ECE 449/590 – OOP and Machine Learning
Lecture 13 Template Method, Prototypes, and Singleton

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

October 5, 2022

Outline

Reading Assignment

- This lecture: More on Design Patterns
- Next lecture: Deep Learning 5

Outline

The Design Problem

```
void eval_add::eval(vars_type &variables, const kwargs_type &kwargs) {
    ... // retrieve a and b from variables
    ... // perform the computation with a and b
    ... // update variables
}
void eval_sub::eval(vars_type &variables, const kwargs_type &kwargs) {
    ... // retrieve a and b from variables
    ... // perform the computation with a and b
    ... // update variables
}
void eval_mul::eval(vars_type &variables, const kwargs_type &kwargs) {
    ... // retrieve a and b from variables
    ... // perform the computation with a and b
    ... // update variables
}
```

- Evaluating these operator types follows very similar steps.
    - But the computations are different.
- How to organize the code to define the steps for an algorithm but leave flexibility to redefine each individual step?

The Template Method Pattern

- A behavioral pattern
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
  - AbstractClass: implement a template method defining the skeleton of an algorithm using primitive operations
  - ConcreteClass: implement the primitive operations to carry out subclass-specific steps
- It has conceptual relationship with the `template`s in the C++ language.

The Template Method

```
class eval_binary: public eval_op {
    virtual tensor compute(const tensor &a, const tensor &b) = 0;
public:
    eval_binary(const expression &expr);
    void eval(vars_type &variables, const kwargs_type &kwargs) final;
}; // class eval_binary

void eval_binary::eval(vars_type &variables, const kwargs_type &kwargs) {
    assert(inputs_.size() == 2);
    auto ita = variables.find(inputs_[0]);
    auto itb = variables.find(inputs_[1]);
    ... // handle errors for ita and itb
    variables[expr_id_] = compute(ita->second, itb->second);
}
```

- The template method (`eval`) invokes primitive operations (`compute`) to complete an operation following an algorithm.
- Pure `virtual` primitive operations like `compute` should be implemented in derived classes.

Discussions

```
class eval_binary: public eval_op {
    virtual tensor compute(const tensor &a, const tensor &b) = 0;
public:
    eval_binary(const expression &expr);
    void eval(vars_type &variables, const kwargs_type &kwargs) final;
}; // class eval_binary
```

- Usually, the template method is `public` and primitive operations are `private`.
  - Primitive operations should only be invoked following a specific order as defined in the template method.
- The template method is usually non-`virtual` or `final`.
  - Derived classes should not redefine the skeleton of the algorithm.

Implement Primitive Operations

```
class eval_add: public eval_binary {
    ...
    tensor compute(const tensor &a, const tensor &b) override;
}; // class eval_add
tensor eval_add::compute(const tensor &a, const tensor &b) {
    ... // make sure a and b to have the same shape
    ... // create c to have the same shape as a and b
    ... // add elements of a and b to obtain elements of c
    return c;
}
```

- There is no need for `eval_add` to implement `eval`.

Summary of Participants of the Template Method Pattern

- AbstractClass (`eval_binary`)
  - Define abstract primitive operations that concrete subclasses define to implement steps of an algorithm
  - Implement a template method defining the skeleton of an algorithm, which calls primitive operations as well as other operations
- ConcreteClass (`eval_add`, `eval_sub`, etc.)
  - Implement the primitive operations to carry out subclass-specific steps of the algorithm

Benefits of the Template Method Pattern

- Template methods are the means for factoring out common behavior in library classes.
  - It is a fundamental pattern for code reuse.
- Force a derive class to extend an operation in the correct way
  - A `virtual` function can be used to allow a derived class to override a base class operation.
  - However, if that operation has to be carried out in specific steps and/or has to call specific functions, the derived class must remember to follow such rules.
  - Defining such behavior as a template method in the base class would enforce such rules, while still allows the derived classes to extend the behavior.

Outline

The Design Problem

```
evaluation::evaluation(const std::vector<expression> &exprs) {
    for (auto &expr: exprs) {
        if (expr.get_op_type() == "Input") {
            ops_.push_back(std::make_shared<eval_input>(expr));
        }
        else if (expr.get_op_type() == "Const") {
            ops_.push_back(std::make_shared<eval_const>(expr));
        }
        else if (expr.get_op_type() == "Add") {
            ops_.push_back(std::make_shared<eval_add>(expr));
        }
        ...
}
```

- You have to modify this function for any new type of operations.
- `evaluation` knows too much about the operation implementations.
  - What if we want our EasyNN framework to support other types of operators provided by a third-party at runtime?
- We need to create an object whose type is only known at runtime (as a string).

The Prototype Pattern

- A creational pattern
- Specify the kinds of objects (operator implementations) to create using a prototypical instance, and create new objects by copying this prototype
  - Prototype: declare the interface for cloning itself
  - Client: create objects by cloning the prototype

The Prototype Interface

```
class eval_op {
    ...
    virtual std::shared_ptr<eval_op> clone(const expression &expr) = 0;
}; // class eval_op
```

- Since all operator implementations are derived from `eval_op`, it works as a common interface of the prototypes.
- The `clone` function is supposed to return a clone of the prototype.
  - As the prototype helps to create new objects, the `clone` function may take additional parameters in order to call corresponding ctors.
  - Indeed, `clone` works more like a constructor than simply making a copy.

Implement a Prototype

```
class eval_const: public eval_op {
    ...
    std::shared_ptr<eval_op> clone(const expression &expr) override;
}; // class eval_const
std::shared_ptr<eval_op> eval_const::clone(const expression &expr) {
    return std::make_shared<eval_const>(expr);
}
```

- For this example, we can call the ctor to clone the object.
  - In some sense, the prototype only provides the type information.

Prototype Storage

```
typedef std::map<std::string, std::shared_ptr<eval_op>> eval_op_proto_map;

class eval_const: public eval_op {
    ...
public:
    static void store_prototype(eval_op_proto_map &proto_map) {
        assert(proto_map.find("Const") == proto_map.end());
        proto_map["Const"] = std::make_shared<eval_const>(); // where is expr?
    }
    ...
}; // class eval_const
```

- We can store the prototypes in a container.
- Since we may need to search for a specific prototype by name, an associative container is necessary.
- Anything missing?

Updating Constructors

```
class eval_op {
protected:
    eval_op(): expr_id_(-1) {}
    eval_op(const expression &expr);
    ...
}; // class eval_op

class eval_const: public eval_op {
public:
    eval_const() {}
    eval_const(const expression &expr);
    ...
}; // class eval_const
```

- Need to provide default ctors for the prototypes.

The Client

```
class evaluation {
public:
    evaluation(const std::vector<expression> &exprs,
        eval_op_proto_map &proto_map);
    ...
}; // class evaluation

evaluation::evaluation(const std::vector<expression> &exprs,
    eval_op_proto_map &proto_map) {
    for (auto &expr: exprs) {
        auto it = proto_map.find(expr.get_op_type());
        if (it == proto_map.end()) ...; // handling errors
        ops_.push_back(it->second->clone(expr));
    }
}
```

- `evaluation` knows nothing about the operator implementations.
- A corresponding `clone` function will be called to generate a object as specified by `get_op_type()`.
- But where does `proto_map` come from?

Summary of Participants of the Prototype Pattern

- Prototype (`eval_op`)
  - Declare an interface for cloning itself
- ConcretePrototype (`eval_const`, `eval_input`, etc)
  - Implement an operation for cloning itself
- Client (`evaluation::evaluation`)
  - Create a new object by asking a prototype to clone itself

Benefits of the Prototype Pattern

- Hide concrete product classes from the client
  - Greatly reduce the number of names the client know about – less coupling
- Add and remove products at runtime, or even configure an application with classes dynamically
- Specify new types of objects by varying values or structure without introducing new class types
- Reduce the necessity of inheritance for other creational patterns
  - How can you add new operator types while reusing previous `evaluation` class design without using prototypes?

Outline

The Design Problem

```
evaluation::evaluation(const std::vector<expression> &exprs,
    eval_op_proto_map &proto_map);
void eval_const::store_prototype(eval_op_proto_map &proto_map);
void eval_input::store_prototype(eval_op_proto_map &proto_map);
...
```

- When calling these functions, all the parameters `proto_map` should refer to the same `eval_op_proto_map` object.
- How to enforce the requirement?
  - People familiar with C may propose to use a global variable `proto_map`. However, there is no guarantee all these functions will use that `proto_map` object.

The Singleton Pattern

- A creational pattern
- Ensure a class only has one instance, and provide a global point of access to it
- Unlike previous patterns, the Singleton pattern won't rely on polymorphism.

The Singleton Interface

```
class eval_op_prototypes {
    // prevent creation of additional instances
    eval_op_prototypes(const eval_op_prototypes &) = delete;

    eval_op_prototypes();

    eval_op_proto_map proto_map_;

public:
    std::shared_ptr<eval_op> locate(std::string name);

    static eval_op_prototypes &instance(); // access the only instance
}; // class eval_op_prototypes
```

- Use `static` member function to provide access to the only class instance.
  - You need an object to call non-`static` member functions. When all constructors are `private`, without a `static` member function, it is not possible to get an object to start with.
- Let's leave the line with `= delete` to later lectures.

The Singleton Implementation

```
eval_op_prototypes &eval_op_prototypes::instance() {
    static eval_op_prototypes instance; // the only instance
    return instance;
}
```

- `static` variable in a function is constructed the first time when the function is called.
  - Will persist throughout the program execution.
  - Most importantly, the C++ runtime guarantees that the variable is constructed <u>exactly once</u> even in a multi-threading environment.
- A `static` member function can access all `private` members.

Use Singleton

```
class evaluation {
public:
    evaluation(const std::vector<expression> &exprs);
    ...
}; // class evaluation

evaluation::evaluation(const std::vector<expression> &exprs) {
    for (auto &expr: exprs) {
        std::shared_ptr<eval_op> p
            = eval_op_prototypes::instance().locate(expr.get_op_type());
        ops_.push_back(p->clone(expr));
    }
}
```

- To access the only instance, one just need to make sure the Singleton class type is available.
- You may need to add error handling code depending on how `locate` handles errors.

Initializing the Singleton Object

```
eval_op_prototypes::eval_op_prototypes() {
    eval_const::store_prototype(proto_map_);
    eval_input::store_prototype(proto_map_);
    ...
}
```

- The prototypes can be initialized in the default ctor.
  - Or you may initialize them from other places before using the singleton.

Benefits of the Singleton Pattern

- Controlled access to sole instance
  - The compiler will enforce the controlled access via protections.
- Reduced name space
  - A global variable would provide global access but will pollute the global name space since you have to name it.
  - The singleton, on the other hand, simply requires a type.
- Permit refinement of operations and representation
  - Since accesses are centralized, it is easy to refine/replace the singleton with an updated implementation.
- Permit a variable number of instances
  - You may extend the pattern to provide more instances and controlled accesses to them.

Summary and Advice

- Behavioral pattern: Template Method
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
  - A fundamental technique for code reuse in class libraries
- Creational pattern: Singleton
  - Ensure a class only has one instance, and provide a global point of access to it
- Creational pattern: Prototypes
  - Hide part types for parts creation
- Try to hide implementations and to enforce rules as much as possible.