Exception handling refactorings

OOA/OOD

xuyingxiao@126.com

Fudan University

- 1) Checked exception:
  - IOException
  - Handle or declare
  - 这类异常都是Exception的子类。
- 2) Unchecked exception:
  - ArithmeticException
  - 这类异常都是RuntimeException的子类，虽然RuntimeException同样也是Exception的子类，但是它们是特殊的，它们不能通过client code来试图解决，所以称为Unchecked exception

Problem

```
public void writeFile(String fileName, String data){
        Writer writer = null;
        /* may throw an IOException */
        writer = new FileWriter(fileName);
        /* may throw an IOException */
        writer.write(data);
}
```

- Declare

```
public void writeFile(String fileName, String data) throws IOException{
        Writer writer = null;
        /* may throw an IOException */
        writer = new FileWriter(fileName);
        /* may throw an IOException */
        writer.write(data);
}
```

Handle

```
public void writeFile(String fileName, String data){
        Writer writer = null;
        try {
                /* may throw an IOException */
                writer = new FileWriter(fileName);
                /* may throw an IOException */
                writer.write(data);
        }
        catch (IOException e)  {
                /* a lot of code*/
                。 。 。 。 。 。
        }
        finally {/* code for cleanup */}
}
```

Bad Smell

- Handle——Bad Smell:
  - Ignored checked exception

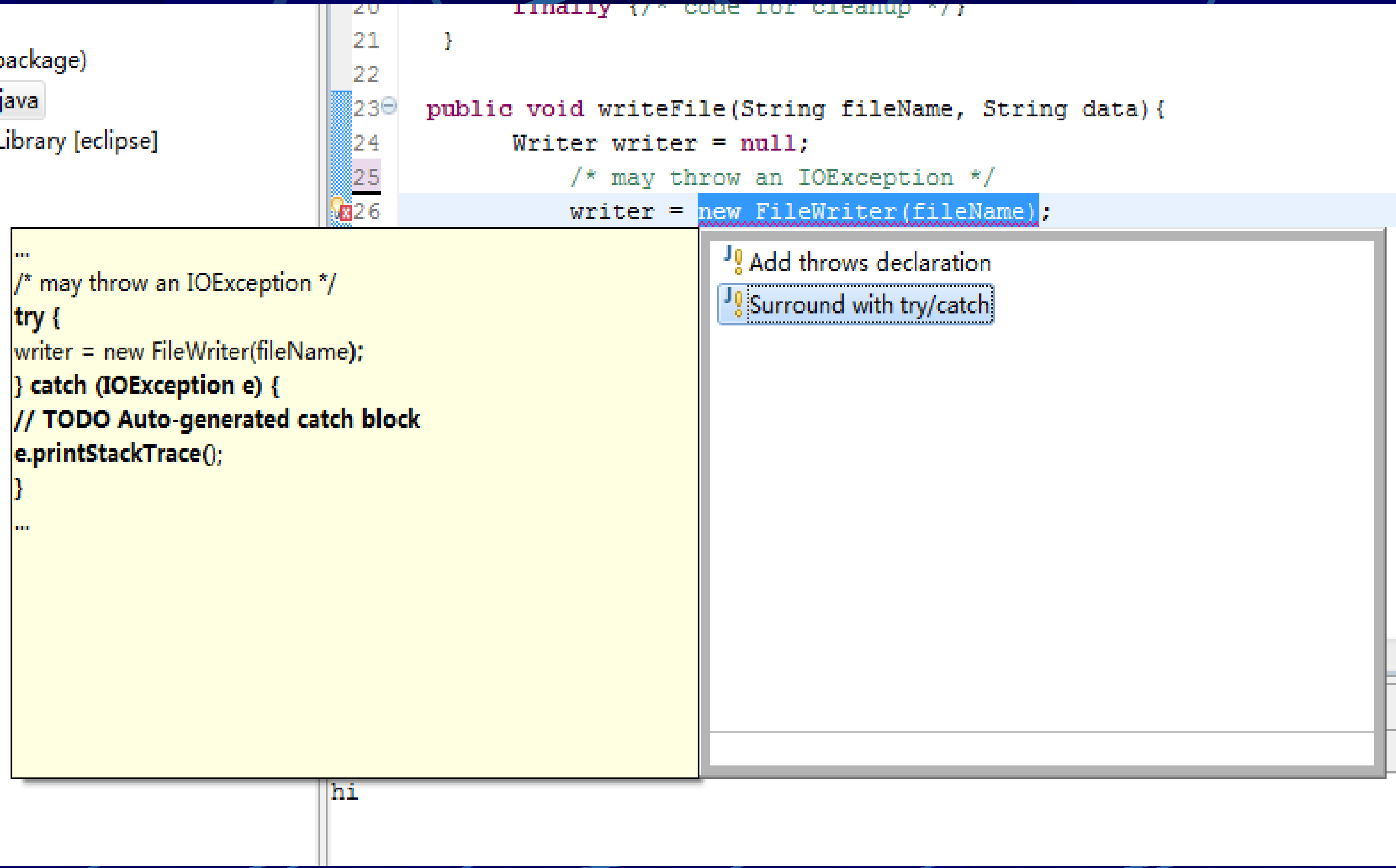
```
public void writeFile(String fileName, String data){
        Writer writer = null;
        try {
                /* may throw an IOException */
                writer = new FileWriter(fileName);
                /* may throw an IOException */
                writer.write(data);
        }
        catch (IOException e) ){ /* TO DO */  }
        finally {/* code for cleanup */}
}
```

- BAD SMELL
  - a checked exception is caught but nothing is done to deal with it,
  - the program is pretending that all is fine when in fact something is wrong.

Bad Smell

- Handle——Bad Smell:
  - Dummy handler

```
public void writeFile(String fileName, String data){
        Writer writer = null;
        try {
                /* may throw an IOException */
                writer = new FileWriter(fileName);
                /* may throw an IOException */
                writer.write(data);
        }
        catch (IOException e){
              e.printStackTrace();
             // or System.out.println(e.toString());
         }
        finally {/* code for cleanup */}
}
```



package)
java
Library [eclipse]

```
        finally {/* code for cleanup */}
    }

    public void writeFile(String fileName, String data){
        Writer writer = null;
            /* may throw an IOException */
            writer = new FileWriter(fileName);
```

```
...
/* may throw an IOException */
try {
writer = new FileWriter(fileName);
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
...
```

```
hi
```

- Solution
  - catch (IOException e)){ throw e; } ?

Refactoring

- Refactoring:
  - Replace ignored checked exception with unchecked exception
  - Replace dummy handler with rethrow
    - wrap the checked exception into an unchecked UnhandledException
    - Throw UnhandledException

```
class UnhandledException extends RuntimeException{
        …
        public UnhandledException(Exception e, String msg){
                …
        }
}
```

```
public void writeFile(String fileName, String data){
        Writer writer = null;
        try {
                /* may throw an IOException */
                writer = new FileWriter(fileName);
                /* may throw an IOException */
                writer.write(data);
        }
        catch (IOException e) {
                throw new UnhandledException(e,"message");
    }
        finally {/* code for cleanup */}
 }
```

Furthor Refactoring

Bad Smell

- Bad Smell
  - Unprotected main program

- public static void main (String[] args){
- 		s.writeFile(“xxx”,”yyy”);
- }

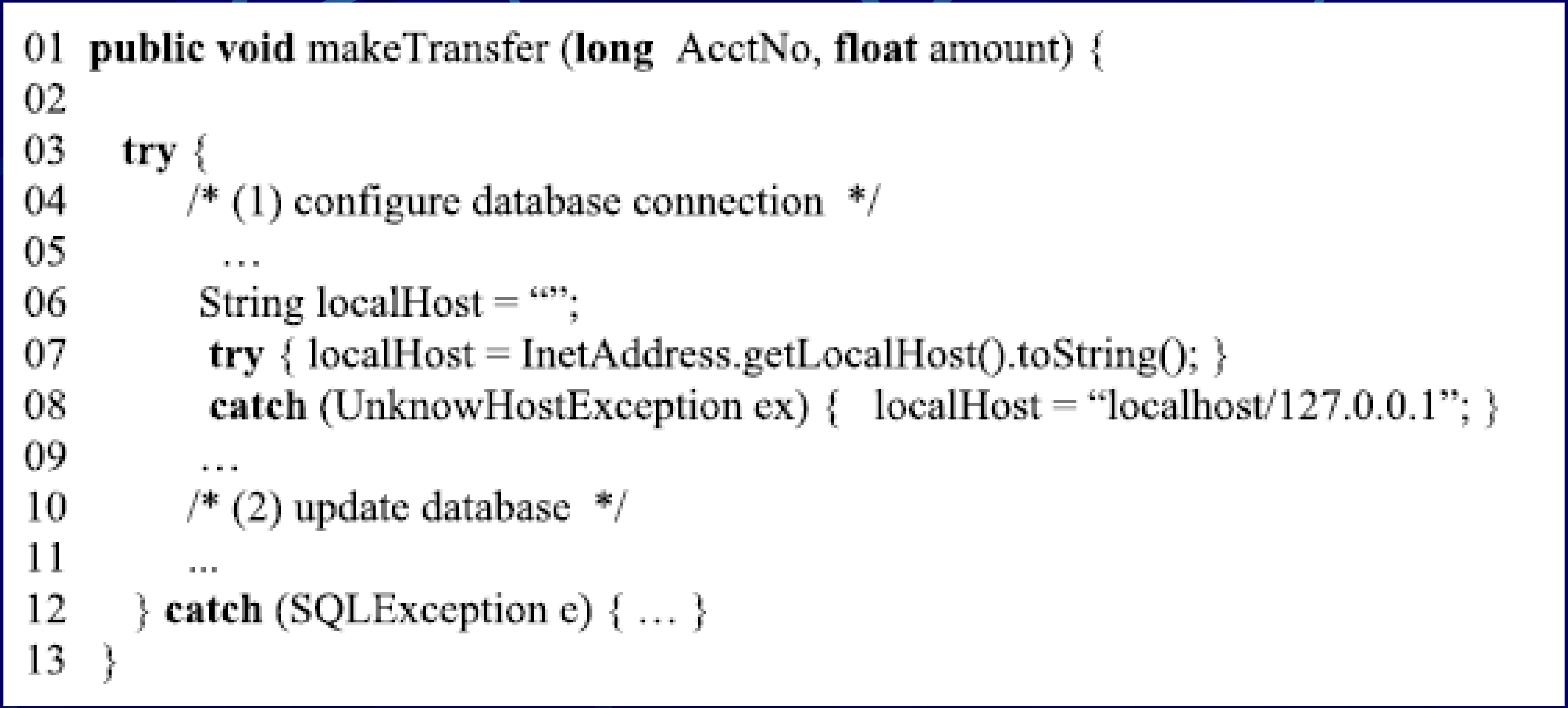
Refactoring

- Avoid unexpected termination with big outer try block

```
public static void main(String[] args){
    try{
            /*some code */
     }
     catch (Throwable e) {/*display e */}
}
```

Bad Smell: Nested try block

```
try {
        in = new FileInputStream(  );
}
finally{
    try {
            if (in != null) in.close ();
    }
    catch (IOException e){
            /* log the exception */
    }
}
```



```
public void makeTransfer (long  AcctNo, float amount) {

   try {
         /* (1) configure database connection  */
            …
          String localHost = “”;
           try { localHost = InetAddress.getLocalHost().toString(); }
           catch (UnknowHostException ex) {   localHost = “localhost/127.0.0.1”; }
          …
         /* (2) update database  */
         ...
    } catch (SQLException e) { … }
 }
```

- Problem
  - yields complicated program
  - structures and easily results in a Long Method

```
public void update() {
        Connection conn = null;
        PreparedStatement ps = null;
        try {
                conn = getConnection();
                ps = conn.prepareStatement(/* update user's data */);
        } catch (SQLException e) {
                /* rollback */
        } finally {
                try {
                        if (ps != null)
                                ps.close();
                        if (conn != null)
                                conn.close();
                } catch (Exception e) {/* log exception */
                }
        }
```

Refactoring

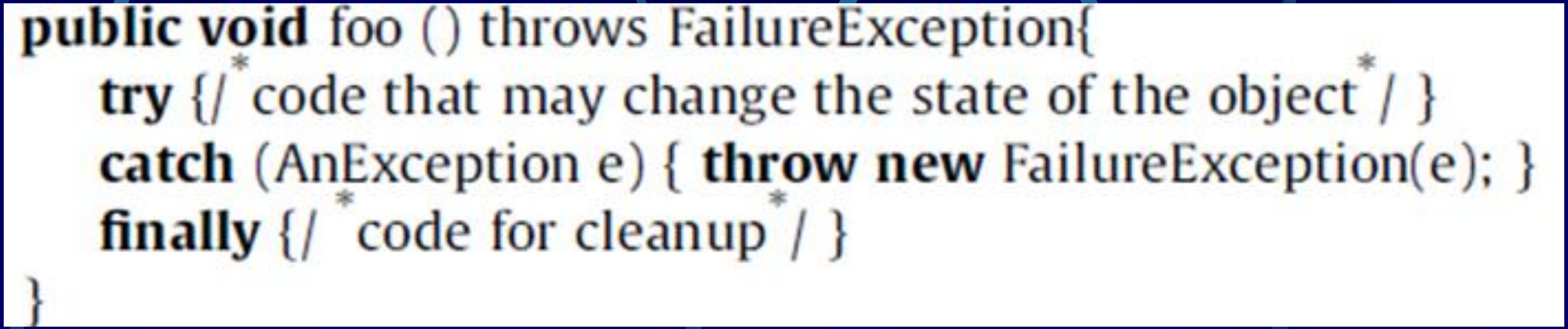
- Refactoring
  - Replace nested try block with method

```
public void update() {
        Connection conn = null;
        PreparedStatement ps = null;
        try {
                conn = getConnection();
                ps = conn.prepareStatement(/* update user's data */);
        } catch (SQLException e) {
                /* rollback */
        } finally {

                     close (ps);
                     close(conn);

                      /* log exception */
                }
        }
}
```

```
public static void close(PreparedStatement obj) {
        try {
                if (obj != null)
                        obj.close();
        } catch (Exception e) {
                /* log exception */
        }
}

public static void close(Connection obj) {
        try {
                if (obj != null)
                        obj.close();
        } catch (Exception e) {
                /* log exception */
        }
}
```

Introduce checkpoint class

- Make sure that the program remains in a correct state.



```
public void foo () throws FailureException{
    try {/* code that may change the state of the object */ }
    catch (AnException e) { throw new FailureException(e); }
    finally {/* code for cleanup */ }
}
```

```
public void checkout(String repository, String workspace, String tmp)
                throws CheckoutException {
        try {
                makeSnapshot(workspace, tmp);
                download(repository, workspace);/* may throw an IOException */
        } catch (IOException e) {
                restore(tmp, workspace);
                throw new CheckoutException(e);
        } finally {
                dropSnapshot(tmp);
        }
}
```

- public FileCheckpoint{
- private String _workspace = null;
- private String _tmp = null;
- public FileCheckpoint(String workspace, String tmp) {
- /* constructor */};
- public void establish(){/* code for establishing */};
- public void restore(){/* code for restoring */};
- public void drop(){/* code for dropping */};
- }

```
public void checkout(String repository, String workspace,String tmp)
throws CheckoutException{
        FileCheckpoint fcp = new FileCheckpoint(workspace, tmp);
        try {
                fcp.establish();
                /* may throw an IOException */}
                download(repository, workspace);
        catch (IOException e){
                fcp.restore();
                throw new CheckoutException(e);
        }
        finally {
                fcp.drop();
        }
}
```

```
public void foo () throws FailureException{
    try {/*code that may change the state of the object*/ }
    catch (AnException e) { throw new FailureException(e); }
    finally {/ *code for cleanup*/ }
}
```

↓

```
public void foo () throws FailureException{
    Checkpoint cp = new Checkpoint (/*parameters*/);
    try {
        cp. establish ();/* establish a checkpoint */
        /*code that may change the state of the object*/ }
    catch (AnException e){
        cp.restore ();/*restore the checkpoint*/
        throw new FailureException(e); }
    finally { cp.drop(); }
}
```

Bad Smell

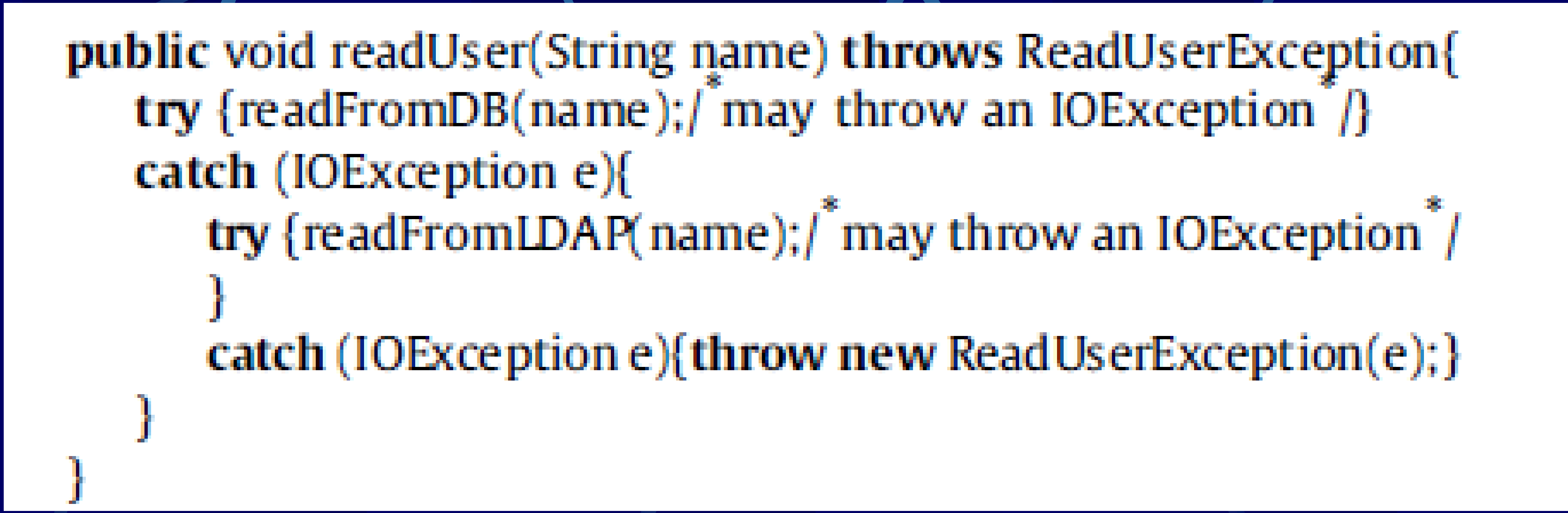
- Catch clause as spare handler

```
try {
    /* primary */
}
catch (SomeException e){
    try {
        /* alternative */
    }
    catch(AnotherException e){
        throw new FailureException(e);
    }
}
```

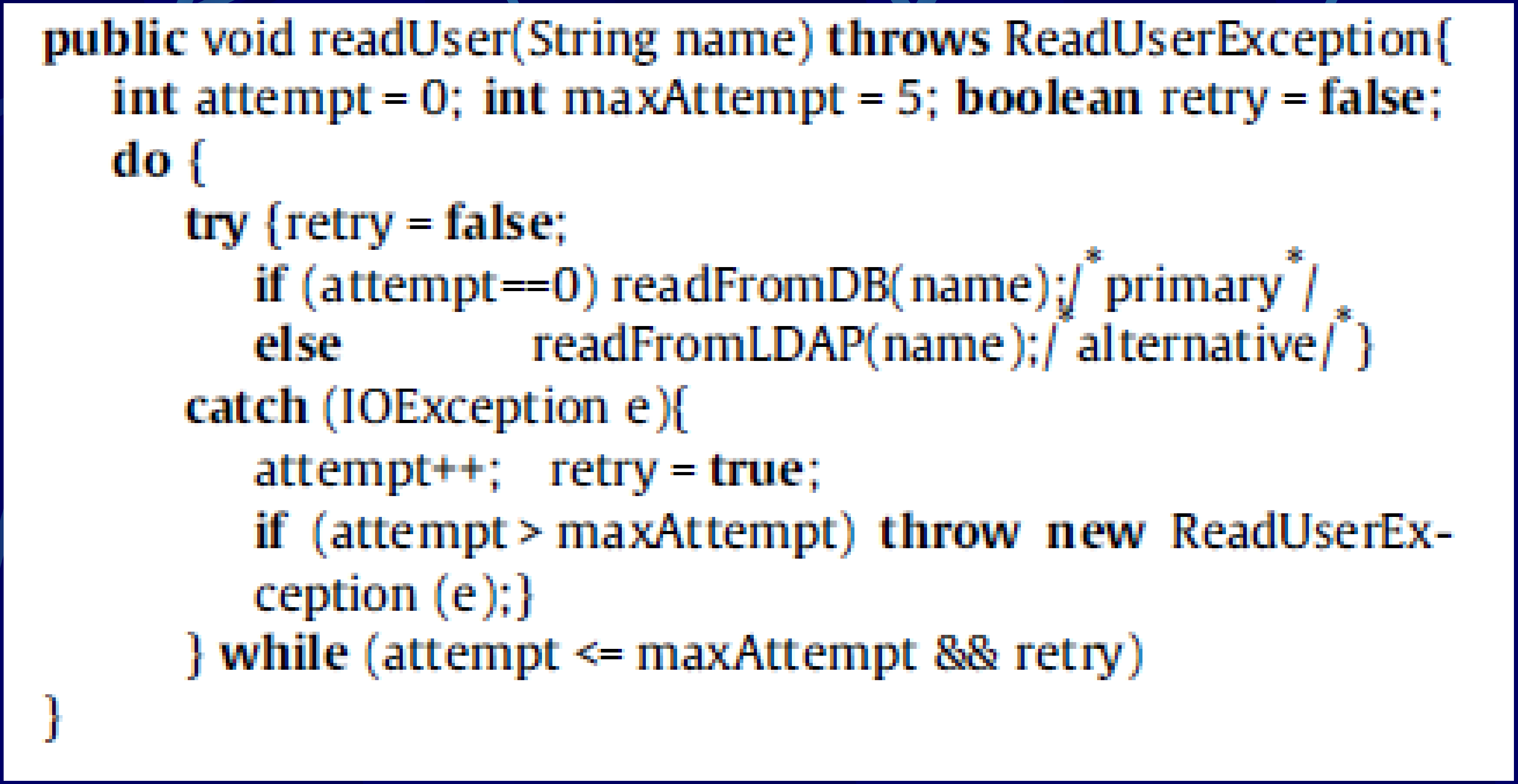
Refactoring

- Introduce resourceful try clause

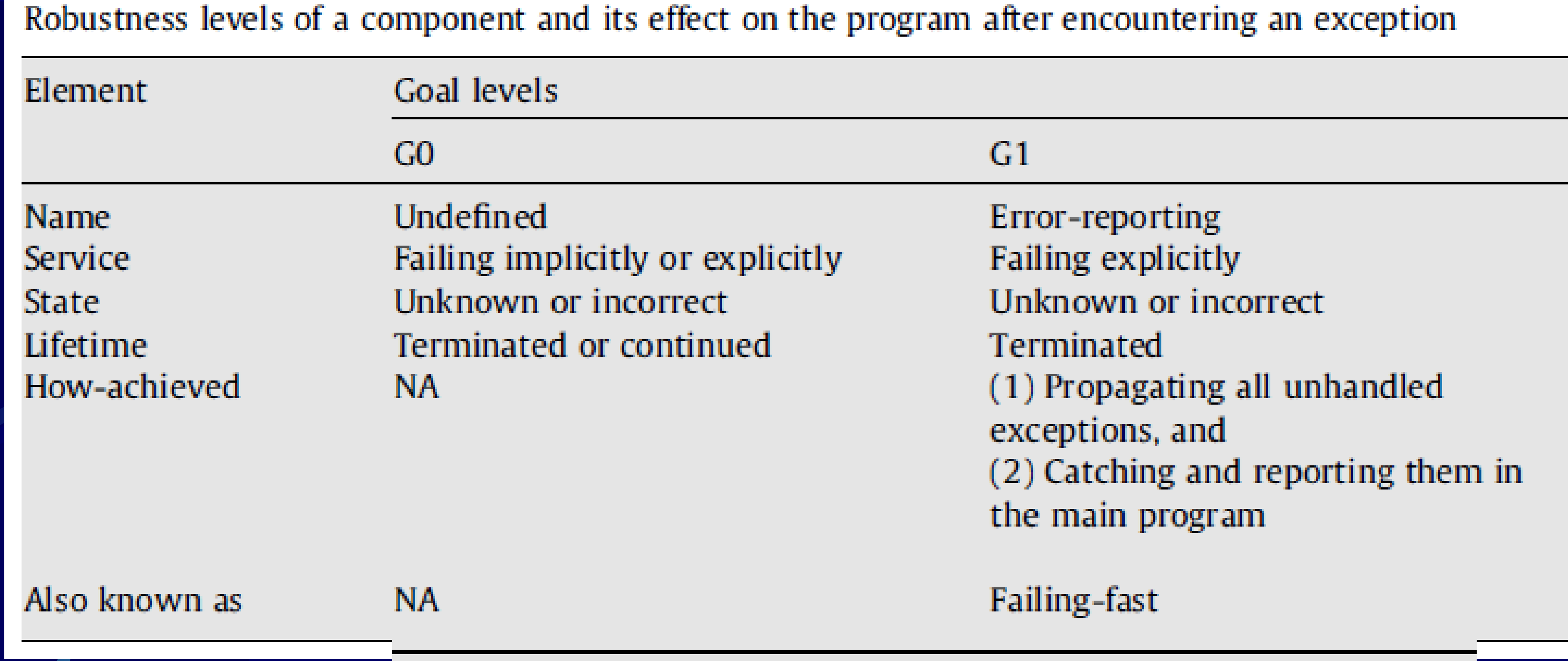
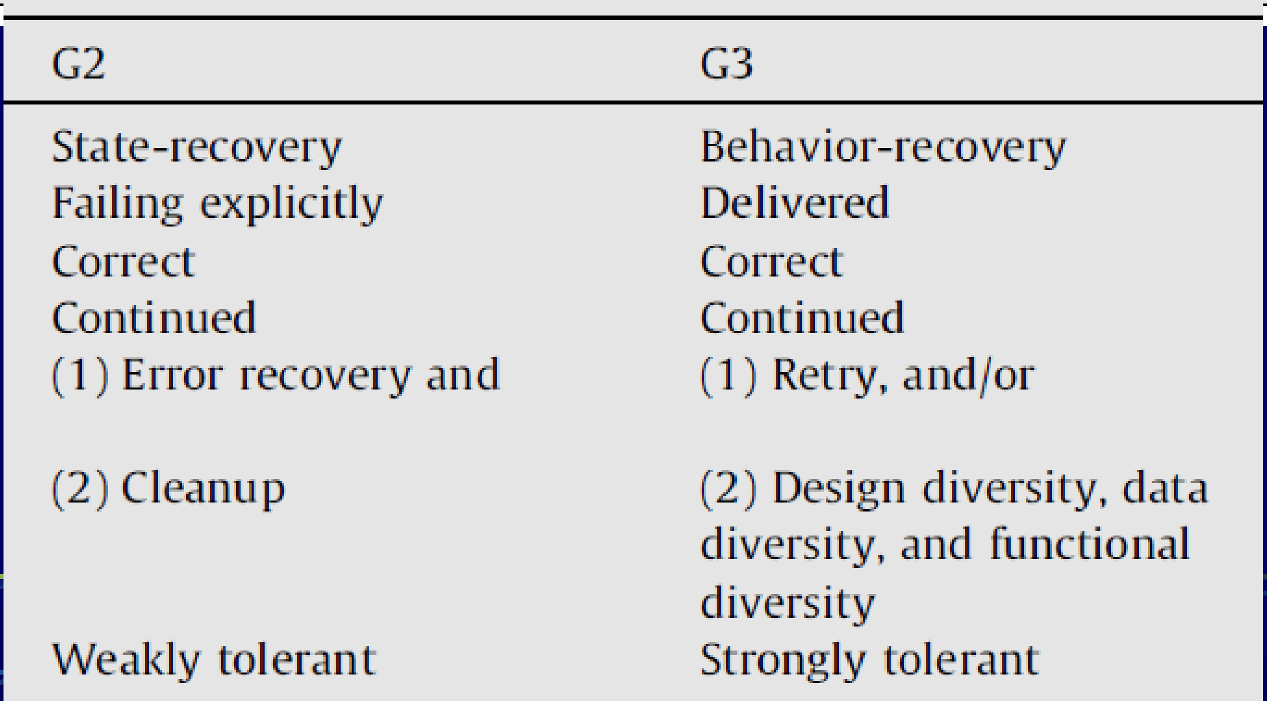
```
int attempt = 0; int maxAttempt = 2; boolean retry = false;
do{
        try {
                retry = false;
                if(attempt==0) {
                        /* primary */
                }
                else {
                        /* alternative */
                }
        }
        catch (SomeException e){
                attempt++;
                retry = true;
                if (attempt > maxAttempt)
                        throw new FailureException(e);
        }
}while (attempt <= maxAttempt && retry)
```



```
public void readUser(String name) throws ReadUserException{
    try {readFromDB(name);/*may throw an IOException */}
    catch (IOException e){
        try {readFromLDAP(name);/*may throw an IOException */
        }
        catch (IOException e){throw new ReadUserException(e);}
    }
}
```



```
public void readUser(String name) throws ReadUserException{
    int attempt = 0; int maxAttempt = 5; boolean retry = false;
    do {
        try {retry = false;
            if (attempt==0) readFromDB(name);/*primary*/
            else            readFromLDAP(name);/*alternative*/}
        catch (IOException e){
            attempt++;   retry = true;
            if (attempt > maxAttempt) throw new ReadUserEx-
            ception (e);}
        } while (attempt <= maxAttempt && retry)
}
```



Robustness levels of a component and its effect on the program after encountering an exception

Element	Goal levels			
	G0	G1	G2	G3
Name	Undefined	Error-reporting	State-recovery	Behavior-recovery
Service	Failing implicitly or explicitly	Failing explicitly	Failing explicitly	Delivered
State	Unknown or incorrect	Unknown or incorrect	Correct	Correct
Lifetime	Terminated or continued	Terminated	Continued	Continued
How-achieved	NA	(1) Propagating all unhandled exceptions, and (2) Catching and reporting them in the main program	(1) Error recovery and (2) Cleanup	(1) Retry, and/or (2) Design diversity, data diversity, and functional diversity
Also known as	NA	Failing-fast	Weakly tolerant	Strongly tolerant

- Java standard use
- an unchecked exception
  - indicates a bug
  - implement failing-fast to achieve G1 by doing nothing about it and allowing your program to abort
- checked exception
  - represents an error that your program should deal with

- in theory
  - standard use seems a sound principle
- in practice
  - it is easy for programmers to mishandle checked exceptions
  - Catching exception and doing nothing about it
  - unchecked exceptions are preferred in several well-known open source projects
    - Eclipse SWT project (http://www.eclipse.org)
    - Spring framework (http://www.springframework.org).

- wrapping a checked exception into an unchecked exception constitutes a legitimate exception handling strategy that achieves G1
- Checked exceptions are best used when the goal is to achieve G2 or G3 because the compilers will remind you about an uncaught checked exception

- three general steps
  - (1) identifying code smellst hat degrade the design,
  - (2) applying refactorings to remove the code smells,
  - (3) verifying satisfaction with the refactored program

- two categories
- Small refactorings or primitive refactorings
  - Introduce individual changes such as renaming a method or relocating a method
- big refactorings or composite refactorings,
  - Apply a coherent series of small refactorings to achieve a larger design goal such as introducing a design pattern and untangling an inheritance mess

- Most existing refactorings focus on improving the software design pertaining to the normal behavior of a system.
  - refactoring is only requested not to alter the external normal behavior
- EH refactorings enhance the system’s robustness by possibly changing the exceptional behavior but without altering the system’s normal behavior