

Composer: all the notes

Lecture 16 Composing notes

- Our program can now output single notes, of various properties.
- We see how we can do this by building on what we already have, and see how this gives rise to an object hierarchy.

References:

- Design Patterns pp. 163–173: Composite pattern.
- Design Patterns Section 1.6: Relating Run-Time and Compile-Time Structures.
- Design Patterns pp. 219–220: Discussion of Structural Patterns, Composite versus Decorator versus Proxy.

PMOOP(0396A)

We finally comes to the interesting bit of the project: making it possible to play a song containing some notes.

- We have actually built a very flexible architecture, so it should be possible to model things as complicated as a song.
- The only missing thing is a class that really do this.
- Do we need to write the notes generating code again?
- If we have really learnt the lesson from Decorators, we know the answer is no. Instead we build from what we already have.
- That is, we re-use the code that we have written to generate a single notes.
- We build a class that manage some notes while adopting exactly the same interface as specified by SampleGenerator.

PMOOP(0396A)-16.1

What we need

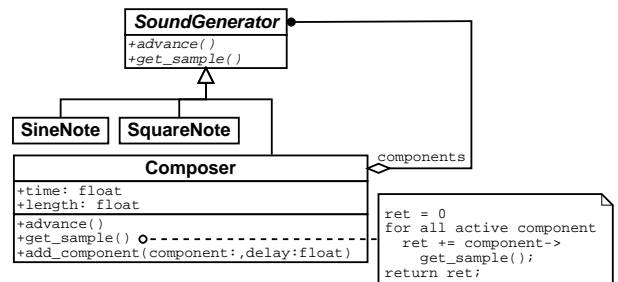
Suppose we want to create a class Composer for making song.

- We should be able to add some notes (at least, SineNote) to the Composer. But why stop at SineNote?
- Instead, let's define Composer to contain some SampleGenerators that is added to it using a function add_component.
- It would not be very interesting if all generators make a sound at the same time. Nor will it be very flexible if generators can only sound one after another without pause.
- Instead we want each component to have a delay, and start generating sound only after that delay.
- Therefore, add_component also need another argument specifying the delay.

PMOOP(0396A)-16.2

Design

Now we know what is the needed class structure:



The filled circle denote that a Composer can have many SoundGenerators. Another way to show this (in UML) is to write a "*" symbol above the connection point of SoundGenerator.

PMOOP(0396A)-16.3

How to deal with delay?

Let's have the following simple design:

- When a generator is added using add_component, we add it to the set of available notes.
- Every time we advance, we repeatedly find the note from the set of available notes which has the least delay, and see whether the delay is passed.
- For each note with the delay passed, we add it to a set of active components.
- Then we advance all active components.
- Once a note return false on advance, it is discarded from the Composer. The Composer itself return false when all notes have been discarded.

PMOOP(0396A)-16.4

On data structures

- What data structure we should use to store the available notes?

We need the operations:

So we use

- What data structure we should use to store the active notes?

We need the operations:

So we use

PMOOP(0396A)-16.5

Interface

```
#include "SampleGenerator.h"
#include <set>

class Composer: public SampleGenerator {
public:
    Composer();
    ~Composer();
    bool advance(float t);
    void add_component(SampleGenerator *, float t);
    float get_sample();
private:
    class queue;
    queue *components;
    std::set<SampleGenerator *> active_components;
    float time;
};
```

PMOOP(0396A)-16.6

Implementation: the priority-queue

```
#include <queue>
using namespace std;

struct ComposerRecord {
    ComposerRecord(SampleGenerator *g, float d)
        : gen(g), delay(d) {}
    SampleGenerator *gen;
    float delay;
    // Higher priority for generators with less delay
    bool operator< (const ComposerRecord &r2) const {
        return delay > r2.delay;
    }
};

class Composer::queue: public priority_queue<ComposerRecord> {};
```

PMOOP(0396A)-16.7

Implementation: creation and destruction

```
Composer::Composer(): components(new queue) {}

Composer::~Composer() {
    while (!components->empty()) {
        delete components->top().gen;
        components->pop();
    }
    delete components;
    std::set<SampleGenerator *>::iterator i;
    while ((i=active_components.begin()) != active_components.end()) {
        delete *i;
        active_components.erase(*i);
    }
}
```

PMOOP(0396A)-16.8

Implementation: adding generator, getting sample

```
void Composer::add_component(SampleGenerator *g, float t) {
    components->push(ComposerRecord(g, t));
}

float Composer::get_sample() {
    float ret = 0;
    for (std::set<SampleGenerator *>::iterator
         i = active_components.begin();
         i != active_components.end();
         ++i)
        ret += (*i)->get_sample();
    return ret;
}
```

PMOOP(0396A)-16.9

Implementation: advancing time

```
bool Composer::advance(float t) {
    time += t;
    while (!components->empty() && components->top().delay < time) {
        SampleGenerator *gen = components->top().gen;
        gen->advance(time - components->top().delay);
        active_components.insert(gen);
        components->pop();
    }
    for (std::set<SampleGenerator *>::iterator
         i = active_components.begin();
         i != active_components.end(); ++i)
        if (!(*i)->advance(t)) {
            delete *i;
            active_components.erase(i);
        }
    return !components->empty() || !active_components.empty();
}
```

PMOOP(0396A)-16.10

Using composer: build_generator()

```
float freq_tab[] = {
    261.63, 277.18, 293.66, 311.13, 329.63,
    349.23, 369.99, 392.00, 415.30, 440.00, 466.16, 493.88, 523.25
};

SampleGenerator *build_generator() {
    Composer *ret = new Composer;
    ret->add_component(new TimeLimiter(
        new SineNote(freq_tab[0], 0.25), 2.5), 0);
    ret->add_component(new TimeLimiter(
        new SineNote(freq_tab[4], 0.25), 2.2), 0.5);
    ret->add_component(new TimeLimiter(
        new SineNote(freq_tab[7], 0.25), 2), 1.0);
    ret->add_component(new TimeLimiter(
        new SineNote(freq_tab[12], 0.24), 1.5), 1.5);
    return ret;
}
```

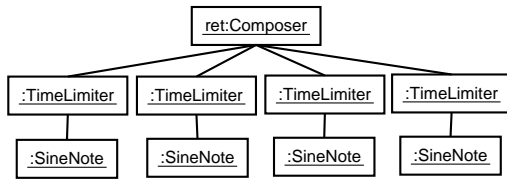
PMOOP(0396A)-16.11

Object diagram

Note that the object returned by the previous `build_generator` is quite big:

- It contains four Limiters.
- Each Limiter contains a SineNote.

So in total 9 objects are in play:



We usually call this the object diagram. Since this forms a hierarchy, we can call this the object hierarchy of the `Composer` returned.

PMOOP(0396A)-16.12

Class diagram vs. Object diagram

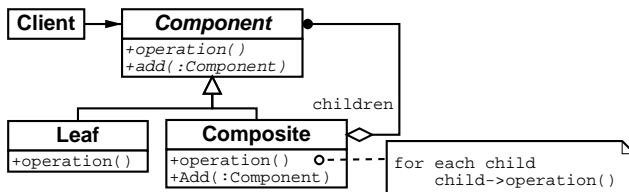
We now have two types of diagrams: class diagrams and object diagrams.

- Class diagrams captures the compile-time structure of your program.
- Object diagrams capture the run-time structure of your program.
- Object diagrams usually becomes interesting when the class is defined recursively. E.g., `Composer` is a `SampleGenerator` that can have some `SampleGenerator`.
- There can be many object of the same class, and a class may even contains an object of the same class!
- The two different diagrams is largely independent. It is impossible to derive complete information of one diagram from information of the other.
- Therefore, the two diagrams are complementary.

PMOOP(0396A)-16.13

The composite pattern

Our `Composer` is similar to the Composite pattern, which has the following structure:



It is in fact very similar to the Decorator pattern.

The difference is more on the intent: Decorator focuses on **adding new functionalities**, while Composite focuses on **how to combine different object**.

PMOOP(0396A)-16.14

Deviation from pattern

We have taken an approach that deviate from the Composite pattern: we define `add_component` operation in `Composer`, not in `SampleGenerator`.

- This means that `Composer` is not transparent. The client need to know that a `SampleGenerator` is a `Composer` to add things to it.
- This also means that `SampleGenerator` is more safe. Without knowing that an object is a `Composer`, there is no way to add things to it.

Our choice is base on that **the add operation** that include a delay parameter is probably **only useful for our Composer**. Clients really need to know about `Composer` to add things to it, so transparency problem is minimal.

When using patterns, we will stick to the patterns as far as reasonable, since that makes it easy for others to know what's going on.

But we can deviate from a pattern if there is compelling reasons.

PMOOP(0396A)-16.15

Consequences of composites

The goods:

- The class hierarchy contains both primitives and composites, so whenever a primitive object is expected, we can also use a composite.
- It is easy to add new components: whenever a new `Leaf` or `Composite` is defined, it automatically works with the `Composites` already there.
- (Only for "real" composites): Clients can be simpler because it can treat `Leaf` and `Composite` uniformly.

The bad:

- It may be overly general: it is difficult to restrict that a `Composite` cannot contain a certain type of `Component`.
- (For "real" composites): It delays some programming error to until run-time, since the sacrifice some type-safety.

PMOOP(0396A)-16.16

Conclusion and exercise

- By employing the techniques of aggregates, we can re-use an interface while allowing more complex objects to be built.
- We choose data structure to implement a pattern (and in general, to implement programs) by considering the operations needed.
- Composites defines classes recursively. For such classes we usually ends up in a rather complicated object hierarchy that is independent from the class hierarchy.

Exercise Suppose we modify our implementation so that `Composer` has no delay, and make a new `Delayer` class to provide the functionality.

1. Will it make sense to use a real `Composite`?
2. Design the system. Put the new design into code.

PMOOP(0396A)-16.17