

Original code, extensibility requirement

Lecture 15

Sound sample generation and decorators

- We go back to our sound creation project, focusing on what kind of sound the program should be capable of generating.
- We will build classes that generate sound. By sub-classing, we can modify the behaviour of one class. We'll learn a new way that modify the behaviour of all classes.

References:

- Design patterns: Section 1.6, "Putting Reuse Mechanisms to Work" (pp. 18–22).
- Design patterns: Decorator (pp. 175–184).

PMOOP(0396A)

So far, our audio project is generating very simple sound: 1kHz sine wave. This is done by the following code:

```
for (int i = 0; i < rate; ++i) {
    so->write_sample(sin(2 * M_PI * t * 1000) * 0.25);
    t += 1.0/rate;
}
```

We want to change this. In particular, we want our program to:

- Make a representation of the sound to be generated, e.g., "square-wave of 2kHz of infinite length" or "sine wave of the middle-C note for 0.5s".
- The main program then just ask that representation to generate the sound. Calculation of the sample should be the responsibility of that representation.
- It should be easy to support new type of sound: extensible.

PMOOP(0396A)-15.1

Abstraction: how we view sample generation

- Our previous experience shows that we will need an abstract class that can generate sound samples. Let's call it SampleGenerator.
- The first question to ask: what's the interface of the class? Or, equivalently, how we will use the class?
- The interface should be **general** enough to be suitable for all sound, but **complete** enough for our use.
- Some considerations:
 1. How the sample generator know the sample frequency?
 2. How the sample generator know the amplitude?
 3. Should the user know the length of the sound?
 4. What is the relation between SampleGenerator and SoundOutput?

PMOOP(0396A)-15.2

The design

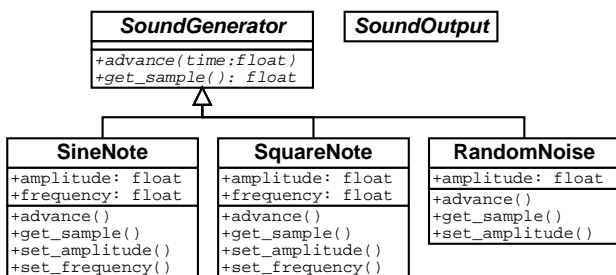
SoundGenerator has the following interface:

- A SampleGenerator represents the status of any kind of sound generating device at the current time.
- The current time is changed by advance(x), which go forward by x second. The function returns false if no more sound can be expected.
- The current audio level is returned by the get_sample() function.
- Sound parameters like frequency and amplitude are defined by concrete sub-classes of SampleGenerator, e.g., SineNote, SquareNote.
- SampleGenerator and SoundOutput are largely independent. As we will see soon, this allows maximum possibility of reuse.

Note that we provide a **very general framework** that should **work for every extension**: key idea of extensibility.

PMOOP(0396A)-15.3

Class diagram



The sound generation user code thus look like this:

```
SoundGenerator *gen = build_generator(); // Build the representation
do so->write_sample(gen->get_sample());
while (gen->advance(1.0/so->get_sample_rate()));
```

PMOOP(0396A)-15.4

Abstract class

The implementation of the abstract class is very simple:

```
#ifndef _SAMPLE_GENERATOR_H
#define _SAMPLE_GENERATOR_H
```

```
class SampleGenerator {
public:
    SampleGenerator() {}
    virtual ~SampleGenerator() {}
    virtual bool advance(float t) = 0;
    virtual float get_sample() = 0;
};
```

#endif

Note that it resembles the class diagram very much. In fact there are tools to let you draw class diagrams and to generate code from there.

PMOOP(0396A)-15.5

Implementation: sine wave that sounds infinitely

```
class SineNote: public SampleGenerator {
public:
    SineNote(float f, float a): freq(f), amp(a) {}
    void set_freq(float f) { freq = f; }
    void set_amp(float a) { amp = a; }
    bool advance(float t) { time += t; return true; }
    float get_sample()
        { return sinf(2 * M_PI * time * freq) * amp; }
private:
    float freq, amp, time;
};

In main program:

SampleGenerator *build_generator() {
    return new SineNote(1000, 0.25);
}
```

PMOOP(0396A)-15.6

Implementation: square wave

Once we have the infra-structure there, other types of notes can be easily made.

```
class SquareNote: public SampleGenerator {
public:
    SquareNote(float f, float a): freq(f), amp(a), time(0) {}
    void set_freq(int f) { freq = f; }
    void set_amp(float a) { amp = a; }
    bool advance(float t) { time += t; return true; }
    float get_sample() {
        if (int(2 * time * freq) % 2 == 0) return amp;
        else return -amp;
    }
private:
    float freq, amp, time;
};
```

PMOOP(0396A)-15.7

Implementation: random wave

Also, largely unrelated classes can be made, since we have a rather general abstract class SampleGenerator.

```
class RandomNoise: public SampleGenerator {
public:
    RandomNoise(float a): amp(a) {}
    void set_amp(float a) { amp = a; }
    bool advance(float t) { return true; }
    float get_sample() {
        return (float(rand())/RAND_MAX-0.5) * 2 * amp;
    }
private:
    float amp;
};
```

PMOOP(0396A)-15.8

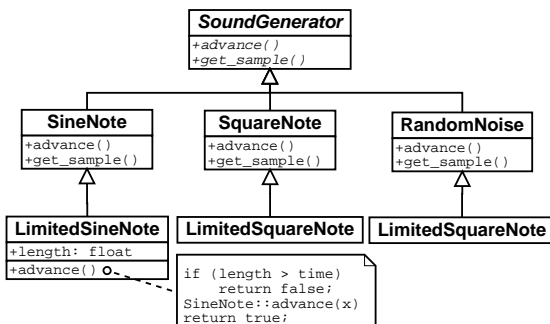
Limiting the length: requirement

- Note that all the generators that we wrote make sound forever: they never stop.
- Our specification allows generators to be written that know when to stop. If a generator stop, it should return *false* during advance().
- Suppose we want both versions for each possible generator, i.e., one version that will stop, and one version that will never stop.
- We should of course be able to specify when to stop, i.e., how long the generator sounds.
- To avoid duplicating all the code for SineNote, SquareNote, RandomNote, etc., we can re-use such a note, until we find that the time expires and thus we “cut the air”.
- But how to do that?

PMOOP(0396A)-15.9

Design 1: subclass design

Beginning OOP coders tends to make a sub-class for each of the SineNote, SquareNote and RandomNoise classes:



PMOOP(0396A)-15.10

Comments on this design

There are some good things about this design:

- Code in SineNote, SquareNote and RandomNoise are not duplicated.
- This is a very natural design: A time-limited SineNote is a special type of SineNote that make sure length is not exceeded.

But there are more bad things than good things about this design:

- Code in Limited-classes are basically duplicates of each other.
- It encourages a tall class hierarchy.
- It is not very scalable: if we have *m* basic classes and *n* types of limiting, we end up having *mn* different combinations and thus classes.
- It creates more work when new notes are added. In particular, everytime we add a new type of note we will need a Limited-version.

PMOOP(0396A)-15.11

Rethink the problem: delegation design

Let's think about our problem more carefully. What we want:

- For each class of SoundGenerator, we should be able to make an object of that class, with all the behaviour of that class, except that we can specify when it stops.
- That object should still be of a sub-class of SoundGenerator.
- We want a single class, say TimeLimiter, to do that.

Note that we don't really need the class to be a sub-class of any existing class other than SoundGenerator.

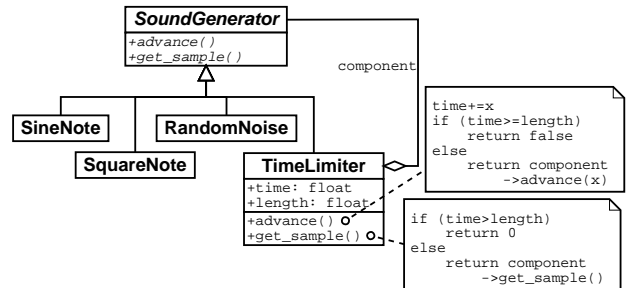
In particular, we don't need to derive from SineNote. But how we can use the functions in SineNote, then?

Solution: **aggregation** and **delegation**. If TimeLimiter has-a SineNote, then it can use the SineNote code by just calling it.

PMOOP(0396A)-15.12

Design 2: delegation design

Now it is easy to see that we don't really need to commit on SineNote: we just need to say that TimeLimiter has-a SoundGenerator:



We say TimeLimiter has a SoundGenerator (aggregation), and its operations are done by delegating to its SoundGenerator component.

PMOOP(0396A)-15.13

Implementation: Limiter

Once we have the class diagram, implementation is easy.

```
class TimeLimiter: public SampleGenerator {
public:
    TimeLimiter(SampleGenerator *sg, float len)
        : component(sg), time(0), length(len) {}
    ~TimeLimiter() { delete component; }
    bool advance(float t) {
        time += t;
        return
    }
    float get_sample()
private:
    SampleGenerator *component;
    float time, length;
};
```

PMOOP(0396A)-15.14

Making the Timelimiter

Now we can build a time-limited version of SineNote this way:

```
SampleGenerator *build_generator() {
    return
}
```

If we instead need a time-limited version of SquareNote, do this:

```
SampleGenerator *build_generator() {
    return
}
```

Note that we don't need to create a class for each of them: we only need to build them appropriately!

Since we don't need to create new classes, we don't have any of the problems of the sub-class design.

PMOOP(0396A)-15.15

Recurring concepts of patterns: Delegations

When designing object-oriented systems, we usually have the opportunity to choose between inheritance and aggregation.

Inheritance has the benefit of

- Directly supported by the language, easy to understand.
- Allow the derived class to modify the behaviour of the base class.
- It is very efficient.

However, there is one **big** cost:

- The derived class is committed to the base class at compile time, and it cannot be changed at run-time.

When this cost is not acceptable, use aggregation and delegation instead. In practice, this means to avoid sub-classes whenever appropriate.

PMOOP(0396A)-15.16

The Decorator pattern

In our solution:

- The model is twisted so that a TimeLimiter has-a, instead of is-a, SineNote—or any other other kind of SampleGenerator.
- We use exactly the same SampleGenerator interface for TimeLimiter, so that a TimeLimiter can be used as a SampleGenerator like a SineNote.
- Thus TimeLimiter has a SampleGenerator, as well as is-a SampleGenerator.
- TimeLimiter adds the capability of length management into any possible SampleGenerator: it decorates any SampleGenerator with a new capability.
- The capability is added to it when we make the TimeLimiter, at run-time.

This is another common pattern in Object-oriented design: **Decorator**.

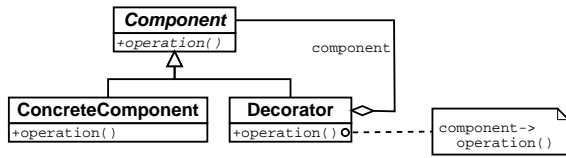
PMOOP(0396A)-15.17

Applicability and structure of Decorators

Applicability: use decorators

- To add responsibility to individual objects dynamically and transparently.
- To add responsibility that can be withdrawn.
- When there are many combination of responsibility to make sub-classing impractical.

Structure



PMOOP(0396A)-15.18

Decorators: consequences

Good things about decorators

- *More flexibility* than inheritance. E.g., we can modify TimeLimiter so that we can change our minds and get back the underlying component.
- *Avoid very feature-rich classes.* Decorators allow features to be added to components only when they are needed. The alternative, i.e., add all possible features into a class, results in large class that is difficult to understand and expensive to use.

Bad things about decorators

- *More small-objects to create.* It usually results in many more objects, each taking up a small responsibility. This can be more difficult to learn and debug.
- *One more level of indirection.* Of course, this means one more dereference to do when executing the methods there.

PMOOP(0396A)-15.19

Conclusion and Exercise

- We design the interface of a class (i.e., define the abstraction) by considering how the class is used and generalized.
- Class diagrams allow us to describe designs at a higher level, avoiding language details.
- It is usually better to use aggregations and delegations instead of sub-classing and inheritance.
- Decorators allow features to be added to a class dynamically.

Exercise:

Make a new decorator, SoftTimeLimiter, that is similar to TimeLimiter, but stops sound by reducing sound level from full to 0 within a period of t seconds, where t is specified when the SoftTimeLimiter object is created.

Group the management of the component in the two decorators to a common abstract super-class of them. (Read the book!)

PMOOP(0396A)-15.20