多型的概念
- 以下舉個例子

Polymorphism

```
#include <stdio.h>

void copy() {
int c;
while ((c=getchar()) != EOF)
putchar(c);
}
```

- getchar() 是從 STDIN 讀取，但 STDIN 是誰？
- putchar() 是寫入 STDOUT，但 STDOUT 是誰？
- 其實這些 functions 就是運用多型，因為它們的行為會根據 STDIN/STDOUT type 而有所不同

Polymorphism

- STDIN/STDOUT 有點像 Java 的 interface，根據不同的 devices 而有不同的實作內容，但 C 又沒有 interface 這種東西，它是怎麼做到的呢？

Polymorphism

- UNIX 作業系統規定每個 IO device driver 都要支援五個 functions
  - open, close, read, write, seek
- FILE 的資料結構如下, 包含五個 pointers to functions

```
struct FILE {
void (*open)(char* name, int mode);
void (*close)();
int (*read)();
void (*write)(char);
void (*seek)(long index, int mode);
};
```

Polymorphism

- 五個 pointers of functions 指到對應的 functions, console 即具有五個 functions 的實作

```
#include "file.h"

void open(char* name, int mode) {/*...*/}
void close() {/*...*/};
int read() {int c;/*...*/ return c;}
void write(char c) {/*...*/}
void seek(long index, int mode) {/*...*/}

struct FILE console = {open, close, read, write, seek};
```

Polymorphism

- 如果 STDIN 是指到 console，則 getchar() 會呼叫 console 中的 read()，而 console 中的 read() 會指到實作的 read()

```
extern struct FILE* STDIN;

int getchar() {
return STDIN->read();
}
```

Polymorphism

- 結論就是多型只是 pointers of functions 的應用，並不是 OO 語言才有多型，但 OO 語言讓多型更安全與方便
- Pointers of functions 實現多型很危險，必須遵循一些規則，如果編程人員忘記的話，產生的 bug 會很難追也很難解
- OO 語言不需要遵循這些規則就可以很簡單實現多型

Polymorphism

- 多型厲害的地方在於，如果現在系統新增一個IO device，假設我們現在想要修改 copy 的行為，基本上完全不用動到 copy 的 code，因為 copy 並沒有依賴於特定 IO device
- 只要新的 IO device 去實作 FILE 的那五個 functions，copy 就可以直接使用
- 簡單來說，IO devices 被視為 copy 的 plugins，所以只要滿足介面，隨時都可以任意替換

Dependency Inversion
依賴倒轉

Dependency Inversion

- 依賴倒轉有兩個原則
  - 高階不應該依賴於低階，兩者都應該依賴於抽象
  - 抽象不應該依賴於具體實作，而具體實作則應該依賴於抽象
- 很饒舌吧，沒關係先繼續往下，等下你就懂了

Dependency Inversion

- 先來幾個名詞解釋
  - 高階與低階
    - 就是呼叫者(Caller)與被呼叫者(Callee)
  - 抽象
    - 介面(Interface)或抽象類別(Abstract Class)

Dependency Inversion

- 具體實作
  - 具有實作內容的非抽象類別
- 依賴
  - Class A 裡面用到 Class B, 就說 A 依賴於 B

Dependency Inversion

- 首先，沒有多型的程式架構會長得像這樣

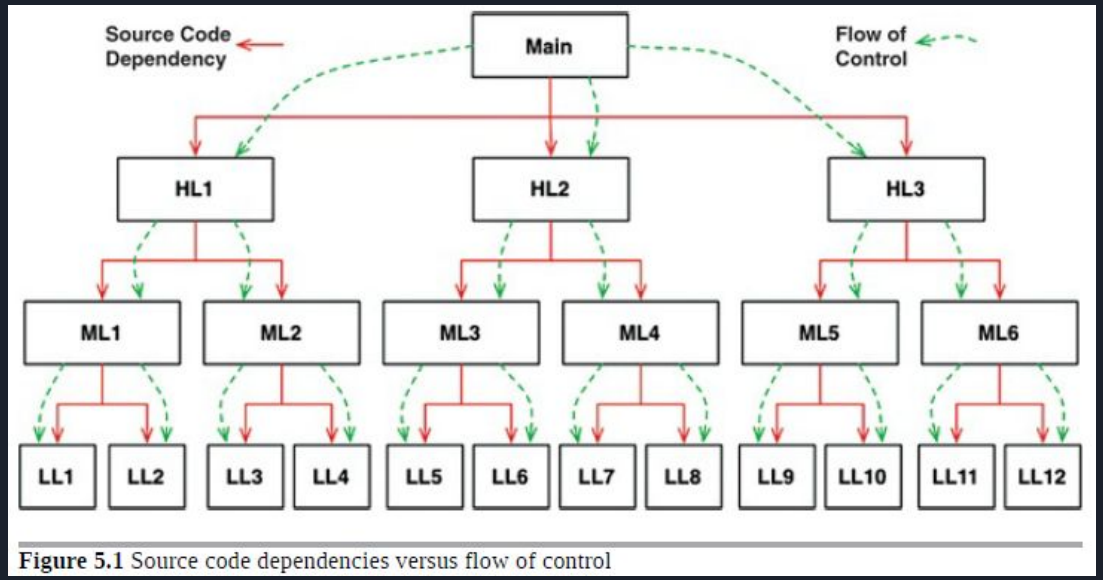

Figure 5.1 Source code dependencies versus flow of control

Dependency Inversion

- 從圖可知，Main 與 HL，HL 與 ML，ML 與 LL 之間都有依賴關係
- 舉 HL 與 ML 來說
  - HL 是高階，ML是低階
  - 高階(HL)依賴於低階(ML)

Dependency Inversion

- 導入介面之後，世界就會變得不一樣

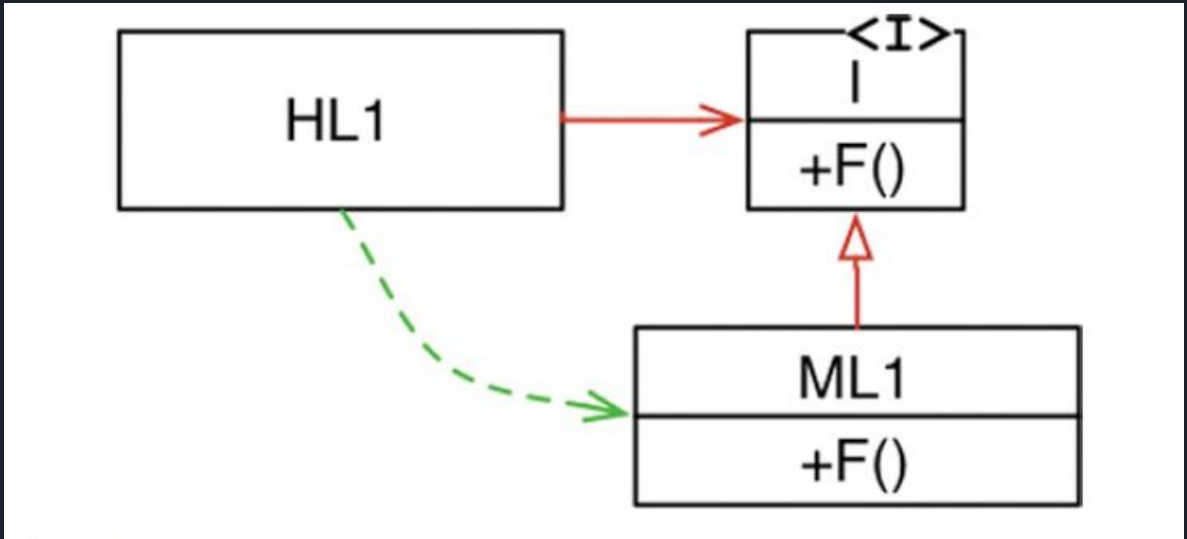

Figure 5.2 Dependency inversion

Dependency Inversion

- 導入介面後我們來回顧一下，依賴倒轉有兩個原則
  - 高階不應該依賴於低階，兩者都應該依賴於抽象
    - 高階(HL)依賴於抽象(I)，低階(ML)也依賴於抽象(I)
  - 抽象不應該依賴於具體實作，而具體實作則應該依賴於抽象
    - 抽象(I)只提供介面並沒有具體實作，具體實作(ML)則依賴於抽象(I)

Dependency Inversion

- 講了這麼多，到底依賴倒轉跟多型有什麼關係
- 依賴倒轉的目就是為了實現多型

Conclusion

Conclusion

- 封裝，繼承，多型都不是 OO 才有的，只是 OO 讓這些概念在使用上更方便更安全
- 改變不同 modules 之間的依賴性，進而達到可抽換 module 的目的