Error Handling in MARY TTS

Marc Schröder

marc.schroeder@dfki.de

DFKI Language Technology Lab,
Saarbrücken, Germany

Why this presentation?

- Raise awareness of error handling as an important issue
- Define conventions to follow when writing MARY TTS code
- Improve code quality in MARY TTS in the long run

Outline

- Concepts and suggestions for best practice
  - Bugs vs. expected vs. unexpected error conditions
  - Method contracts
  - Throwing exceptions is communication!
    - Recipients of error messages
    - Which exceptions to throw where
  - Levels of guarantee
    - fundamental, basic, rollback, no-throw guarantee
- Conventions specific to MARY TTS

Examples of bad style error handling

- The never-ever-do-it sin-of-all-sins:

```
try {
   ...
} catch (Exception e) {
    e.printStackTrace();
}
```

Examples of bad style error handling

- What's wrong with

```
try {
   ...
} catch (Exception e) {
    e.printStackTrace();
}
```

?

  - Error is de facto ignored, processing continues
    - quality of service is unpredictable
    - violates the “fundamental guarantee”
  - no distinction expected/unexpected errors
    - no recovery strategy for expected errors
    - no global processing possible for unexpected errors
  - Error information does not end up in the log file
    - decreases the chance of sys admins and programmers learning about the error
- Worse than no exception handling at all

Bugs vs. expected vs. unexpected error conditions

Principles of error handling in software

- Two types of errors in software
  - Bugs in the code
  - Data does not have expected format
- If errors are not handled, program behaviour will be unpredictable
- Two goals of error handling
  - preserve quality of service to the extent possible
  - identify the source of the error in order to fix it

Suggestions for best practice

- First questions to ask yourself
  - Am I dealing with a bug or a wrong data format?
    - Error messages on bugs must be informative for the programmer
    - Error messages on data format must be informative for the user
  - Given this error condition,
    - can the program as a whole continue or should it abort?
    - can the current method continue or should it abort?

Suggestions for best practice

- General strategies
  - 1. Use “assert” statements to test for bugs
    - “no matter what the input was, if I get here, the following must be true”
    - very powerful and easy to use tool, use frequently!
    - preconditions of private methods within the class
    - AssertionError is expected to go high up the call stack, either aborting the program or being caught at a high level
    - never use “assert” to test for wrong data formats

```
assert interpolatedLogF0.length == logF0.length;
```

or

```
assert interpolatedLogF0.length == logF0.length : "interpol not same length";
```

Expected vs. unexpected error conditions

- A key distinction
  - **expected** error conditions are locally predictable deviations from normal processing
    - e.g., a method returns null instead of an object;
      a string cannot be converted into a floating point number
    - often related to boundary cases of data processing
    - meaningful continuation of processing may be possible
  - **unexpected** error conditions cover all the rest
    - OutOfMemoryError
    - Exceptions bubbling up from methods I call
    - … what else?

Suggestions for best practice

- General strategies
  - 2. Throw informative exceptions when testing expected errors
    - wrong data formats
    - expected but unhandleable boundary conditions during processing

```
} catch (Exception ex) {
    throw new MaryConfigurationException("Cannot build unit selection voice '"+name+"'", ex);
}
```

Examples of bad style error handling

- Expected errors: Ignoring boundary conditions
  - MARY TTS ticket:339
    - F0PolynomialFeatureFileWriter throws a NullPointerException in getInterpolatedLogF0Contour()

```
double[] rawLogF0 = getLogF0Contour(s, f0FrameSkip);
double[] logF0;
if (interpolate) {
    logF0 = getInterpolatedLogF0Contour(rawLogF0);
} else {
    logF0 = rawLogF0;
}
```

Examples of bad style error handling

- Expected errors: Ignoring boundary conditions
  - MARY TTS ticket:339
    - F0PolynomialFeatureFileWriter throws a NullPointerException in getInterpolatedLogF0Contour()

```
double[] rawLogF0 = getLogF0Contour(s, f0FrameSkip);
double[] logF0;
if (interpolate) {
    logF0 = getInterpolat                ntour(rawLogF0);
} else {
    logF0 = rawLogF0;
}
```

```
/**
 * For the given sentence, obtain a log f0 contour.
 * [...]
 * @return a double array representing the F0 contour, sampled at skipSizeInSeconds,
 * or null if no f0 contour could be computed
 */
private double[] getLogF0Contour(Sentence s, double skipSizeInSeconds)
throws IOException {
```

Examples of bad style error handling

- Expected errors: Ignoring boundary conditions
  - this may be the most frequent type of bad programming as a consequence of sloppiness

```
String durString = phoneElement.getAttribute(“d”);
int dur = Integer.parseInt(durString);
```

```
String durString = phoneElement.getAttribute(“d”);
int dur;
try {
    dur = Integer.parseInt(durString);
} catch (NumberFormatException nfe) {
    log.debug(“Duration '”
        +durString+“' is not an integer”, nfe);
    dur = 0;
}
```

Assume default

Escalate but explain

```
String durString = phoneElement.getAttribute(“d”);
int dur;
try {
    dur = Integer.parseInt(durString);
} catch (NumberFormatException nfe) {
    throw new SynthesisException(“Duration '”
       +durString+ “' is not an integer”, nfe);
}
```

Method contracts

Examples of bad style error handling

- Expected errors: Not documenting boundary conditions
  - Makes it hard for users of your method to handle expected errors

```
/**
 * Get the datagrams spanning a particular time range from a particular time location,
 * given in the timeline's sampling rate.
 *
 * @param targetTimeInSamples the requested position, in samples given
 * the timeline's sample rate.
 * @param timeSpanInSamples the requested time span, in samples given
 * the timeline's sample rate.
 *
 * @return an array of datagrams
 */
public Datagram[] getDatagrams(long targetTimeInSamples, long timeSpanInSamples)
throws IOException {
    return getDatagrams(targetTimeInSamples, timeSpanInSamples, sampleRate, null);
}
```

Examples of bad style error handling

- Expected errors: Not documenting boundary conditions
  - Makes it hard for users of your method to handle expected errors

Can this return null? An empty array?

What if I give it an invalid time argument?

Under what circumstance will it throw an IOException?

```
/**
 * Get                                                          tion,
 * giv
 *
 * @pa
 * the
 * @pa
 * the timeline's sample rate
 *
 * @return an array of datagrams
 */
public Datagram[] getDatagrams(long targetTimeInSamples, long timeSpanInSamples)
throws IOException {
    return getDatagrams(targetTimeInSamples, timeSpanInSamples, sampleRate, null);
}
```

Examples of bad style error handling

- Expected errors: Not documenting boundary conditions
  - Makes it hard for users of your method to handle expected errors

```
/**
 * Get the datagrams spanning a particular time range from a particular time location,
 * given in the timeline's sampling rate.
 *
 * @param targetTimeInSamples the requested position, in samples given
 * the timeline's sample rate. Must not be negative.
 * @param timeSpanInSamples the requested time span, in samples given
 * the timeline's sample rate.
 *
 * @return an array of datagrams containing at least one datagram
 * @throws IllegalArgumentException if targetTimeInSamples is negative
 * @throws IOException if there is a problem reading data from the underlying file
 */
public Datagram[] getDatagrams(long targetTimeInSamples, long timeSpanInSamples)
throws IllegalArgumentException, IOException {
    return getDatagrams(targetTimeInSamples, timeSpanInSamples, sampleRate, null);
}
```

Suggestions for best practice

- General strategies
  - 3. Define a “contract” for each method you write
    - What is the meaning of each parameter? What parameter values are acceptable?
    - What will the method do if it gets parameters that violate the contract?
    - What are the possible return values of the method? Also document the boundary cases:

      “will return null / the empty string / an empty set / … if no such element can be found”
    - Which exceptions will the method throw?
  - Document the contract in the method's javadoc

Throwing exceptions is communication!

Recipients of error messages

- Who are error messages directed to?
  - users
    - want system to work, or at least be in a well-defined state
  - system administrators
    - need to know what went wrong, and whether they can do anything to fix it
  - programmers
    - need detailed information to find and fix bugs

Credit: Software Engineering Radio
Episode 7: Error Handling

Suggestions for best practice

- Which exceptions to throw where?
  - When should a method just declare that it throws low-level exceptions, and when should it wrap them into well-defined “interface” exceptions?
  - Draw a clear line between internal “business logic” of a processing unit and the “interface” between processing units
    - within the business logic, let Exceptions bubble up (e.g., within private and protected methods)
    - when moving from one processing unit to another (e.g. from a module to a request handler), wrap an Exception into a meaningful explanation
    - only at user interface, communicate the Exception to the user

Which exceptions to throw where?

- Outermost layer: user interface
  - catch everything, make sure it is communicated to the log and to the user

```
void RequestHandler.run() {
    try {
        module.process(inputData);
    } catch (Throwable t) {
        log.info("Module "+module.getName()+" cannot process", t);
        clientInterface.reportFailure(t);
    }
}
```

Which exceptions to throw where?

- Intermediate layer: interface between processing units
  - cast everything into a meaningful wrapper

```
public MaryData process(MaryData input) throws SynthesisException {
    try {
        // actual processing steps on input
    } catch (Exception e) {
        throw new SynthesisException("Problem processing the following input:\n"
            + input.toString(), e);
    }
    return output;
}
```

Which exceptions to throw where?

- Intermediate layer: interface between processing units
  - cast everything into a meaningful wrapper

```
public MaryData process(MaryData input) throws SynthesisException {
    try {
        // actual processing steps on input
    } catch (Exception e) {
        throw new SynthesisException("Problem processing the following input:\n"
            + input.toString(), e);
    }
    return output;
}
```

Make sure to include the cause!

Which exceptions to throw where?

- Business logic layer: Where you do the actual stuff
  - private and protected methods
  - arbitrary depth
  - throw everything, don't catch
    - unless wrapping adds important information about the meaning of the Exception

```
private Point getNextPoint(Point previous) throws IOException, SAXException, AnyOtherException {
    // individual calls to whatever processing logic is needed
}
```

Levels of guarantee

Levels of Guarantee

- Possible “promises” a piece of code can make
  - “**fundamental guarantee**”: no resource leaks, no code accessible that is in an undefined state
    - initialisation in constructors prevents code in undefined state
  - “**basic guarantee**”: if something goes wrong, the code can still be called
    - easier with “stateless” objects: no global non-constant variables – if a method call fails due to bad input, the next one can succeed
  - “**rollback guarantee**”: if something goes wrong, we return to the state before the call
    - facilitated by immutable objects
  - “**no-throw guarantee**”: promise never to throw any exception
    - important in some embedded environments: flight control, pacemaker, ...

Examples of bad style error handling

- Code structure that violates the fundamental guarantee: uninitialised class

```
public class MyClass {

    public MyClass() {
    }

    public void load(String filename) throws IOException {
    ...
    }

    public MyResult compute(MyInput input) {
    ...
    }
}
```

Examples of bad style error handling

- Code structure that violates the fundamental guarantee: uninitialised class

```
public class MyClass {

    public MyClass() {
    }

    public void load(String filename) throws IOException {
    ...
    }

    public MyResult compute(MyInput input) {
    ...
    }
}
```

What if load() is never called? What if it throws an exception?

Examples of bad style error handling

- Loading from constructor means that, if an object is ever created, it is initialised
  - "new MyClass(filename)" never returns a reference to an object if an exception is thrown

```
public class MyClass {

    public MyClass(String filename) throws IOException {
        load(filename);
    }

    private void load(String filename) throws IOException {
    ...
    }

    public MyResult compute(MyInput input) {
    ...
    }
}
```

Examples of bad style error handling

- Code structure that violates the basic guarantee:

Levels of Guarantee

- Possible “promises” a piece of code can make
  - “**fundamental guarantee**”: no resource leaks, no code accessible that is in an undefined state
    - initialisation in constructors prevents code in undefined state
  - “**basic guarantee**”: if something goes wrong, the code can still be called
    - easier with “stateless” objects: no global non-constant variables – if a method call fails due to bad input, the next one can succeed
  - “**rollback guarantee**”: if something goes wrong, we return to the state before the call
    - facilitated by immutable objects
  - “**no-throw guarantee**”: promise nev exception
    - important in some embedded environme pacemaker, ...

Think about which of these guarantees your code can give!

Conventions for error handling in MARY TTS

Conventions for error handling in MARY TTS

- In MARY TTS server, use fail-early strategy during startup
  - any unexpected data conditions found during the startup procedure must be escalated to the top of the call stack
  - => test for unexpected data conditions at startup!
  - Classpath, config files, language resources, voice data files
  - This follows the logic of the fundamental guarantee: if the MARY server starts up, it is expected to be fit for service.

Conventions for error handling in MARY TTS

- Key “interface” exceptions in MARY TTS
  - MaryConfigurationException
    - “The server is not configured properly, it cannot run.”
    - Should only be thrown at startup time (fail-early principle)
  - SynthesisException
    - “This request cannot be handled.”
    - Should be the only exception thrown by MaryModule (this is currently not the case)

Types of “guarantees” in MARY TTS code

- Overall server: basic guarantee
  - fundamental: if it starts, it's supposed to work
  - +basic: when a request fails, the server still works
- MaryModule: basic guarantee + thread-safe
  - fundamental: if the module constructor and startup() succeed, the module is operational
  - +basic: when a one call to process() fails, the module can still take new calls to process()
  - +thread-safe: several process() calls can run in parallel

Summary

- Handle errors by asking yourself:
  - is it a bug, an expected data format deviation, or an unexpected condition?
    - assert
    - let bubble up within business logic
    - wrap in “interface exceptions” at interfaces
    - output to log and user?
  - is my method contract clearly defined in the Javadoc?
  - have I addressed all boundary conditions?
  - which guarantees can I give: fundamental, basic, rollback?
- Follow the conventions!
  - errors will be fewer
  - finding errors will be easier

Thank you for your attention!

http://mary.dfki.de