Interfaces
Writing your own

Alternative version using decorator, strategy and composition

File objects for various applications

Let’s say we’d want a class representing files for e.g. a file browser. We’ll call the class FBFile and it should have two public methods:

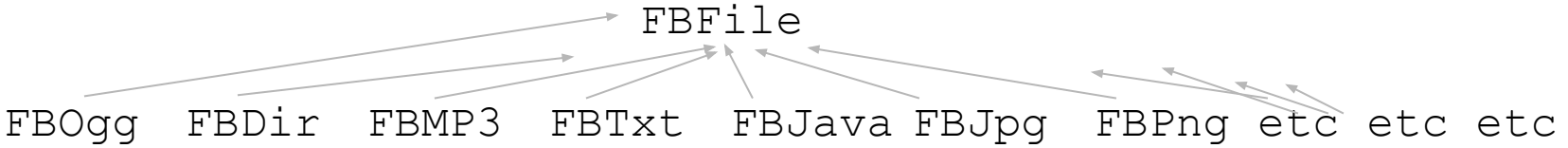
`String name()` (name of the file)

`String thumbnail()` (a string representation of a thumbnail, e.g. `[txt]`)

We’d also like to be able to re-use this class for other applications, like a media player (which also handles files, the user can add files and play them).

Problem with inheritance

If we'd opt to use inheritance for all kinds of files our applications should handle, we'd end up with an insane class explosion:



Some of the above could implement an interface `Playable` with a `play()` method, and all of them should override the `thumbnail()` method to have unique thumbnails.

Everytime we'd like to support a new file type, we'd need a new subclass...

Our first strategy

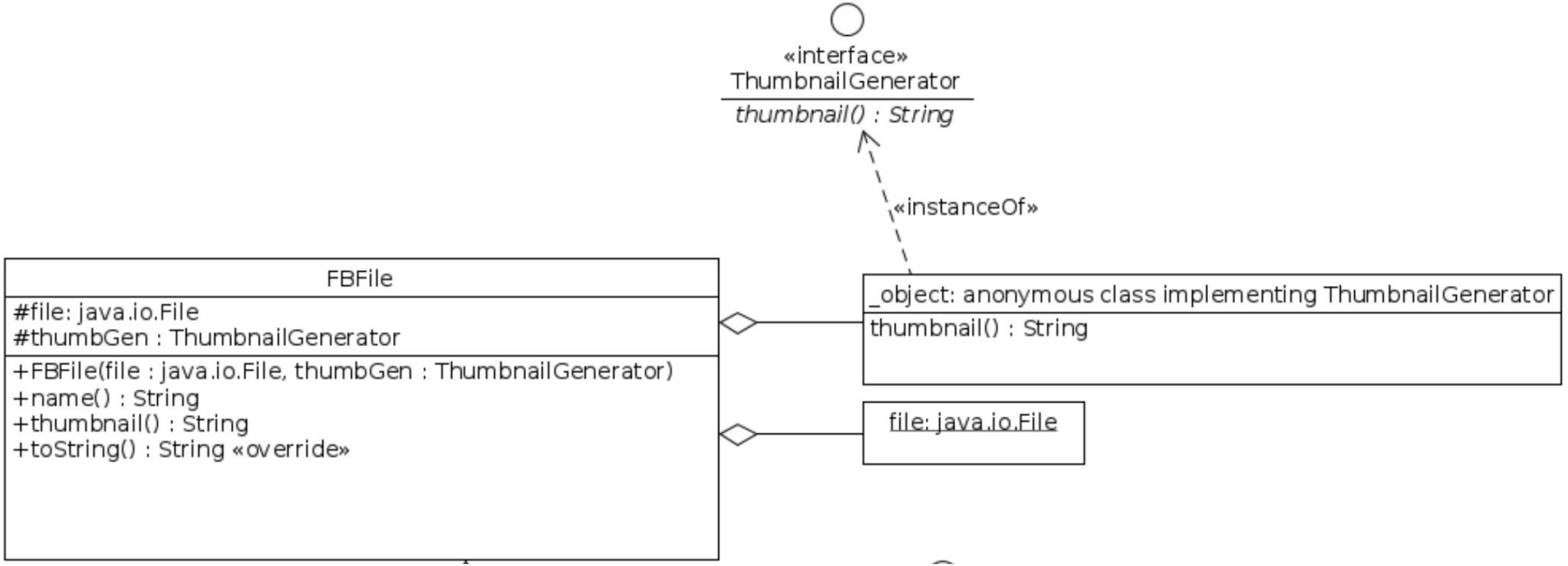
Let's encapsulate the thumbnail from the FBFile class, since it's the only thing which differs from different file types.

- Every `FBFile` object should have its own `ThumbnailGenerator`
  - Provided to the constructor and saved in an instance variable
- `ThumbnailGenerator` is of interface type
- `ThumbnailGenerator` declares one method, `String thumbnail()`
- `FBFile`'s `thumbnail()` method forwards the call to the `ThumbnailGenerator`
- Every FBFile object should also have a java.io.File instance variable
  - Also stored in an instance variable, provided to the constructor
  - Used by the name() method of FBFile

`FBFile` is a composition

- `FBFile` is now composed of `java.io.File` and our own interface `ThumbnailGenerator`
- Composition is nothing new, we've used it many times
  - for instance a Person class might "**have a**" name instance variable (which is a `String`)
- This is very common, encapsulate what varies, and compose your classes with specialized objects
- ThumbnailGenerator knows how to generate the thumbnail
- java.io.File knows about file names (and more)

UML for FBFile



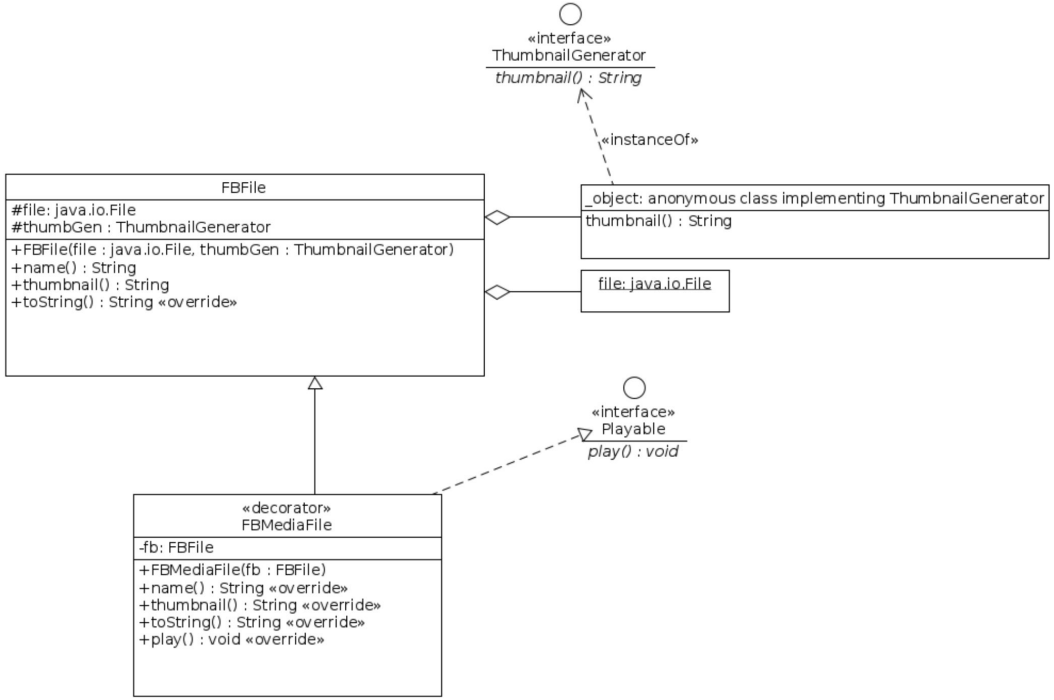
What about a media player?

Now we can use `FBFile` in a file browser application. But what about a media player?

- It would be nice to re-use `FBFile` for some parts of a media player app
- Media players also handle files (and could have use for the thumbnail)
- Media players, however, are only interested in Playable files
- So let's create a `Playable` interface, declaring a `play()` method

```
public interface Playable {
  public void play();
}
```

Let’s decorate FBFile to become FBMediaFile



Using inheritance in a new way

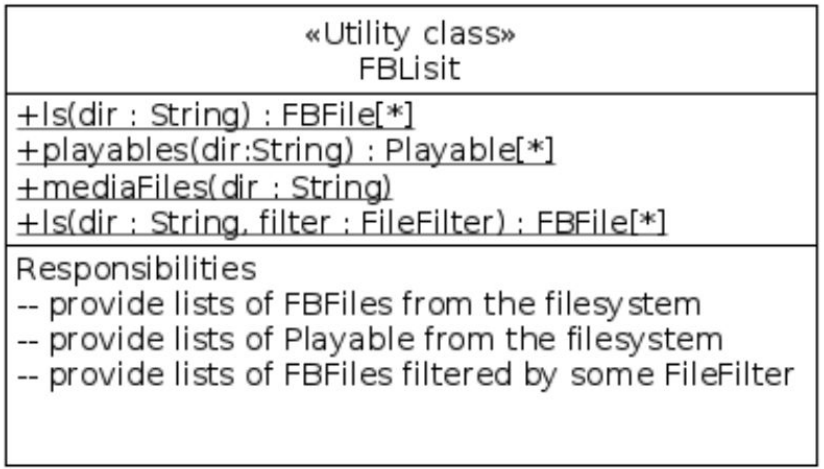
- FBMediaFile extends FBFile and implements Playable
- It extends FBFile but is also composed with an FBFile
  - stored in an instance variable and provided via the constructor
- Overrides name() and thumbnail()
  - Using the internal FBFile object
- Implements play() from the Playable interface
- This is sometimes called *decoration*
- Decoration like this can be done in several layers:

```
FBStreamableMediaFile streamableMedia = new FBStreamableMediaFile(
  new FBMediaFile(someFBFile));
// Now has both play() and stream() and still is an FBFile
```

Last piece of the puzzle - a utility class

Using a utility class, we can

- List a directory and create a list of files with correct thumbnail generators
- Dynamically create thumbnail generators on the fly



Utility methods

- public static List<FBFile> ls(String dir)
- public static List<FBFile> ls(String dir, FileFilter filter)
  - java.io.FileFilter has been around since Java 1.2
- public static List<Playable> playables(String dir)
  - If you want a list of Playable objects
- and you could add more useful methods...

Building a list of FBFiles w. thumbnail generators

The `ls` method could create a list of `FBFile` objects from a directory, and create a `ThumbnailGenerator` on the fly:

```
public static List<FBFile> ls(String dir, FileFilter filter) {
    List<FBFile> files = new ArrayList<>();

    for (File f : new File(dir).listFiles(filter)) {
      if (f.isDirectory()) {
        files.add(new FBFile(f, () -> "[dir]"));
      } else {
        if (f.getName().contains(".")) {
          files.add(new FBFile(f, () -> "[" + suffix(f) + "]"));
        } else {
          files.add(new FBFile(f, () -> "[file]"));
        }
      }
    }
    return files;
  }
}
```

Using a lambda as the ThumbnailGenerator

```
new FBFile(f, () -> "[dir]")
```

The second argument, `() -> "[dir]"` is a “lambda expression”.

Since ThumbnailGenerator only declares one method, you can create an object implementing that interface using a so called lambda.

- `()` means “no arguments method” (the only method being thumbnail())
- `->` means “the return value follows this arrow”
- `"[dir]"` is the value the method should return

Ersätter exempelvis:

```
new ThumbnailGenerator() {
    public String thumbnail() { return "[dir]"; }
}
```

Client code examples

```
// Anonymous inner class
FBFile javaFile =
      new FBFile(new File("FileBrowser.java"),
                 new ThumbnailGenerator() {
                   public String thumbnail() {
                     return "[java]";
                   }
        });
System.out.println(javaFile);
// Lambda expression
FBFile classFile = new FBFile(new File("FileBrowser.class"),
                              () -> "[class]");
System.out.println(classFile);
```

Client code examples

```
System.out
  .println("Pretending to be a mediaplayer and playing" +
           " all playables obtained from FBList.playables():");

for (Playable playable : FBList.playables(dir)) {
  playable.play();
}
```

Client code examples

```
System.out.println("Listing all Text files (.txt | .java) in " +
                   dir + " using a custom file filter:");

for (FBFile fb :
     FBList.ls(dir, f -> f.getName().endsWith(".java") ||
                   f.getName().endsWith(".txt"))) {
  System.out.println(fb);
}
```

Summary

- We didn’t want a class explosion (FBMp3, FBTxt, FBAvi, FBOgg,……)
- We ended up with two classes
  - FBFile
  - FBMediaFile implements Playable
- We used composition; an FBFile
  - *has a* java.io.File
  - *has a* ThumbnailGenerator (all that differed was the thumbnail!)
  - additionally, an FBMediaFile *has an* FBFile (decorator pattern)
- Encapsulating the generation of the thumbnail allowed us to use one single class (FBFile)
- A ThumbnailGenerator could be constructed using a lambda

Links

https://docs.oracle.com/javase/8/docs/api/java/io/FileFilter.html

https://github.com/progund/interfaces/tree/master/interfaces-instead-of-class-explosion