Thinking in Classes

Dheeraj Jha

www.jhadheeraj.com

Agenda

- Who am I?
- Disclaimer
- Class
- What is the Design Goal?
- SOLID Design Principles
- Hypothetical UseCase
- Core Guidelines

Who am I?

- Practicing software engineering for more than a decade.
- Primarily using C++.
- Have worked in different domains. Currently working in IoT.
- Started CppIndia 2 years back.
- Recently started blogging and podcast.
- Visit www.jhadheeraj.com
- Follow me LinkedIn(@jhadheeraj), and Twitter(@dheerajjha03).

Disclaimer

There is no perfect design. Your design will evolve over the time.

Design is a journey not a destination.

Design may change with the change in requirements. Your responsibility is to minimise the change.

You will learn by try and error.

Classes

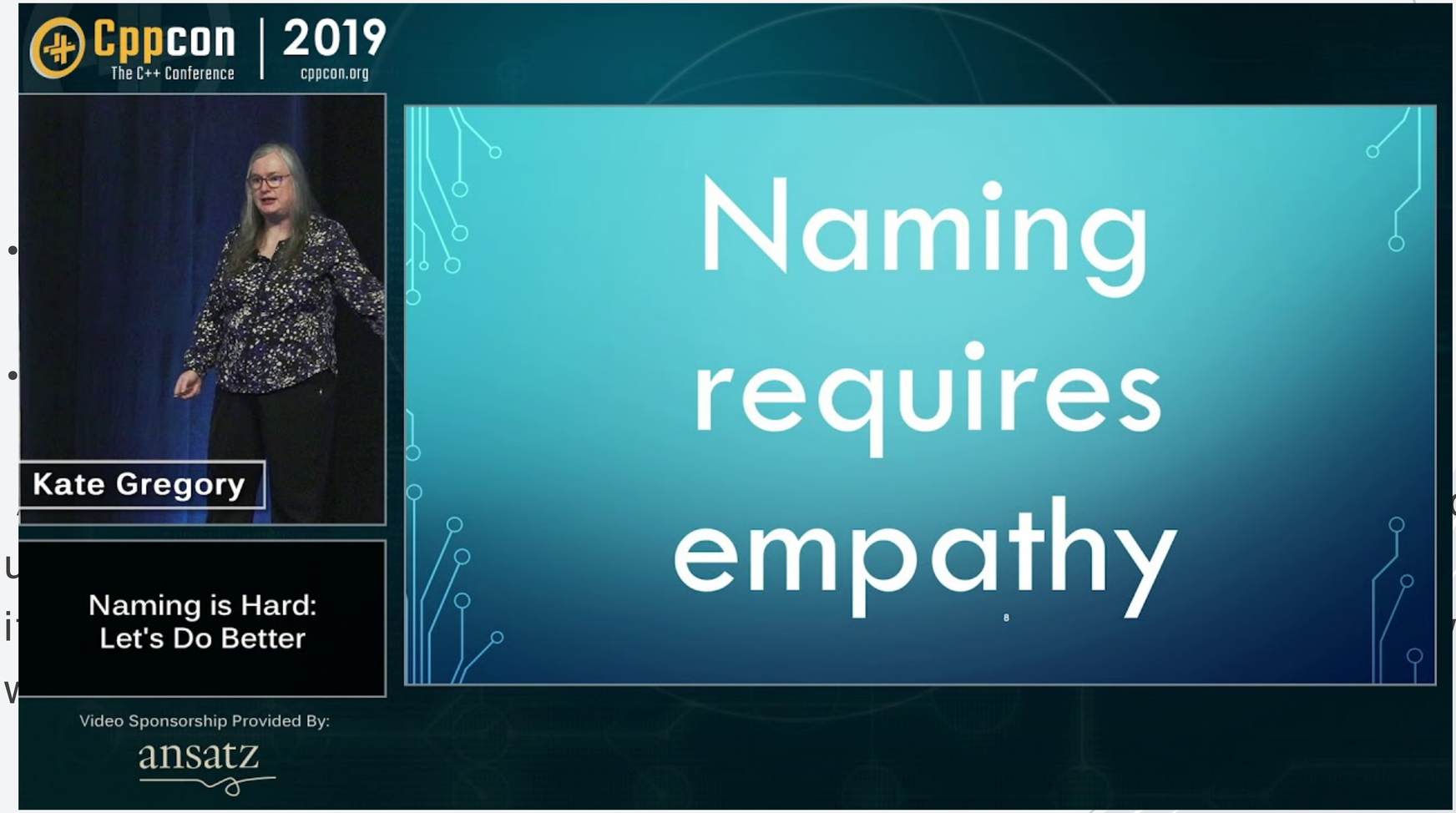
[Bjarne Stroustrup: Why I Created C++ I Big Think]

Classes

The class as the thing you have in the program to represent a concept in your application world. So if you are a mathematician, a matrix will become a class, if you are a businessman, a personnel record might become a class, in telecommunications a dial buffer might become a class—you can represent just about anything as a class.

- Bjarne Stroustrup

Classes - Naming



What is the Design Goal?

The goal of software architecture is to minimise the human resources required to build and maintain the required system.

SOLID Design Principles

S: Single Responsibility Principle

O: Open-Closed Principle

L: Liskov Substitution Principle

I: Interface Segregation Principle

D: Dependency Inversion Principle

S: Single Responsibility Principle

A module should be responsible to one, and only one, actor.

S: Single Responsibility Principle

```
class Employee{
    int employeeId;
    std::string name;
    std::string address;
    int mobileNumber;
    double reportedHours;

    public:
    double calculatePay();
    double reportedHours();
    void save();
}
```

Problem Statment

S: Single Responsibility Principle

```
class HourReporter{
    public:
    double reportHours();
};


class EmployeeSaver{
    public:
    bool saveEmployee()
};




class Employee{
    // ...
    HourReporter* reporter;
    EmployeeSaver* saver;
    public:
    Employee(HourReporter* reporterObj=nullptr,
EmployeeSaver* empSaverObj=nullptr){
        reporter = reporterObj ?: new HourReporter;
        saver = empSaverObj ?: new EmployeeSaver;
    }
~Employee(){
// delete memory
}
double reportHours(){
     return reporter->reportHours();
}
bool saveEmployee(){
    return saver->saveEmployee();
}
};
```

O: Open-Closed Principle

A class should be open for extension but close for modification.

O: Open-Closed Principle

```
class Product {
    string  name;
    COLOR   color;
    SIZE    size;  };

class Filter{
    vector<product *> filterByColor(vector<product
*> prod, const COLOR productColor){
        vector<product *> result;
        for (auto &i : prod)
            if (i->color == productColor)
                result.push_back(i);
        return result;
    }

    static vector<product *>
filterBySize(vector<product *> prod, const SIZE
productColor) {
        vector<product *> result;
        for (auto &i : result)
            if (i->size == productColor)
                result.push_back(i);
        return result;
    }
};
```

Problem Statment

O: Open-Closed Principle

```
class Specification{
    virtual ~Specification() = default;
    virtual bool is_satisfied(product *item) const = 0;
};

class ColorSpecification : Specification {
    COLOR color;
    ColorSpecification(COLOR productColor) : color(productColor) {}
    bool is_satisfied(Product *item) const { return item->color ==
color; }
};

class SizeSpecification : Specification{
    SIZE size;
    SizeSpecification(SIZE productSize) : size(productSize) {}
    bool is_satisfied(Product *item) const { return item->size == size;
}
};

class Filter {
    virtual vector<Product *> filter(vector<Product *> items, const
Specification &spec) = 0;
};
class BetterFilter : Filter {
    vector<Product *> filter(vector<Product *> items, const
Specification &spec) {
        vector<Product *> result;
        for (auto &p : items)
            if (spec.is_satisfied(p))
                result.push_back(p);
        return result;
    }
};
```

L: Liskov Substitution Principle

Subtypes must be substitutable for their base types without altering the correctness of the program.

Functions that use pointers/references to base classes must be able to substitute by its derived classes.

L: Liskov Substitution Principle

```
class Bird {
public:
    virtual void setLocation(double
longitude, double latitude) = 0;
    virtual void setAltitude(double altitude)
= 0;
    virtual void draw() = 0;
};
```

```
class Penguin : public Bird{


    void
Penguin::setAltitude(double
altitude)
    {
        //altitude can't be set
because penguins can't fly
        //this function does nothing
    }
};
```

Problem Statment

L: Liskov Substitution Principle

```
// Not so good solution
void ArrangeBirdInPattern(Bird* aBird)
{
    Pengiun* aPenguin = dynamic_cast<Pengiun*>(aBird);
    if(aPenguin)
        ArrangeBirdOnGround(aPenguin);
    else
        ArrangeBirdInSky(aBird);
}
```

L: Liskov Substitution Principle

```
// Solution 1
void ArrangeBirdInPattern(Bird*
aBird)
{
    if(aBird->isFlightless())
        ArrangeBirdOnGround(aBird);
    else
        ArrangeBirdInSky(aBird);
}
```

```
// Solution 2 - Better one
class Bird {
public:
    virtual void draw() = 0;
    virtual void setLocation(double
longitude, double latitude) = 0;
};
class FlightfulBird : public Bird {
public:
    virtual void setAltitude(double
altitude) = 0;
};
```

I: Interface Segregation Principle

Clients should not be forced to depend on interfaces that they do not use.

I: Interface Segregation Principle

```
class Itv{
    public:
        bool changeChannel() = 0;
        bool increaseVolume() = 0;
        bool decreaseVolume() = 0;
        bool playAmazon() = 0;
        bool PlayNetflix() = 0;
};
```

Problem Statment

```
class SmartTv : public Itv{
    public:
        bool changeChannel() { ... }
        bool increaseVolume() { ... }
        bool decreaseVolume() { ... }
        bool playAmazon() { ... }
        bool PlayNetflix() { ... }
};
class NonSmartTv : public Itv{
        public:
        bool changeChannel() { ... }
        bool increaseVolume() { ... }
        bool decreaseVolume() { ... }
        bool playAmazon() {}
        bool PlayNetflix() {}
}
```

I: Interface Segregation Principle

```
class ITvFeature{
    public:
        bool changeChannel() = 0;
        bool increaseVolume() = 0;
        bool decreaseVolume() = 0;
};

class ISmartTvFeature{
    public:
        bool playAmazon() = 0;
        bool PlayNetflix() = 0;
};
```

```
class TV : public ItvFeature{
    public:
        bool changeChannel() { }
        bool increaseVolume() { }
        bool decreaseVolume() { }
    }
class SmartTv : public ItvFeature, ISmartTvFeature{
    public:
        bool changeChannel() { ... }
        bool increaseVolume() { ... }
        bool decreaseVolume() { ... }
        bool playAmazon() { ... }
        bool PlayNetflix() { ... } };
```

D: Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

D: Dependency Inversion Principle

```
class FileUploader{
        AwsHandler awsObj;
    public:
        bool connect(){
            awsObj.connectS3();
        }
        bool uploadFile(){
            awsObj.uploadFileToS3();
        }
};
```

```
class AwsHandler{
    public:
        connectS3();
        uploadFileToS3();
};
```

Problem Statment

D: Dependency Inversion Principle

```
class ICloudHandler{
    public:
        connect() = 0;
        uploadFile() = 0;
};

class AwsHandler : public ICloudHandler{
    public:
        connect();
        uploadFile();
};
```

```
class FileUploader{
        ICloudHandler* cloudObj;
    public:
        bool connect(){
            cloudObj->connect();
        }
        bool uploadFile(){
            cloudObj->uploadFile();
        }
};
```

Hypothetical UseCase

- Let's say you have to process few types of messages.
- You will be receiving these messages and storing these in a queue,
- and you have to send this to other party.

Problem Statment

Hypothetical UseCase - cont…

How will you handle multiple types of messages?

```
class SendAndReceiveMsg{
    private:
        std::queue msgQue;
    public:
        Void storeMsg();
        Void sendMsg();
        Void receiveMsg();
}
```

Hypothetical UseCase - Cont...

- There comes a new requirement and before you store these messages in queue, you have to add some headers to all the messages.

DRY (Don't repeat yourself)

Hypothetical UseCase - Cont...

```
class MsgQue{
    private:
    std::queue<Type> msgQue;
    public:
    void push(Type& msg){
        //Add common header to message.
        msgQue.push(msg);
}
```

Hypothetical UseCase - Cont...

```
class sendMsg{
    private:
        Type msg;
        MsgQue &queue;
    public:
        sendMsg(MsgQue &que)
:queue{que} {}
        Type getMsgFromQue();
        bool sendMsg(Type);
}
```

```
class recvMsg{
    private:
        Type msg;
        MsgQue &queue;
    public:
        recvMsg(MsgQue &que)
:queue{que} {}
        Type pushMsgFromQue();
        bool recvMsg(Type);
}
```

Core Guidelines

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c-core-guidelines

Core Guidelines - Organising

C.1: Organise related data into structures (structs or classes)

```
void draw(int x, int y, int x2, int y2);

void draw(Point from, Point to);
```

C.2: Use class if the class has an invariant;
use struct if the data members can vary independently

Core Guidelines - Encapsulation

C.3: Represent the distinction between an interface and an implementation using a class

```
class CloudConnection{
    public:
        void connect();

    private:
        bool checkInternet();
        string getConnecctionString();
};
```

Core Guidelines - Encapsulation

C.9: Minimise exposure of members

```
class Foo {
public:
    int bar(int x) { check(x);
return do_bar(x); }
protected:
    int do_bar(int x); // do some
operation on the data
private:
    // ... data ...
};
```

```
class Dir : public Foo {
    int mem(int x, int y)
    {
        /* ... do something ... */
        return do_bar(x + y);
    }
};
```

Core Guidelines - Inheritance

C.1 rchies

C.122: Use abstract aration of interface and

C.1 reason

Core Guidelines - Default Constructors

C.20: If you can avoid defining default operations, do

The rule of zero

C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all

The rule of five

Core Guidelines - Construction

C.41: A constructor should create a fully initialised object

C.42: If a constructor cannot construct a valid object, throw an exception

C.45: Don’t define a default constructor that only initialises data members; use in-class member initialisers instead

Core Guidelines - Destruction

C.30: Define a destructor if a class needs an explicit action at object destruction

C.31: All resources acquired by a class must be released by the class's destructor

C.33: If a class has an owning pointer member, define a destructor

Ref:

- Bjarne Stroustrup: Why I Created C++ I Big Think
- Back to Basics: RAII and the Rule of Zero - Arthur O'Dwyer - CppCon 2019
- CppCon 2019: Kate Gregory “Naming is Hard: Let's Do Better”
- Facade Pattern

Questions are guaranteed in life; Answers aren't.

Thank You