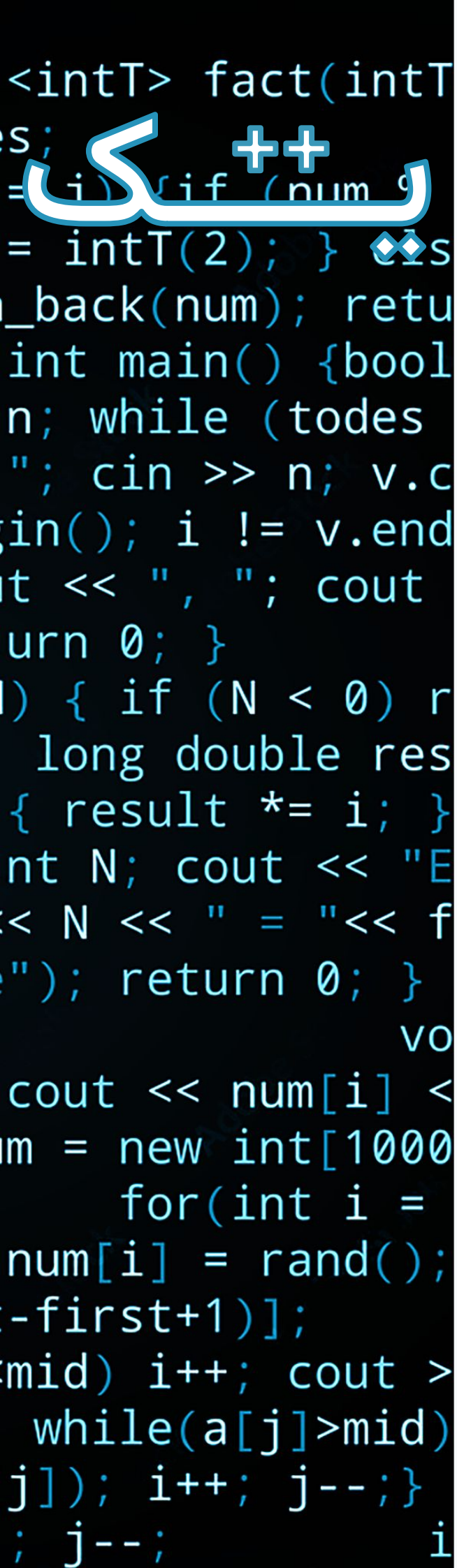

سی++ بعلاوه

std::cout

<< اصول SOLID

<< ها-Design Pattern

<< std::endl;

دانشکده مهندسی برق و کامپیوتر دانشگاه تهران

اصول SOLID و Design Pattern-ها

تا الان با طراحی شی‌گرا آشنایی اولیه‌ای داشته‌اید. در این مطلب می‌خواهیم که نخست اصول SOLID را شرح داده و سپس به معرفی یکی از design pattern-ها بپردازیم.

اصول SOLID

قواعد SOLID، برای اولین بار، در سال 2000 توسط Robert C. Martin در مقاله "Design Principles and Design Patterns" معرفی شدند. بعدها Michael Feathers این قواعد را گسترش داد و آن‌ها را با مخفف SOLID به جامعه برنامه‌نویسان تحویل داد. اصول SOLID شامل پنج اصل برای برنامه‌نویسی شی‌گرا است که در نهایت باعث تولید کدی خواناتر، با قابلیت نگهداری بیش‌تر و منعطف‌تر در مقابل تغییرات می‌شود. در ادامه با این اصول بیش‌تر آشنا می‌شویم.

S برای Single Responsibility

این قاعده، این موضوع را بیان می‌کند که هر کلاس تنها باید یک وظیفه را پیاده‌سازی کند. این در چندین مورد به ما در داشتن و نگهداری کدی تمیزتر کمک می‌کند:

- تست‌نویسی برای کلاس مورد نظر ساده‌تر شده و تعداد حالاتی که باید مورد آزمون قرار بگیرند کم‌تر می‌شود.
- روابط بین کلاس‌ها ساده‌تر شده و پیچیدگی کد کاهش می‌یابد.
- حجم کلاس‌ها کم‌تر شده که موجب افزایش خوانایی و دیباگ راحت‌تر هر بخش می‌شود.

برای فهم بهتر این اصل به مثال زیر توجه کنید:
فرض کنید یک کلاس Student داریم. در این کلاس attribute-هایی مانند نام دانشجو، شماره دانشجویی و غیره وجود دارد که در توابع آن مقداردهی و استفاده می‌شوند. بخشی از کلاس در کد زیر نشان داده شده است:

```
class Student {
public:
  void study(const Course& course) {
    Book course_book = course.get_book();
    course_book.read();
  }
  void live() {
    // Implementation
  }

private:
  std::string name_;
  std::string std_id_;
};
```

در اینجا همانطور که واضح است، کلاس Student دو وظیفه را پیاده‌سازی می‌کند.
در ادامه دلایلی را که ممکن است بخواهیم پیاده‌سازی این کلاس را تغییر دهیم، لیست می‌کنیم. اولین دلیل مربوط به نحوه پیاده‌سازی study است؛ شاید بخواهیم نحوه درس خواندن دانشجو را از «کتاب خواندن» به «دیدن ویدیوهای آموزشی» تغییر دهیم.

دومین دلیل مربوط به تغییر سبک زندگی او است؛ شاید بخواهیم یک سرگرمی جدید به زندگی دانشجو اضافه کنیم، یا شاید بخواهیم ورزش را به برنامه روزانه او اضافه کنیم. در اینجا واضح است که حداقل دو دلیل برای تغییر کلاس داریم که ناقض قاعده Single Responsibility است. شاید یک پیاده‌سازی بهتر، حذف live از دانشجو باشد.

در نهایت مهم‌ترین چیزی که در این قاعده باید به خاطر بسپارید این است: هر کلاس تنها یک دلیل برای تغییر باید داشته باشد.

O برای Open-Closed

نام این قاعده کمی ناواضح بوده و مفهوم آن از روی ظاهرش ناپیداست. Open-Closed در اصل به معنای باز بودن نسبت به گسترش (Open for extension) و بسته بودن نسبت به تغییرات (Closed for modification) است. به سخنی دیگر، این قاعده اجازه تغییر در کدی که از قبل نوشته شده را از ما سلب می‌کند در حالی که برای ایجاد قابلیت‌های جدید به برنامه، می‌توانیم آن‌ها را گسترش دهیم. برای درک بهتر به مثال زیر توجه کنید (مثال از این لینک برداشته شده است):

فرض کنید یک گیتار به صورت زیر تعریف کرده‌ایم:

```
class Guitar {
public:
  Guitar() {
    // Implementation
  }

protected:
  std::string brand_;
  std::string model_;
  std::string volume_;
};
```

پس از مدتی متوجه می‌شویم که به یک گیتار به اصطلاح «خفن‌تری» نیاز داریم؛ طراحان گیتار تصمیم می‌گیرند به آن، پترن شعله‌های آتش را اضافه کنند تا جوان‌پسندتر شود. برای اینکار شاید برخی پیشنهاد وسوسه‌کننده‌ای بدهند که کد قدیمی کلاس Guitar را تغییر دهیم تا پترن مورد نظر را به آن اضافه کنیم. این راه حل می‌تواند موجب ناپایداری کد قدیمی و ایجاد باگ‌های جدید در آن شود؛ علاوه بر آن گاهاً ممکن است برنامه‌نویسانی که کد قدیمی را نوشته بودند عوض شده باشند و جای خودشان را به برنامه‌نویسان جدیدی داده باشند که اطلاعی از پیاده‌سازی قدیمی ندارند. برای آنکه از خواندن و فهمیدن کد قدیمی اجتناب کنیم، می‌توانیم به قاعده Open-Closed پناه ببریم. راه حل درست‌تر گسترش این کلاس به صورت زیر خواهد بود:

```
class CoolGuitarWithFlames: public Guitar{
private:
  std::string pattern_;
};
```

در این صورت می‌توانیم از این موضوع اطمینان حاصل کنیم که کد قدیمی دست نخورده و پایدار باقی خواهد ماند.

L برای Liskov Substitution

قاعده Liskov این موضوع را بیان می‌کند که هر کلاس قابل جایگزین شدن با subclass-های خودش باشد.

برای درک بهتر به مثال زیر توجه کنید:

فرض کنید کلاس زیر را برای دسته پرندگان ساخته‌ایم:

```
class Bird {
public:
  virtual void fly() {
    // Implementation
  }
};
```

حال اگر بخواهیم کلاسی برای پنگوئن بسازیم، به طوری که زیرنوعی از کلاس پرنده باشد می‌توانیم به صورت زیر عمل کنیم:

```
class Pinguin : public Bird {
public:
  void fly() override {
    throw std::runtime_error(
      “I can’t; I have little wings!!”);
  }
};
```

واضح است که پنگوئن قابلیت پرواز نداشته و نمی‌تواند تابع fly را پیاده‌سازی کند. بنابراین اگر کلاس Penguin که زیرنوعی از کلاس Bird است را جایگزین کلاس Bird کنیم، پیاده‌سازی موردنظر قابل قبول نخواهد بود. این یک مثال واضح از نقض قاعده Liskov است. یک راه برای حل این مشکل، می‌تواند استفاده از پیاده‌سازی‌هایی باشد که پرندگانی که نمی‌توانند پرواز کنند را نیز در نظر بگیرد.

I برای Interface Segregation

این اصل بیان می‌کند که interface-هایی که برای کلاس‌های خود تعریف می‌کنیم نباید بیش از حد بزرگ باشند به طوری که متدهایی که برای آن تعریف می‌کنیم بدون استفاده بمانند.

طبق این اصل، یک interface در صورت بزرگ بودن، باید به interface-هایی که وظیفه کوچک‌تری دارند تقسیم شود.

فرض کنید که می‌خواهیم برای پرینتر و اسکنر یک interface بنویسیم:

```
class IPrinterAndScanner {
Public:
  virtual void print() = 0;
  virtual void scan() = 0;
};
```

در این صورت کلاس‌های پرینتر و اسکنر به شکل زیر نوشته می‌شوند:

```
class SimplePrinter : public
                    IPrinterAndScanner {
public:
  void print() override {
    // Implementation
  }
  void scan() override {
    // Does nothing
  }
};

class DigitalScanner : public
                    IPrinterAndScanner {
public:
  void print() override {
    // Does nothing
  }
  void scan() override {
    // Implementation
  }
};
```

همانطور که می‌بینیم، در interface ما بیشتر از چیزی که یک کلاس نیاز دارد قرار گرفته است و کلاسی که آن را پیاده‌سازی می‌کند به ناچار آن را خالی می‌گذارد.

با پیروی از اصل Interface Segregation، کد به صورت زیر تغییر می‌کند:

```
class IPrinter {
public:
  virtual void print() = 0;
};

class IScanner {
public:
  virtual void scan() = 0;
};
…
```

```
class SimplePrinter : public IPrinter {
public:
  void print() override {
    // Implementation
  }
};
class DigitalScanner : public IScanner {
public:
  void scan() override {
    // Implementation
  }
};
```

در این حالت، هر کلاس فقط توابع مورد نیازش را از interface مورد نظر گرفته و در صورت نیاز به هر دو آنها، هر دو را پیاده‌سازی می‌کند.

D برای Dependency inversion

این اصل یعنی کلاس‌های سطح بالا نباید به طور مستقیم به کلاس‌های سطح پایین وابسته باشند. کلاس‌های سطح بالا باید با استفاده از رابطی به کلاس‌های سطح پایین دست یابند تا در صورت تغییر کلاس سطح پایین، کلاس سطح بالا تحت تأثیر آن قرار نگیرد.

یک راه حل که می‌تواند در بسیاری از مواقع به ما کمک کند، استفاده از Interface-ها است. در این صورت کلاس سطح بالا، به Interface وابسته می‌شود و نه به کلاس سطح پایین.

به عنوان مثال فرض کنید که می‌خواهیم یک سیستم عامل جدید به نام OS ایجاد کنیم. این سیستم عامل باید Mouse و Keyboard را پشتیبانی کند. در این صورت می‌توانیم دو کلاس Keyboard و Mouse داشته باشیم:

```
class OS {
public:
  OS() {
    keyboard_ = new Keyboard();
    mouse_ = new Mouse();
  }
private:
  Keyboard* keyboard_;
  Mouse* mouse_;
};
```

با این کار ما کلاس سطح بالای OS را به کلاس‌های Keyboard و Mouse وابسته کرده‌ایم. این کار مشکلاتی از جمله سخت شدن تست کلاس OS، و وابستگی به دو کلاس را ایجاد می‌کند. وجود وابستگی، امکان تعویض Keyboard با یک نوع کیبورد دیگر، یا Mouse با موسی دیگر را از بین می‌برد.

باید این کلاس‌ها را به طریقی از یکدیگر جدا کنیم تا وابستگی مستقیم را رفع کنیم. برای این کار می‌توان interface-هایی برای دو کلاس تعریف کرد و داخل OS از آنها استفاده کرد.

دو رابط MouseInterface و KeyboardInterface را تعریف می‌کنیم و Mouse و Keyboard را طوری تغییر می‌دهیم که interface-های متناظرشان را پیاده‌سازی کنند. حال کلاس OS به شکل زیر می‌شود:

```
class OS {
public:
  OS(KeyboardInterface* k,
     MouseInterface* m) {
    keyboard_ = k;
    mouse_ = m;
  }

private:
  KeyboardInterface* keyboard_;
  MouseInterface* mouse_;
};
```

کنون می‌توانیم از هر موس یا کیبوردی که interface-اش را پیاده‌سازی می‌کند در این سیستم عامل استفاده کنیم. این دو رابط کلاس‌های سطح پایین مربوط به خود را هندل می‌کنند و چیزی که ما با آن کار می‌کنیم، این رابط‌ها هستند و نه خود کلاس‌های سطح پایین که با تغییرات احتمالی، کد سطح بالا را تحت تأثیر قرار می‌دهند.

ها-Design Pattern

مقدمه

الگوهای طراحی، نوعی رویکرد و راهبرد برای حل مشکلات در زمینه‌های مختلف هستند به طوری که می‌توانیم از آنها در حل مشکلات خاص بهره ببریم.
خوبی استفاده از این الگوها این است که با استفاده از آن‌ها می‌توانیم با اطمینان بیشتری به توسعه برنامه بپردازیم، سرعت و کیفیت خود را افزایش دهیم و با تمرکز بیش‌تری روی پروژه کار کنیم. در اکثر مواقع، این کار موجب افزایش خوانایی و تمیزتر شدن کد هم می‌شود. هنگام ریفکتور کردن کدها هم استفاده از الگوهای طراحی حائز اهمیت است.
موردی که باید حواسمان به آن باشد این است که نیاز نیست تمام این الگوها را از بر باشیم و نحوه پیاده‌سازی آنها را موبه‌مو بدانیم. بلکه صرفا باید بدانیم که چه زمان لازم است از کدام الگو استفاده کنیم. در کل نباید در استفاده از این الگوها دچار افراط و تفریط شویم!
در ادامه به یکی از الگوهای طراحی اشاره می‌کنیم. توصیه ما این است که سایر الگوهای طراحی را مطالعه کنید و به دانش خود را در این زمینه بیفزایید. برای آشنایی با الگوهای دیگر، می‌توانید به این لینک مراجعه کنید.

دسته بندی

الگوهای طراحی اولین بار در کتاب Design Patterns: Elements of Reusable Object-Oriented Software که توسط چهار نویسنده - که به آنها Gang of Four، یا به اختصار GoF می‌گویند - جمع‌آوری و عرضه شد.
این کتاب به دو بخش تقسیم شده که بخش اول درباره طراحی شی‌گرا و بخش دوم 23 الگو طراحی را شرح می‌کند. این 23 الگو به 3 قسمت تقسیم شده اند:

1. الگوهای ابداعی (Creational Patterns): این الگوها بیشتر در رابطه با مکانیزم‌های ایجاد اشیاء صحبت می‌کنند که این باعث افزایش انعطاف‌پذیری کد می‌شود. همچنین استفاده مجدد کد را بالا می‌برد.
2. الگوهای ساختاری (Structural Patterns): این الگوها در رابطه با نحوه جمع‌آوری و نگه‌داری اشیاء در کنار هم است؛ به گونه‌ای که ساختارهای ما انعطاف‌پذیر و کارآمد باقی بمانند.
3. الگوهای رفتاری (Behavioral Patterns): این الگوها در مورد ایجاد روابط موثر میان اشیاء و اختصاص مسئولیت‌ها به شکل درست بین آنها می‌باشند.

Factory Design Pattern

الگوی Factory یک الگوی ابداعی است که در آن از یک Interface برای ایجاد کلاس‌ها استفاده می‌شود؛ این Interface به ما یک کلاس abstract را می‌دهد، ولی می‌توانیم با آن به هر تایپی که بخواهیم اشاره کنیم. به زبان ساده‌تر این یک الگویی است که برای ما اشیا مختلف را می‌سازد و آنها را به صورت abstract در اختیار ما قرار می‌دهد.
مثلا می‌خواهیم برای تبادل داده بین کامپیوترهای مختلف، در صورت متصل بودن به شبکه وای‌فای از آن، و در غیر این صورت از بلوتوث استفاده کنیم.

کلاس‌ها بدین صورت خواهند بود:

```
class Wifi {
public:
  Wifi();
  ~Wifi();
  void connect();
  void disconnect();
  ...
};

class Bluetooth {
public:
  Bluetooth();
  ~Bluetooth();
  void connect();
  void disconnect();
  ...
};
```

کد برنامه به صورت زیر خواهد بود:

```
if (NetUtils::has_wifi(node)) {
  Wifi wifi;
  wifi.connect();
}
else {
  Bluetooth bluetooth;
  bluetooth.connect();
}
// ...
if (NetUtils::has_wifi(node)) {
  Wifi wifi;
  wifi.disconnect();
}
else {
  Bluetooth bluetooth;
  bluetooth.disconnect();
}
```

این حالت حاوی کد تکراری است و از چند if یکسان استفاده شده.

انجام این کار در تابع top level-تری که باید به طور انتزاعی توابع اصلی دیگر را فراخوانی کند درست نیست.

در اینجا می‌توانیم از فکتوری استفاده کنیم. بدین صورت که یک کلاس مانند Network ایجاد می‌کنیم و دو زیر کلاس برای آن تعریف می‌کنیم.

در این حالت با توجه به اینکه از چندریختی هم استفاده کرده‌ایم برخی از کارکردها را مخفی کرده‌ایم و کد ما گسترش پذیری بیشتری خواهد داشت. پس کلاس‌های ما بدین صورت خواهند شد:

```
class Network {
  Network() {...}
  ~Network() {...}
  virtual void connect() = 0;
  virtual void disconnect() = 0;
};

class Wifi : public Network {
  Wifi() {...}
  ~Wifi() {...}
  void connect() override {...}
  void disconnect() override {...}
  ...
};

class Bluetooth : public Network {
  Bluetooth() {...}
  ~Bluetooth() {...}
  void connect() override {...}
  void disconnect() override {...}
  ...
};
```

حال تابع فکتوری را می‌نویسیم:

```
Network get_network_interface(Node node) {
  return NetUtils::has_wifi(node) ?
         Wifi() : Bluetooth();
}
```

در نهایت کد برنامه به شکل زیر تغییر می‌کند:

```
Network nw = get_network_interface(node);
nw.connect();
// ...
nw.disconnect();
```

این یک نمونه از کاربرد فکتوری بود. البته این پترن در سناریوهای دیگری هم کاربرد دارد که می‌توانید در این باره بیشتر تحقیق کنید.

در طول سالیان متمادی، همواره تلاش و وظیفه دستیاران آموزشی، کمک به ارائه مفیدتر درس و انتقال بهتر مطالب به دیگران بوده است؛ دستیاران آموزشی درس برنامه‌سازی پیشرفته نیز از این قاعده مستثنی نبوده و در طول ترم‌های گذشته همواره سعی کرده‌اند در قالب‌های متفاوت، به انتقال مفاهیم این درس کمک کرده باشند.

سی بعلاوه پلاس‌پلاس مجله‌ای در راستای همین هدف است که اولین موضوع از آن در ترم بهار 1402 ارائه شد و همچنان ادامه دارد. موضوعات این مجله فراتر از مقاصد پایه درس بوده و صرفا برای اطلاعات بیش‌تر و درک بهتر مفاهیم ارائه می‌شوند. خواندن و یادگیری آن اجباری نیست ولی برای یادگیری عمیق‌تر توصیه می‌شود.

تاریخ انتشار: بهار 1402

نویسندگان: سامان اسلامی نظری، علی عطاللهی

ویراستاران: میثاق محقق

طراحان: الهه خداوردی، شهریار عطار

دستیار آموزشی ارشد: طاها فخاریان