ECE 449/590 – OOP and Machine Learning
Lecture 09 The Builder Pattern

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

September 21, 2022

Outline

Reading Assignment

- This lecture: The Builder Pattern
- Next lecture: Accelerated C++ 9

Outline

Design Patterns

The Builder Pattern

Opaque Pointer

Class Library Design

- Reusable
  - The library should be reused without being modified.
- Polymorphism
  - Enable library designers to design classes and programs that can work with any class satisfying certain constraints
  - Runtime polymorphism: duck typing in Python, interface/inheritance in C++/Java.
  - Compile-time polymorphism: C++ templates and library algorithms.

Design Patterns

- Common OOD/OOP practices to solve software design problems.
  - How should we use the language features like interface and inheritance effectively.
- Learn design experiences from experts
  - Design patterns are solutions that are applied routinely.
  - Design patterns are independent of programming languages.
- Enable effective communication between designers
  - Design patterns are technical jargons for designers.
  - Allow you to quickly understand how a big piece of software is organized.
  - Allow others to quickly comprehend your design idea.

More Details

- Categories of design patterns
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- For our EasyNN library,
  - How to pass DAG from Python to C++ code?
  - How to support multiple implementations of DAG computations?
  - How to implement different operations?
  - In a way that can be easily extended (not modified)?

Outline

The Design Problem

```
class Expr:
    def __init__(self, op, inputs):
        self.op = op
        self.inputs = inputs
        ...
```

- The EasyNN DAG is captured in Python as expressions.
  - `self.op` stores what operation should be executed.
  - `self.inputs` refers to the expressions that generate inputs for this expression.
- While you may evalute the DAG just use this data structure, it could be benefitial to create new data structures dedicated for evaluation.
  - Avoid the need to do recursion.
  - Evaluate in a different language, or even using special hardware like GPU and FPGA.
- Or, can we reuse our DAG evalution methods for other machine and deep learning libraries?

The Builder Pattern

- A creational pattern
- Separate the construction of a complex object (the EasyNN DAG) from its representation (various data structures for DAG evaluation)
  - Builder: an abstract interface for creating the complex object from its parts
  - Director: construct the complex object using the Builder interface

The Builder Interface

```
class Builder:
    def append(self, expr):
        ...

    def build(self):
        ...
```

- An abstract interface without any implementation
  - Define steps to construct the complex object from its parts independent of the specification
  - Since duck typing is used for polymorphism in Python, there is no need to actually define an interface – two methods with specific names are required for any class to work as an EasyNN builder.
- How the complex object is constructed, is not specified.
  - Even the classes for the complex object and the parts are not defined – leaving great flexibility.
  - In other words, only the responsibility itself is specified.
- Assume errors are handled through exceptions.

The Director Implementation

```
class Expr:
    ...
    def compile(self, builder):
        self.__dfs_post({}, lambda that: builder.append(that))
        return builder.build()
```

- The director follows the specification to build the complex object through the builder interface.
  - Without any knowledge of the complex object
- For EasyNN, the director performs depth-first search to call `append` in the `builder`, and call `build` to finalize the building process.
  - `builder` is able to process expressions in the topological order they should be evaluated, where inputs are always ready.
- Program for the interface but not the implementation.

Implement a Builder

```python
# easynn_golden.py
class Builder:
    def __init__(self):
        self.program = []
    def append(self, expr):
        self.program.append(expr)
    def build(self):
        return Eval(self.program)
```

- To implement the builder, we need to implement the Builder interface.
  - Python uses duck typing for polymorphism so there is no need to define a base class first for the interface – it is sufficient to define member functions with the desired names.
- For the golden implementation of EasyNN in NumPy,
  - In `append`, store all expressions in the topological order.
  - In `build`, return an `Eval` object to take care of evaluation.

Implement Another Builder

```python
# easynn_cpp.py
class Builder:
    def __init__(self):
        self.program = _libeasynn.create_program()
    def append(self, expr):
        ...
        _libeasynn.append_expression(...)
        for k, v in op.parameters.items():
            ...
            _libeasynn.add_op_param_double(...)
    def build(self):
        return Eval(_libeasynn.build(self.program))
```

- For the C++ implementation of EasyNN,
  - Both `append` and `build` call corresponding functions from the shared library `libeasynn.so`.
  - An additional function `add_op_param_double` is introduced to the shared library to build the operator.

Put Everything Together

```
def is_same(p, n, *args):
    e0 = p.compile(cpp.Builder())
    e1 = p.compile(golden.Builder())
    ...
```

- Starting with the same EasyNN DAG `p`, create two objects for evaluation using two builders.
  - Then they can be evaluated and compared to see if your C++ implementation is correct.

Summary of Participants of the Builder Pattern

- Builder
  - Specifies an abstract interface for creating a complex object from its parts
- Concrete Builder
  - Constructs and assembles parts of the product object by implementing the Builder interface
  - Provides means for retrieving the product object and/or finalizing the creation
- Director
  - Constructs a complex object using the Builder interface
  - Can be a class to handle more complicated creation process
- Product
  - Represent the complex object under construction
  - Details are revealed to and only to Concrete Builder for creation.

Benefits of the Builder Pattern

- It lets you vary a product's internal representation.
  - The internal representation of the product, i.e. the product types, together with the method to assemble it, is hidden from the director.
  - All you have to do to change the product's internal representation is to define a new kind of Concrete Builder.
- It isolates code for construction and representation.
  - Code for creation from the specification is centralized in Director.
  - Product types are no longer responsible for creation – it's now the responsibility of Concrete Builder.
  - Director and Concrete Builder can change independently.
- It gives you finer control over the construction process.
  - The product is constructed step by step under the director's control.

Outline

Design Patterns

The Builder Pattern

Opaque Pointer

Opaque Pointer

- A method to provide abstraction and encapsulation.
  - And potentially polymorphism.
- Allow to write code following OOP principles in languages not supporting OOP directly.
  - E.g. `FILE *` in C may refer to files, network sockets, IPC pipes, devices, etc.
  - This helps EasyNN as we need to fall back to C to bridge Python and C++ code.
- Allow to hide implementions completely in languages supporting OOP.
  - As a comparison, you may still see class members from class header files.

The EasyNN Shared Library Interface

```
// libeasynn.h
...
extern "C" program *create_program();
extern "C" void append_expression(program *prog, ....);
extern "C" int add_op_param_double(program *prog, ....);
extern "C" evaluation *build(program *prog);
extern "C" void add_kwargs_double(evaluation *eval, ....);
extern "C" int execute(evaluation *eval, ....);
```

- Programs that need to access the EasyNN shared library only need to use information within `libeasynn.h`
- `program *` and `evaluation *` are opaque pointers.
  - `libeasynn.h` and any header files included by it do not provide definition of these two types.
- But how could one use a type without first defining it in C++?

Forward Declarations

```
// libeasynn.h
class program; // forward declaration
class evaluation; // forward declaration

extern "C" program *create_program();
extern "C" void append_expression(program *prog, ....);
extern "C" int add_op_param_double(program *prog, ....);
extern "C" evaluation *build(program *prog);
...
```

- Since `program *` and `evaluation *` are pointers, compilers just need to deal with addresses of the objects.
  - There is no need for the compiler to know the details of the objects as long as you don't need to access their members.
  - Actually you are NOT supposed to access their members directly – you are only allowed to call those functions.
- We just need to tell the compiler these are two class types.
  - Through forward declarations.

Using Opaque Pointers

```
// easynn_test.cpp
#include "src/libeasynn.h"
int main() {
    program *prog = create_program();

    int inputs0[] = {};
    append_expression(prog, 0, "a", "Input", inputs0, 0);

    int inputs1[] = {0, 0};
    append_expression(prog, 1, "", "Add", inputs1, 2);

    evaluation *eval = build(prog);
    ...
}
```

- The `main` function uses `program *` and `evaluation *` without knowing any details about `program` and `evaluation`.
- This provides another example of Director in the Builder pattern.

The EasyNN Shared Library Implementation

```
// libeasynn.cpp
...
#include "libeasynn.h"
#include "program.h"
#include "evaluation.h"

program *create_program() {
    program *prog = new program;
    printf("program %p\n", prog);
    return prog;
}

void append_expression(program *prog, ....) {
    ...
    prog->append_expression(....);
}
```

- When implementing functions using opaque pointers, it is necessary to `include` definitions of those types.
  - So the compiler is able to access members.
- Note that these implementations are always hidden from the users that only use these functions.

Creation

```
// libeasynn.cpp
program *create_program() {
    program *prog = new program;
    ...
}

// easynn_test.cpp
#include "src/libeasynn.h"
int main() {
    program *prog = create_program();
    ...
}
```

- There must be a function to create an opaque pointer for a user to make use of it.
  - Concepturally similar to constructors – though constructors cannot be used by users since all details are hidden.
  - Sometimes people call this function a factory.
- Usually the objects will be created on the heap.
  - So a user may use opaque pointers to manage multiple such objects.

No delete?

- You probably realized we never `delete` those opaque pointers in our EasyNN shared library implemention.
  - This is indeed an issue while we do so just for simplicity.
- There should be functions allowing users to destroy objects pointed by opaque pointers.
  - Users should not just `delete` those pointers by themselves since there is no guarantee the objects are created using a compatible `new`.
  - C++ compilers won't allow to `delete` opaque pointers anyway.
- For EasyNN, if such functions are available, you'll need to understand Python GC to call them at correct places.

Summary and Advice

- Creational pattern: Builder
  - Separate system specification from system creation
- Opaque pointers hide details to achieve abstraction, encapsulation, and polymorphism.