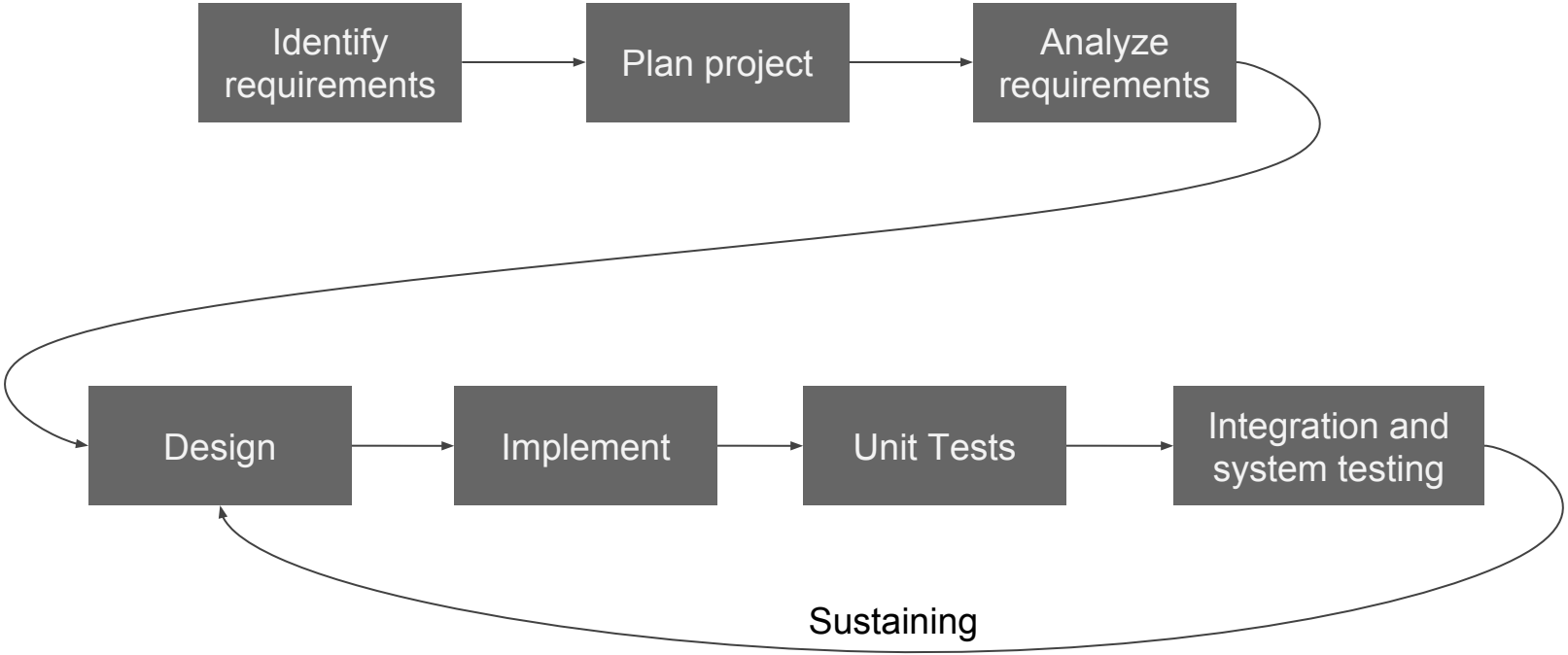
Unit and Integration testing

U need testing :)

Kirill Volkov
Senior SW Developer @ MERA

Software engeneering cycle



Meaning of unit testing

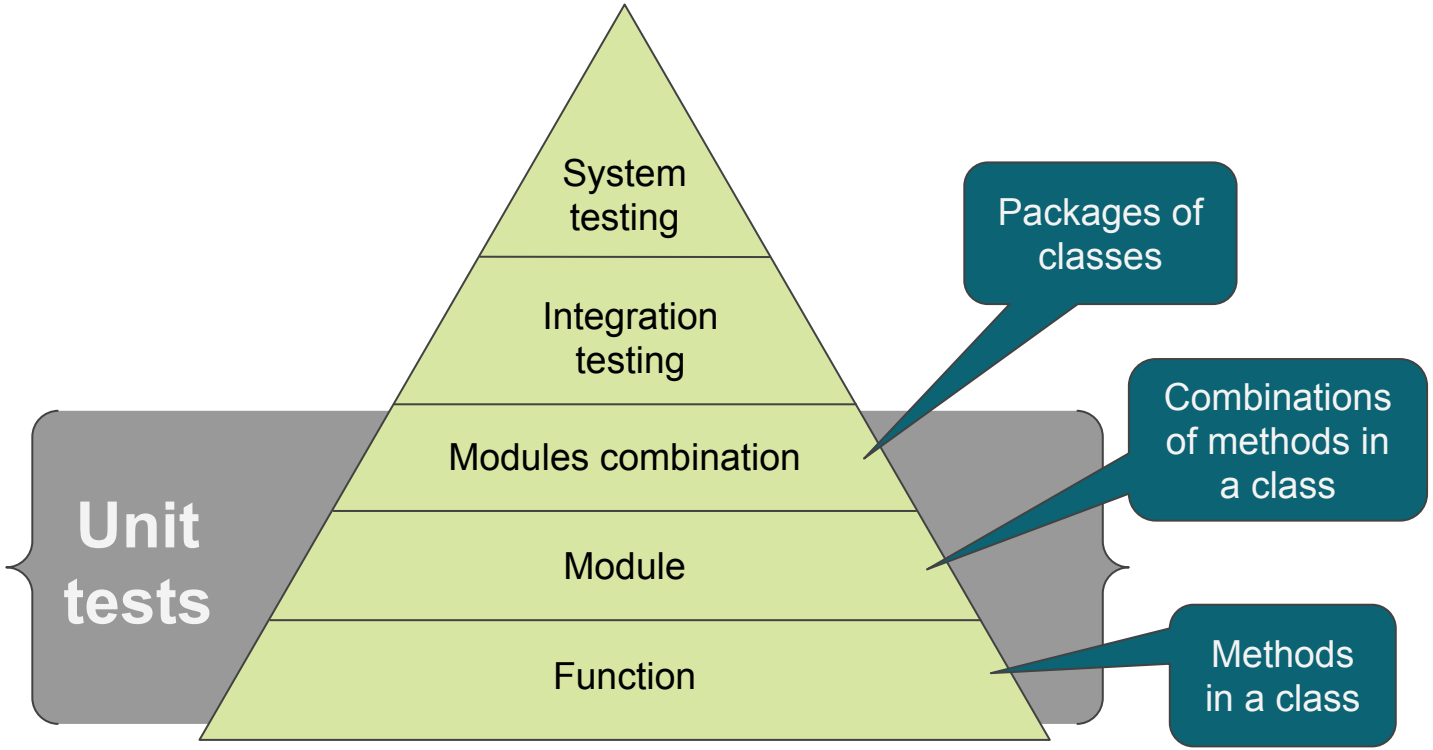
Goal:

Maximize the number and severity of defect found per dollar spent
-> TEST EARLY

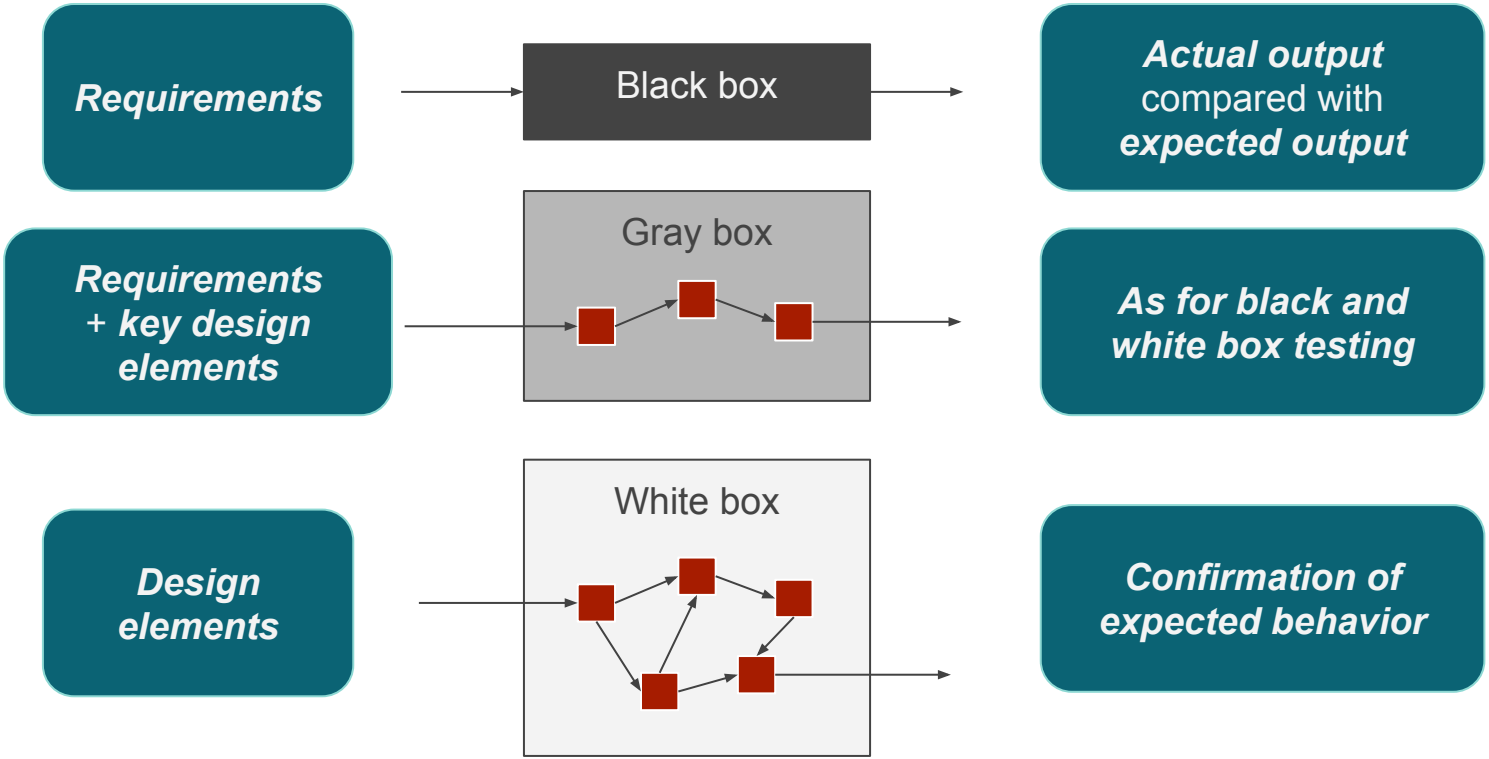
Limits:

Testing can only determine the presence of defects, not their absence
-> Use proofs of correctness to establish “absence”

Big picture

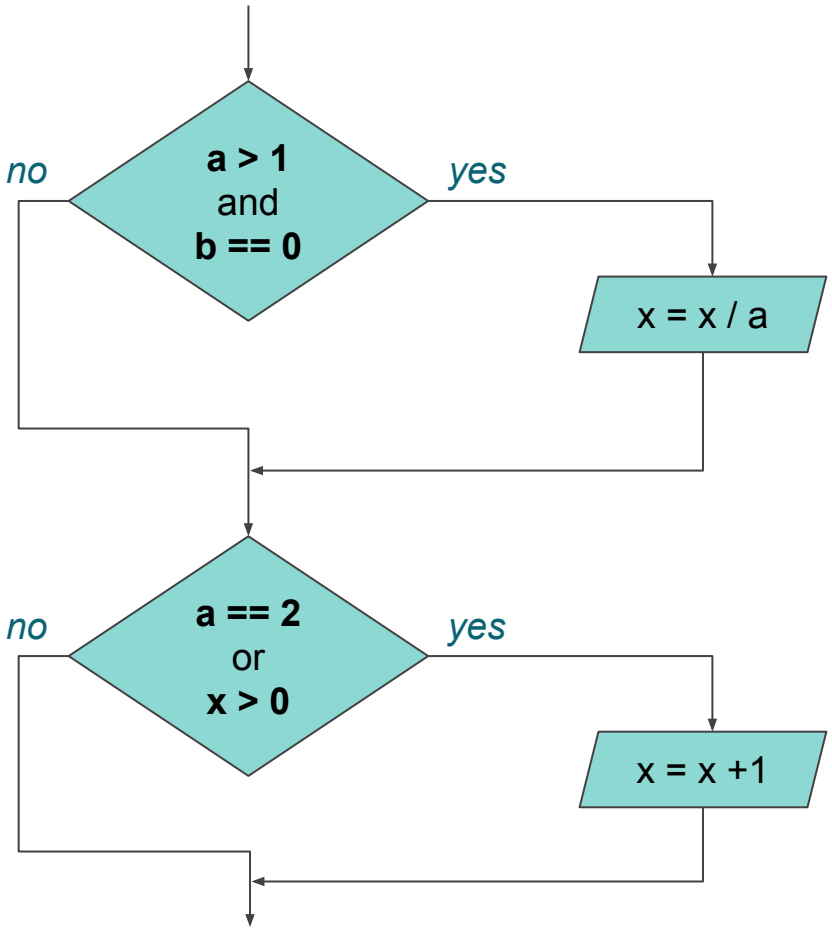

System testing
Integration testing
Modules combination
Module
Function
Unit tests
Packages of classes
Combinations of methods in a class
Methods in a class

Types of testing



Blackbox testing

```
float calc(float x, float a, float b)
{
    float retVal = x;
    if (a > 1 && b == 0) {
        retVal /= a;
    }

    if (a == 2 || x > 0) {
        ++retVal;
    }

    return retVal;
}
```



Blackbox testing

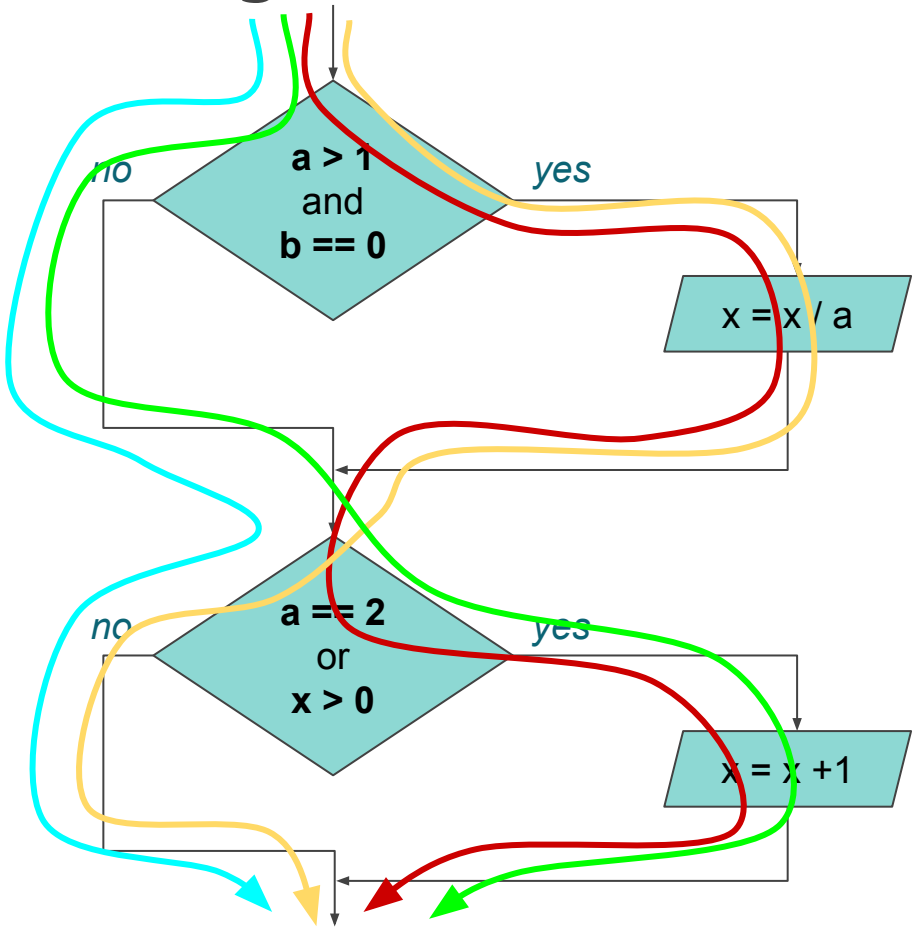
```
float calc(float x,
           float a,
           float b) {
    float retVal = x;
    if (a > 1 && b == 0) {
        retVal /= a;
    }

    if (a == 2 || x > 0) {
        ++retVal;
    }

    return retVal;
}
```

```
x = 3; a = 2; b = 0;
float value = calcX(x, a, b);

verify(value == 2.5); // -> TEST PASSED
```

Now change this to **x > 3**

```
x = 3; a = 2; b = 0;
float value = calcX(x, a, b);

verify(value == 2.5); // -> TEST STILL
                      //         PASSED! :(
```

Test coverage



Blackbox testing Java Example

```
public class StringUtils {
    public static String concatenate(String one, String two) {
        return one + two;
    }
}
```

```
import org.junit.Test;
import static org.junit.Assert.*;

public class StringUtilsTest {

    @Test
    public void testConcatenate() {
        assertEquals("onetwo", StringUtils.concatenate("one", "two"));
        assertEquals("twoone", StringUtils.concatenate("two", "one"));
        assertEquals("justone", StringUtils.concatenate("justone", ""));
        assertEquals("justtwo", StringUtils.concatenate("", "justtwo"));
        assertEquals("", StringUtils.concatenate("", ""));
    }
}
```

Testing Java Code: Assertions

```
assertArrayEquals()
assertEquals()
assertTrue() + assertFalse()
assertNull() + assertNotNull()
assertSame() + assertNotSame()
assertThat()
```

```
assertArrayEquals(expectedArray, resultArray);
assertEquals("onetwo", result);
assertTrue(myUnit.getTheBoolean());
assertFalse(myUnit.getTheBoolean());
assertNull(myUnit.getTheObject());
assertNotNull(myUnit.getTheObject());
assertSame(myUnit.getTheSameObject(), myUnit.getTheSameObject());
assertNotSame(myUnit.getTheSameObject(), myUnit.getTheSameObject());

assertThat();   // REQUIRES A MATCHER...
```

Testing Java Code: Matchers

```
    public void assertThat(Object o, Matcher matcher){
        ...
    }
...
        assertThat("this string", is("this string"));
        assertThat(123, is(123));
```

```
Core

    any()           // Matches anything

    is()            // A matcher that checks if the given objects are equal.

    describedAs()   // Adds a descrption to a Matcher
```

Testing Java Code: Matchers

```
    public void assertThat(Object o, Matcher matcher){
        ...
    }
...
        assertThat("this string", is("this string"));
        assertThat(123, is(123));
```

```
Logical

    allOf()       // Takes an array of matchers, and all matchers must
                  // match the target object.

    anyOf()       // Takes an array of matchers, and at least one of the
                  // matchers must report that it matches the target object.

    not()         // Negates the output of the previous matcher.
```

Testing Java Code: Matchers

```
    public void assertThat(Object o, Matcher matcher){
        ...
    }
...
        assertThat("this string", is("this string"));
        assertThat(123, is(123));
```

```
Object

    equalTo()        // A matcher that checks if the given objects are equal.

    instanceOf()     // Checks if the given object is of type X or
                     // is compatible with type X

    notNullValue() + nullValue()
                     // Tests whether the given object is null or not null.

    sameInstance()   // Tests if the given object is the exact same
                     // instance as another.
```

Testing Java Code: Matchers

```
...
    assertThat(testedObject, matches("constant string"));
...
```

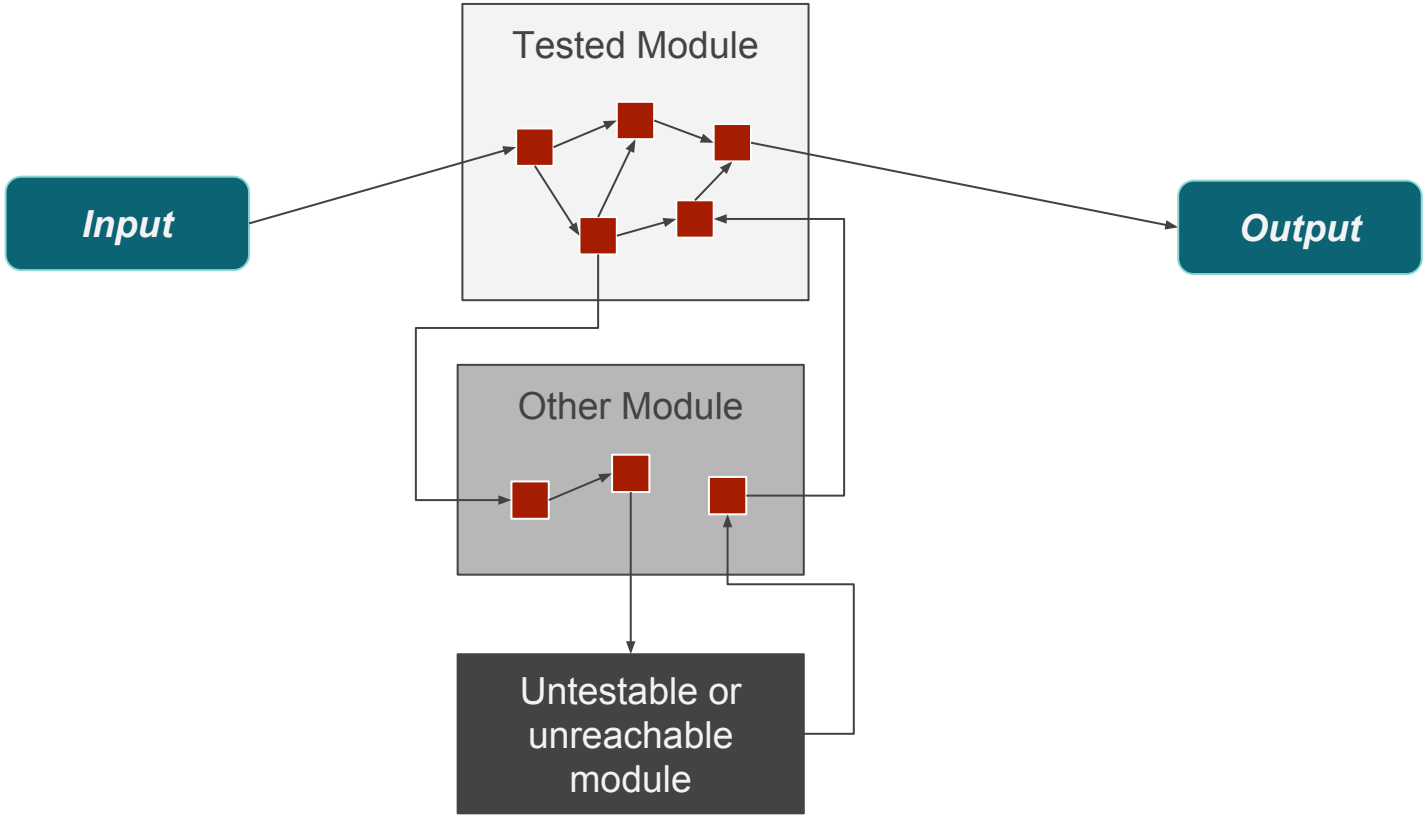
```
public static Matcher matches(final Object expected){

    return new BaseMatcher() {

        protected Object mExpected = expected;

        public boolean matches(Object o) {
            return mExpected.equals(o);
        }

        public void describeTo(Description description) {
            description.appendText(mExpected.toString());
        }
    };

}
```

Testing Java Code: Exceptions

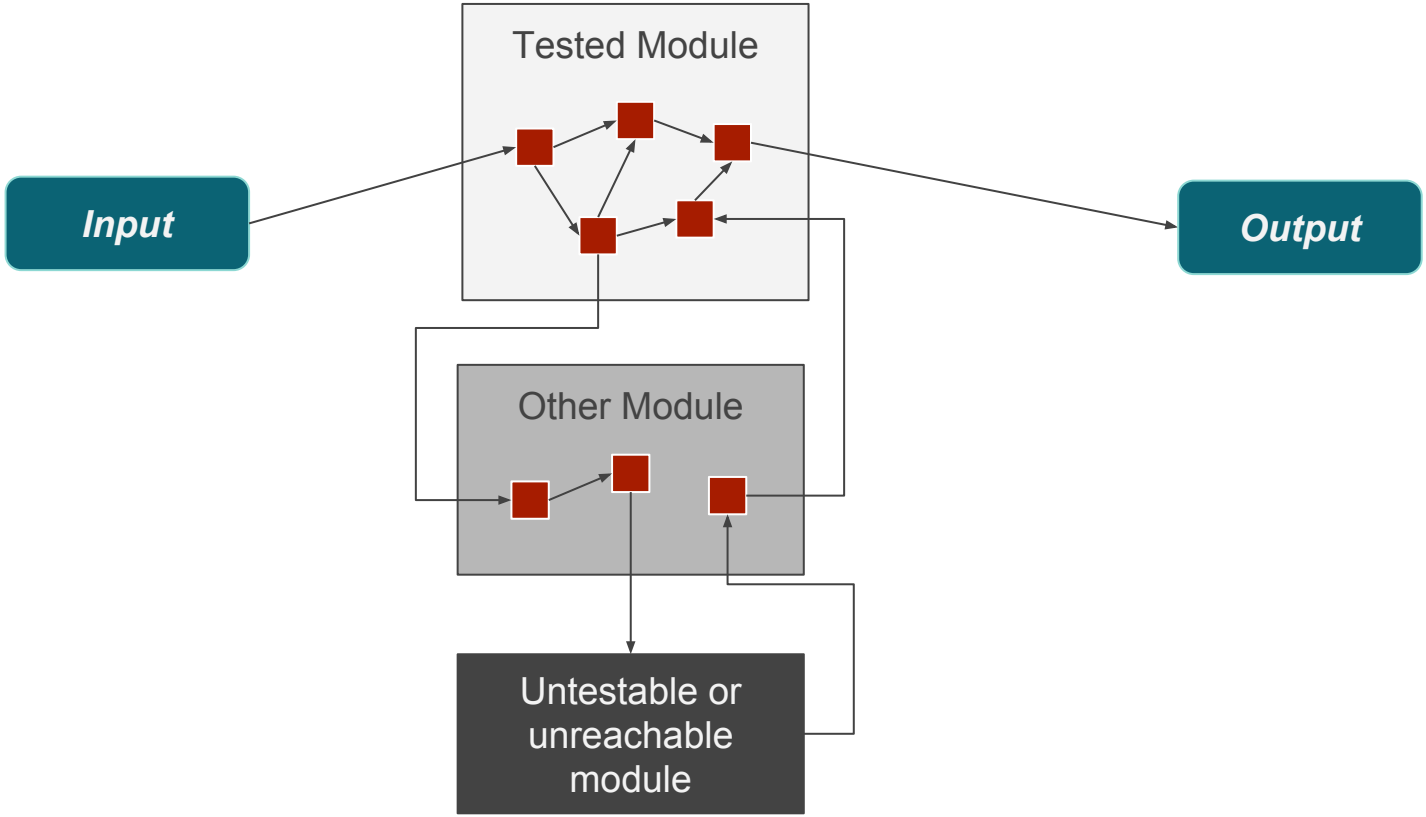
```
@Test(expected = IllegalArgumentException.class)
public void testForException1() {
    TestedObject testedObject = new MyTestedObjectUnit();
    testedObject.throwIllegalArgumentException();
}
```

```
@Test
public void testForException2() {
    TestedObject testedObject = new TestedObject();

    try {
        testedObject.throwIllegalArgumentException();

        fail("expected IllegalArgumentException");

    } catch(IllegalArgumentException e){
        // ignore, this exception is expected.
    }
}
```

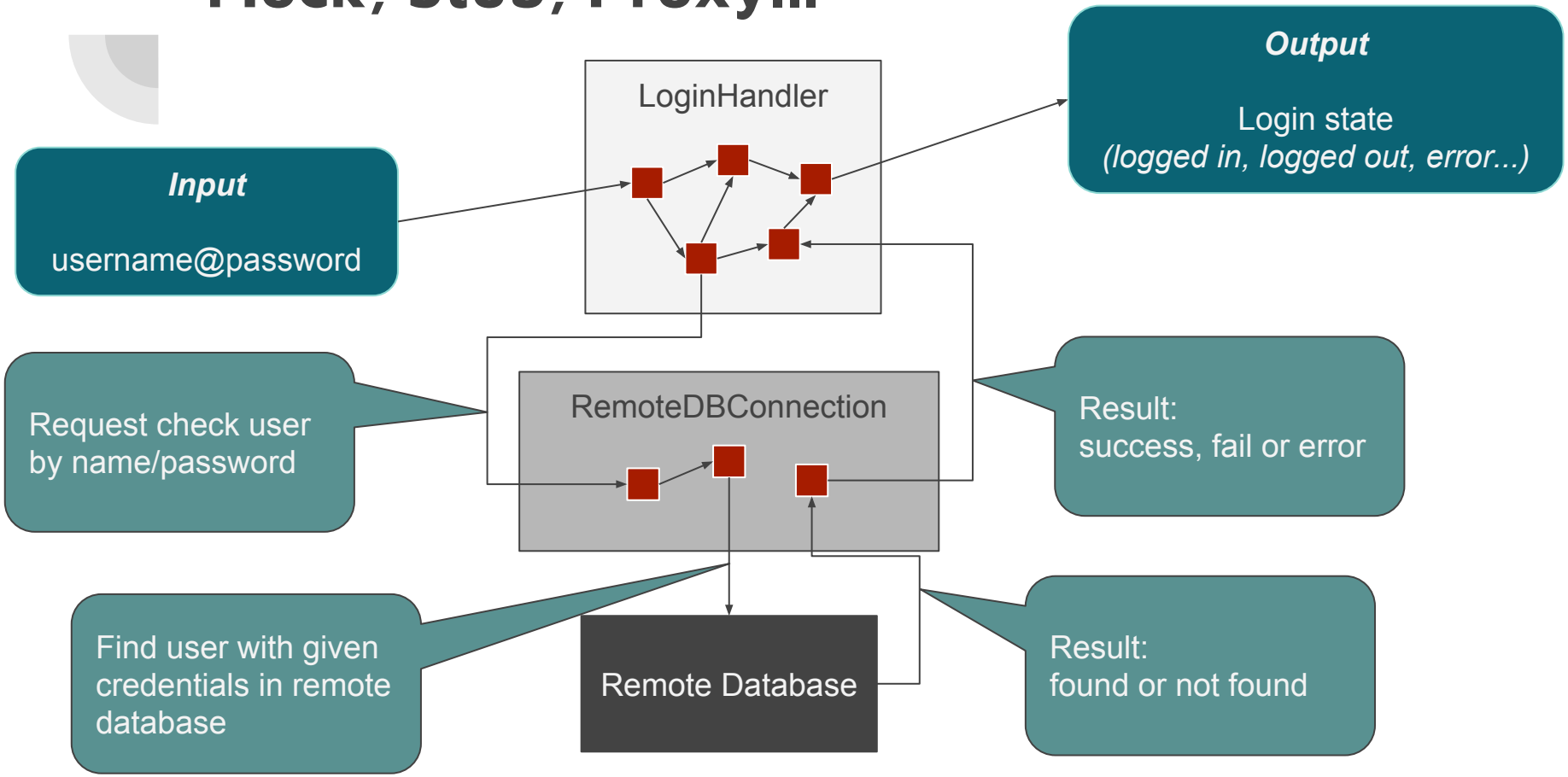
Mock, Stub, Proxy...



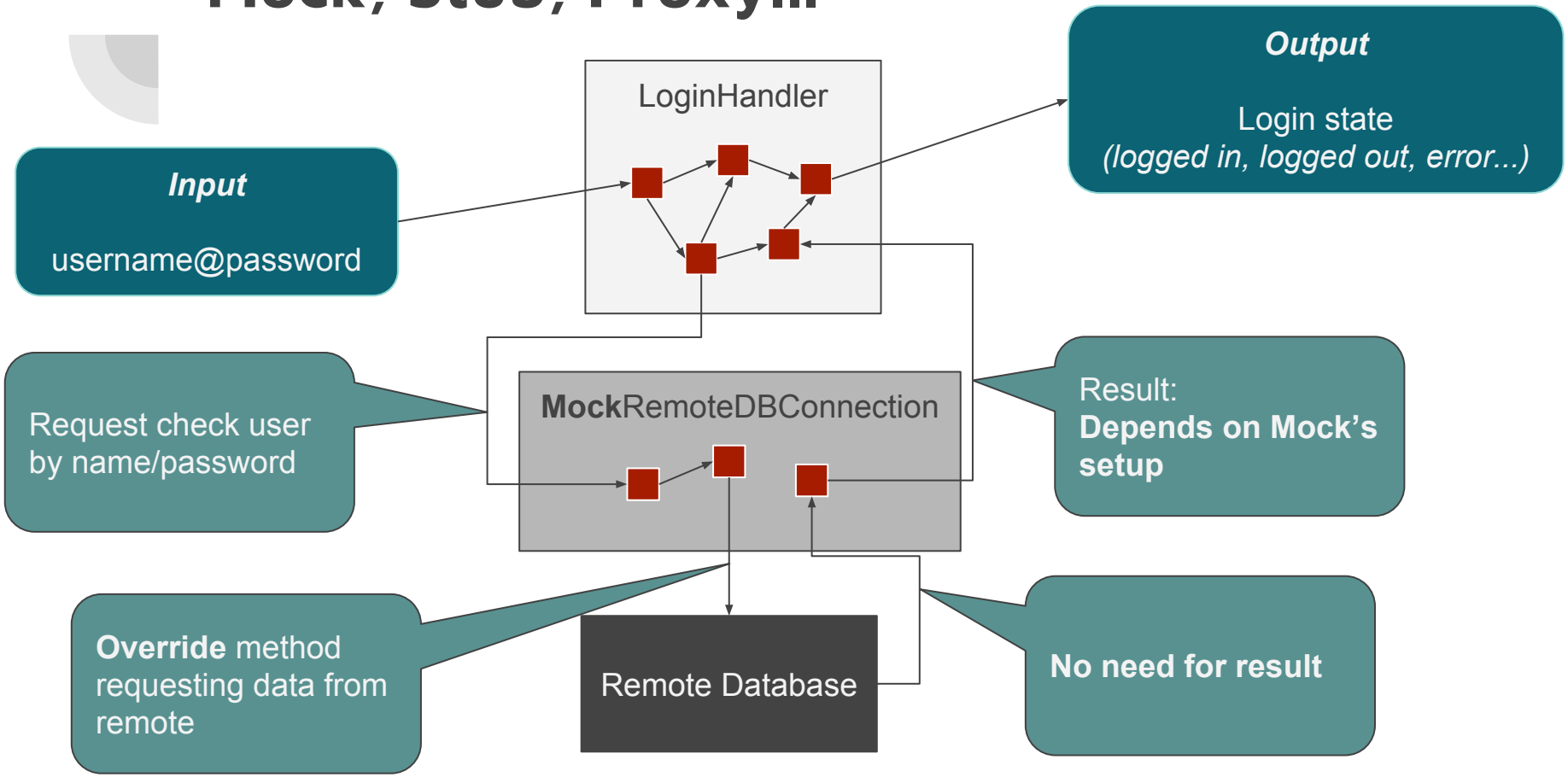
Mock, Stub, Proxy...



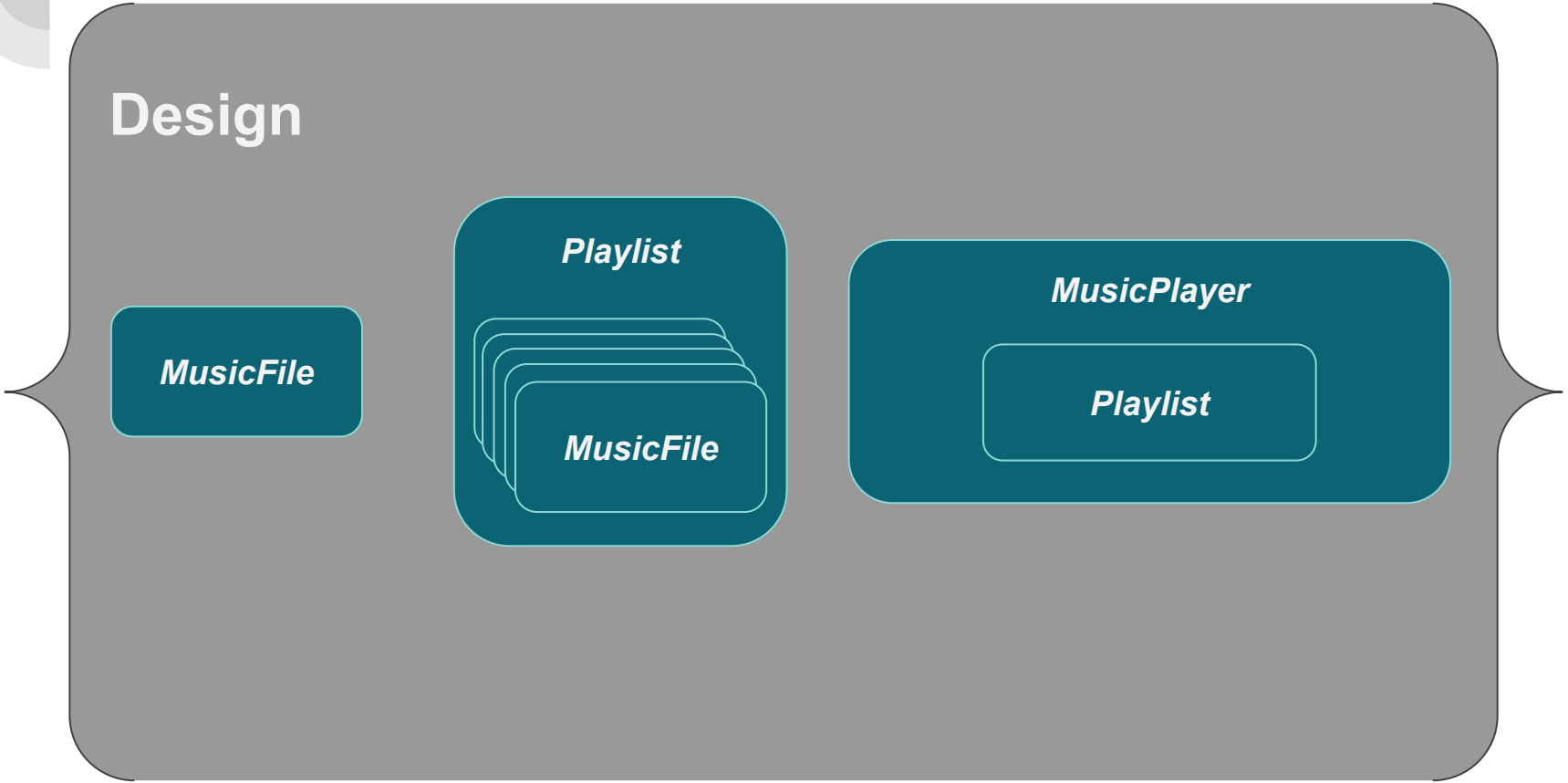
Mock, Stub, Proxy...



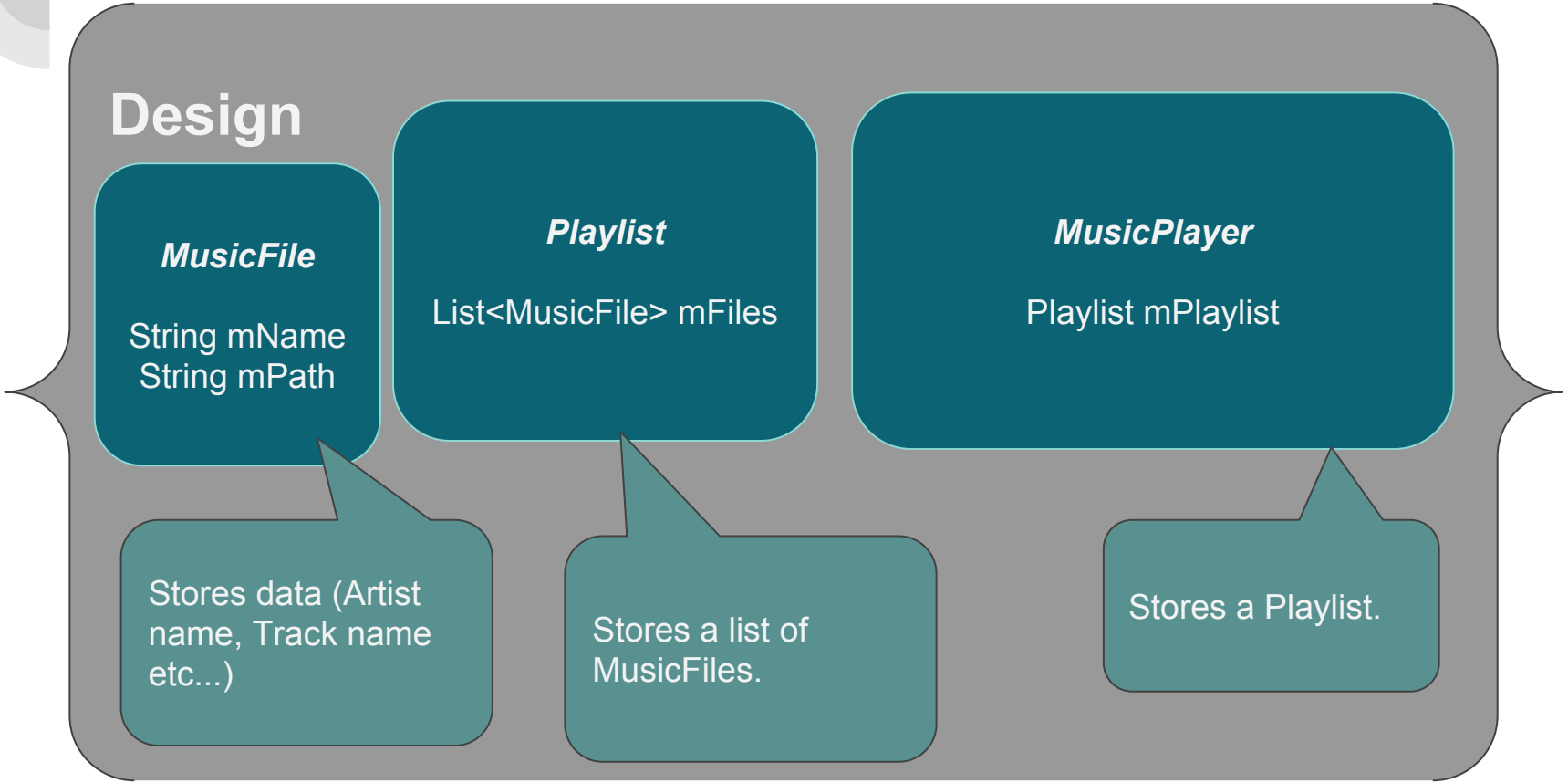
Mock, Stub, Proxy...



Designing a Music Player application



Designing a Music Player application



Designing a Music Player application

Design

MusicFile

load()
verify()
cleanup()

Load data from disk that can be played. Stores data (Artist name, Track name etc...)

Playlist

addFiles()
getCurrentTrack()
nextTrack()
previousTrack()
cleanup()

Stores a list of MusicFiles.
Stores current track.
Switch to next, previous track.

MusicPlayer

createPlaylistFromDirectory()
play()
pause()
stop()
next()
previous()

Stores a Playlist.
Controls playback.

MusicFile

```
public class MusicFile {

    enum FileType {
        UNSUPPORTED,
        MP3,
        WAV,
        AAC
    };

    private String mName;
    private FileType mFileType;
    private File mFile;

}
```

MusicFile

```
public void load(String path) throws Exception {
    mFile = new File(path);
    if (!mFile.isFile()) {
        throw new Exception("No file exists with a given path: "
                                + path);
    } else {
        mName = mFile.getName();
        verify();
    }
}

public void cleanup() {
    // TODO: cleanup
}

@Override
public String toString() {
    return mName + " of type: " + mFileType.toString();
}
```

MusicFile

```
private void verify() {
    // check extension
    String extension = "";
    int pos = mName.lastIndexOf('.');
    if (pos > 0) {
        extension = mName.substring(pos + 1);
    }

    if (extension.toLowerCase().equals("mp3")) {
        mFileType = FileType.MP3;
    } else if (extension.toLowerCase().equals("wav")) {
        mFileType = FileType.WAV;
    } else if (extension.toLowerCase().equals("aac")) {
        mFileType = FileType.AAC;
    } else {
        mFileType = FileType.UNSUPPORTED;
    }
}
```

Playlist

```
public class Playlist {

    private List<MusicFile> mFilesList = new LinkedList();
    private ListIterator<MusicFile> mCurrentTrackIterator = null;
    private MusicFile mCurrentTrack = null;

    public void cleanup() {
        for (MusicFile musicFile : mFilesList) {
            musicFile.cleanup();
        }
        mFilesList.clear();
    }

}
```

Playlist

```
public void addFiles(List<MusicFile> files) throws Exception {
    if (files == null || files.isEmpty()) {
        throw new Exception("Playlist is invalid or empty!");
    }

    mFilesList.addAll(files);
    mCurrentTrackIterator = mFilesList.listIterator(0);
    mCurrentTrack = mCurrentTrackIterator.next();
}

public MusicFile getCurrentTrack() {
    return mCurrentTrack;
}
```

Playlist

```
public MusicFile nextTrack() {
    mCurrentTrack = mCurrentTrackIterator.next();

    return mCurrentTrack;
}

public MusicFile previousTrack() {
    mCurrentTrack = mCurrentTrackIterator.previous();

    return mCurrentTrack;
}
```

MusicPlayer

```
public class MusicPlayer {

    enum PlayerState {
        UNKNOWN, STOPPED, PLAYING, PAUSED
    }

    private Playlist mPlaylist = null;
    private PlayerState mState = PlayerState.UNKNOWN;

}
```

MusicPlayer

```
public void openDirectory() {
    if (mState != PlayerState.UNKNOWN) {
        stopIfWasPlaying();
        if (mPlaylist != null) {
            mPlaylist.cleanup();
        }
    }

    mPlaylist = new Playlist();
    try {
        mPlaylist.addFiles(null);
    } catch (Exception e) {
        // TODO: handle
    }
}
```

MusicPlayer

```
public void play() {
    if (mState == PlayerState.PLAYING) {
        return;
    }

    startPlayback();
    mState = PlayerState.PLAYING;
}

public void pause() {
    if (mState == PlayerState.PAUSED) {
        return;
    }

    pausePlayback();
    mState = PlayerState.PAUSED;
}
```

MusicPlayer

```
public void stop() {
    if (mState == PlayerState.STOPPED) {
        return;
    }

    stopPlayback();
    mState = PlayerState.STOPPED;
}

private boolean stopIfWasPlaying() {
    if (mState == PlayerState.PLAYING) {
        stop();

        return true;
    }

    return false;
}
```

MusicPlayer

```
public void next() {
    boolean continuePlayback = stopIfWasPlaying();

    mPlaylist.nextTrack();

    if (continuePlayback) {
        play();
    }
}

public void previous() {
    boolean continuePlayback = stopIfWasPlaying();

    mPlaylist.previousTrack();

    if (continuePlayback) {
        play();
    }
}
```

MusicPlayer

```
private void startPlayback() {
    System.out.println("Play file "
                        + mPlaylist.getCurrentTrack().toString());
}

private void pausePlayback() {
    System.out.println("Pause file "
                        + mPlaylist.getCurrentTrack().toString());
}

private void stopPlayback() {
    System.out.println("Stop file "
                        + mPlaylist.getCurrentTrack().toString());
}
```