

Extensible file format

Lecture 18

Input processing and Plug-in

There is one missing bit of our program so far: we must modify the program itself in order to play a different sound.

The missing bit is input. We will show how we can handle input, and how input can drive a plug-in architecture that allows maximum flexibility.

References:

- Design Patterns: Factory.
- Man pages of dlopen(3).
- Reusable C++: Section 12.1, The Static Initialization Problem.

A program with a very extensible architecture is of little use if the input itself is not extensible.

Suppose we want our program to read the following type of file:

```
Composer (
  TimeLimiter SineNote C3 0.1 2.5
  0
  TimeLimiter SineNote E3 0.1 2.2
  0.5
)
SineNote 1000 0.1
```

Note that it is easy to add the support of a new type of notes in such a file.

A side note: other choices exists, e.g., XML. We use a simpler file format for illustration.

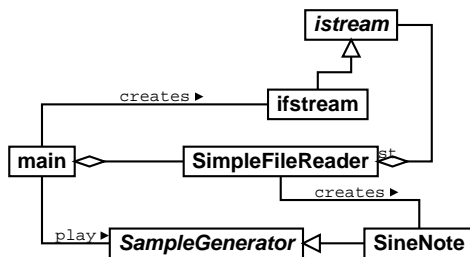
PMOOP(0396A)

PMOOP(0396A)-18.1

The architecture

One possibility is to have an object reading such file. The main program will get an object for input, and pass the object to the object.

We thus have the following class structure:



PMOOP(0396A)-18.2

PMOOP(0396A)-18.3

Main program modification

The main program thus create a SimpleFileReader and get sample generators from it.

```
int main() {
  SoundOutput *so;
  ...
  SimpleFileReader sfr(argv[1]);
  ...
  while (SampleGenerator *gen = sfr.get_generator())
    do so->write_sample(gen->get_sample());
  while (gen->advance(1.0/so->get_sample_rate()));
}
```

We will thus run our program like this:

```
> note2wav testfile test.wav
```

One way to read such file

The object can parse the object like this: (CVS: lecture-18-a)

```
SampleGenerator *SimpleFileReader::get_generator() {
  string type;
  *st >> type; // st is a istream* field of SimpleFileReader
  if (!*st) return 0;
  SampleGenerator *ret;
  if (type == "SineNote") {
    double freq = read_freq(*st), amp;
    *st >> amp;
    if (!*st) throw FileError("SineNote: missing amplitude");
    return new SineNote(freq, amp);
  } else if ... // For other types
}
```

However, this makes it difficult to extend the program. To add a new type we must modify SimpleFileReader::get_generator().

Make it extensible: how we know the type?

Suppose we want the program to load new code at run-time to deal with new generators.

The problem of the approach is that it uses **code** (i.e., if statements) to select code. We want to use **data** to select code instead.

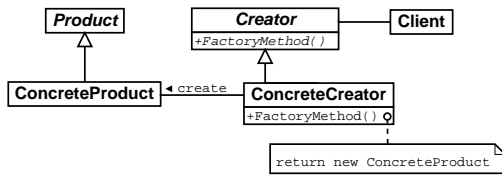
- The program can modify data easily, but not code.
- The data structure can say that "if the next word is "SineNote", then create a SineNote." To add a new type, we add a new entry.
- The if part is easy: we store a map which store the key "SineNote".
- But how to do the then part? The data structure must store "something" that says "create a SineNote".
- In other words, **we want to delay the decision to create a SineNote** (rather than other type of SampleGenerator) **until run-time**.

PMOOP(0396A)-18.4

PMOOP(0396A)-18.5

Using a factory to deal with input

Now we turn to our pattern book, and search for something that would work. A good choice is a **factory**:



The SimpleFileReader acts as the client, which has a few ConcreteCreator's to create sound generators of different types.

All of them are of type Creator (SFRFactory). They can create a sub-type of Product (SampleGenerator) of its choice.

PMOOP(0396A)-18.6

Resulting code for file reader

Now the get_generator() function is really general.

```

class SFRFactory {
public:
    virtual SampleGenerator *get(istream &, SimpleFileReader *) = 0;
};

std::map<string, SFRFactory *>gadget_factories;

SampleGenerator *SimpleFileReader::get_generator() {
    string type;
    *st >> type;
    if (!*st) return 0;
    if (SFRFactory *f = gadget_factories[type])
        return f->get(*st, this);
    throw FileError(string("Unknown generator ") + type);
}
  
```

PMOOP(0396A)-18.7

Factory code

It is easy to provide a factory:

```

class SFRSineNoteFactory: public SFRFactory {
    SampleGenerator *get(istream &, SimpleFileReader *);
};

SampleGenerator *SFRSineNoteFactory::get
(istream &st, SimpleFileReader *) {
    double freq = read_freq(st), amp;
    st >> amp;
    if (!st)
        throw FileError("SineNote: missing amplitude");
    return new SineNote(freq, amp);
}
  
```

PMOOP(0396A)-18.8

Registering the factories

The only remaining thing to do is to add it to the gadget_factories map.

We do it like this (CVS: lecture-18-b):

```

void SFR_register_SineNote_factory() {
    gadget_factories["SineNote"] = new SFRSineNoteFactory;
}

int main(int argc, char *argv[]) {
    ...
    SFR_register_SineNote_factory();
    SFR_register_SquareNote_factory();
    ...
    SimpleFileReader sfr(argv[1]);
    ...
}
  
```

PMOOP(0396A)-18.9

Extensibility through plug-ins: benefits

One question remains: why we want such flexibility?

The program **still have to compile in all the possible factories**, so at run-time we have a fixed set of factories!

- The answer is, the program really don't have to compile in all the factories.
- Instead, the program can **load the functions for a class at run-time**.
- We call a file storing the class a **plug-in**, or a **loadable module**.
- Therefore, we will try to look at the gadget_factories map, and use it if an entry is found. Otherwise we will try to load it from a plug-in.
- Benefit: we can support additional type **without recompiling!**
- This is an advanced feature of Unix used by apache, Mozilla, gimp, etc.

PMOOP(0396A)-18.10

Shared library as plug-ins

What is a plug-in in Unix, then?

- Plug-ins in Unix is called "Shared Library".
- They are very similar to programs: you create them by linking together some .o files and some libraries.
- You will add a flag "-shared" (or "-G" in Solaris) to tell the compiler that you want to create a shared library.
- The main program can link-in a shared library using a call dlopen(), specifying the path to load. It returns a "handle", a pointer used for further operation on the library.
- If there is any error, the returned handle is NULL, and dlerror() shows the error.

PMOOP(0396A)-18.11

The new SimpleFileReader

```
SampleGenerator *SimpleFileReader::get_generator() {
    string type;
    *st >> type;
    if (!st) return 0;
    if (SFRFactory *f = gadget_factories[type])
        return f->get(*st, this);
    string str = string(".") + "/"SFR" + type + "Factory.so";
    if (!dlopen(str.c_str(), RTLD_NOW)) // Resolve symbols NOW
        throw FileError(string("Can't load module: ") + derror());
    if (SFRFactory *f = gadget_factories[type])
        return f->get(*st, this);
    throw FileError(string("Unknown generator ") + type);
}
```

The code tries to load the plug-in if the map returns NULL. If that fails, or if the map still returns NULL, we signify error.

PMOOP(0396A)-18.12

How the plug-in modify the program?

How the plug-in actually modify the behaviour of the program? We still need to add the mapping from "SineNote" to the SFRSineNoteFactory somewhere!

The answer is: an init object. For example, in SFRSineNoteFactory (CVS: lecture-18-c):

```
class InitObj {
public:
    InitObj() {
        gadget_factories["SineNote"] = new SFRSineNoteFactory;
    }
};
static InitObj initobj;
```

When we load the library, all static object are created, which call constructors. The constructor is written specifically to make the mapping.

PMOOP(0396A)-18.13

Conclusion and Exercise

To actually get most of the benefit of an extensible program, we need an input format that is extensible.

By using a factory, we can delay the decision about what object to create to run-time.

This allow us to dlopen() to support a very flexible plug-in architecture, so that the program can be extended without recompilation.

Exercise:

Read the CVS source tree of this lecture to understand how the program is actually implemented.

Modify your Lisp parser to support dynamic loading of modules to support new forms or functions.

Add a function LoadPlugin that actually load such a plug-in. Try it out by moving some of the functions to plug-ins.

PMOOP(0396A)-18.14