# Lecture 23

ECE 1145: Software Construction and Evolution

## Error Handling

# Announcements

- Iteration 8 (last one!): Frameworks and MiniDraw due Dec. 12
  - My recommendations:
    - **THIS WEEK**: Frameworks (36.36), MiniDraw Integration (test out gradle tasks), Subject behavior (36.37), Observer updates (36.38)
    - **NEXT WEEK**: Tool development (36.39-40, 42-44), SemiCiv GUI
- OMET Teaching Surveys open until Dec. 12
  - Link in Canvas navigation
- Office Hours Thurs. Dec. 2 10:30 – 11:00 AM
- Final Exam: 12:00 AM Wed. Dec. 15 – 11:59 PM Fri Dec. 17 (similar format as midterm)
  - Office hours during scheduled exam time 10:00 – 11:50 AM Wed. in 1211A

# Questions for Today

How do we detect, log, and notify when errors occur?

# What is Error Handling?

- Detection of a problem
- A message showing detection of the problem
- A way to resolve / log / act in response to the problem
- A way to make a program more robust
- A way to make a “project” into a “product”

# Error Handling: GUIs

Error handling is especially important for user interfaces

→Don't allow the whole application to crash

→Provide an informative message to the user, maybe they can fix the problem?

# Error Handling: Try/Catch, Throw

## Language support for exception handling:

```
try {
    // attempt some operation
} catch (FileNotFoundException e) {
    // handle different exceptions different ways
    // or just notify the user/ client
}
…   // other catch blocks for other error types

} finally {
    // guaranteed to run
    // close files, recover resources, etc.
}
```

```
/** The CivDrawing should not allow client side
 * units to add and manipulate figures; only figures
 * that renders game objects are relevant, and these
 * should be handled by observer events from the game
 * instance. Thus this method is 'killed'.
 */
public Figure add(Figure arg0) {
  throw new RuntimeException("Should not be used...");
}
```

# Error Handling: Try/Catch, Throw

Grouping/differentiating error types:

```
catch (FileNotFoundException e) {
    ...
}

catch (IOException e) {
    ...
}
```

```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

```
/** The CivDrawing should not allow client side
 * units to add and manipulate figures; only figures
 * that renders game objects are relevant, and these
 * should be handled by observer events from the game
 * instance. Thus this method is 'killed'.
 */
public Figure add(Figure arg0) {
  throw new RuntimeException("Should not be used...");
}
```

```
public class RuntimeException extends Exception {
```

```
// A (too) general exception handler
catch (Exception e) {
    ...
}
```

# Error Handling: Try/Catch, Throw

Benefits:

- Keep error handling code out of main program logic
- Readability

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html

# Error Handling: Try/Catch, Throw

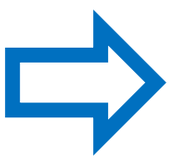
Benefits:

- Keep error handling code out of main program logic
- Readability
- Propagate errors up the call stack (more later), caller may know better what to do
- Consolidate response actions (cohesion)

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```



```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

# Error Handling: Try/Catch, Throw

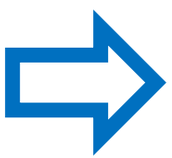
Liabilities:

- Error handling code is in a different place than the error occurrence (lower cohesion)
- What if the caller doesn't know what the error means?
- Inconsistent handling by callers

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```



```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

# Error Handling: Testing

Error handling code is often less tested!

→Write error-handling test cases

→Code coverage analysis can help (but recall that branch coverage won't necessarily include error paths)

# Error Handling: Options

Return a neutral value

→ Continue operation, return empty string, 0, etc.

Substitute the next piece of valid data

→ e.g., processing a stream of data

Return the same answer as the previous time

→ e.g., sampling, reading from sensors

Substitute the closest legal value

→ Making assumptions may be dangerous

# Error Handling: More Options

Log a message to a file

→ Keep a record of errors

Return an error code

→ May trigger an exception in higher-level modules

Call a centralized error processing routine

→ May be hard to reuse

Display an error message

→ User interfaces

Shut down?

→ Depends on system, context, severity of the error

# Error Handling: User

Ask the user for input

→ Assumes a ”user” who can respond

Procedure:

1. Detect the error
   - May be anticipation of a likely error, e.g., overwriting a file
2. Give the user choices or ask for input
3. Retry the action
4. Have an “exit strategy” if the user gives up

# Error Handling: Forward Control

Propagate errors up the call stack

→Exceptions at a lower level, assumes higher levels know what to do in response

# Error Handling: Guidelines

- In general, try to do something expected
- Communicate (with the user, client, etc.)
- Strike a balance

**No error handling**
- Fragile
- Buggy
- Opaque

**Check for everything**
- Robust
- Costly
- Impractical (Impossible?)



# Error Handling: Types of Errors

1. Problems with external data or conditions
   - Notify the user/client, don't crash
2. Internal errors (bad arguments, out of memory, unexpected result from a function call)
   - Debug: Notify programmer, then crash
   - Production: Don't crash, recover gracefully if possible

# Error Handling: Severity

Fatal errors

- Cannot continue execution (or it would be meaningless to)
  - e.g., out of memory

Nonfatal (for now)

- Potentially fatal later
- Start recovery strategy, inform user/client before it is too late

Nonfatal

- Recovery is possible, may want to inform anyway

# Error Handling: More Guidelines

Depending on type and severity:

- Catch if possible
- Handle specifically but systematically
- Log (System.out, logging framework, file…)

# Error Handling: More Guidelines

Depending on type and severity:

- Catch if possible
- Handle specifically but systematically
- Log (System.out, logging framework, file…)

Even before that:

- Anticipate likely errors, try to avoid them in the first place
- Weigh the risk of not handling vs. cost of handling exceptions
  - Sometimes better to abort
  - Don’t ignore or assume errors won’t happen (without good cause)

# Defensive Programming

Like “defensive driving”, but for code:

- Expect “unexpected” problems
- Protect against bad inputs
  - Function arguments, file contents, user input


https://xkcd.com/327/

# Defensive Programming

**Assertions**: Check that everything is operating as expected (error if not)

- Document assumptions made in code, pre/post-conditions
- Intended to be silent, things that should never occur

# Defensive Programming

**Exceptions**: Notify other parts of the program about errors that should not be ignored

- Only for conditions that are really “exceptional”
- If it could be ignored, use a code instead
- If possible to handle locally, do that

# Defensive Programming

**Exceptions**: Notify other parts of the program about errors that should not be ignored

- Only for conditions that are really “exceptional”
- If it could be ignored, use a code instead
- If possible to handle locally, do that

Don’t throw errors in constructors and destructors!

Throw at the right level of abstraction

- Don’t expose implementation details
- Catch exceptions from lower levels and translate

# Exception Handling: Chaining

**Chaining**: Associations between exceptions

- Throw a “custom exception” within an exception explaining what it means in context

# Exception Handling: Chaining

```
try{
    // setting up some connection
    // with lots of risky I/O operations!
}
catch (java.io.IOException rootCause){
    throw new ConnectionException(rootCause);
}
```

```
public class ConnectionException extends java.io.IOException{
    public ConnectionException(Throwable cause){
        super(cause)
    }
}
```

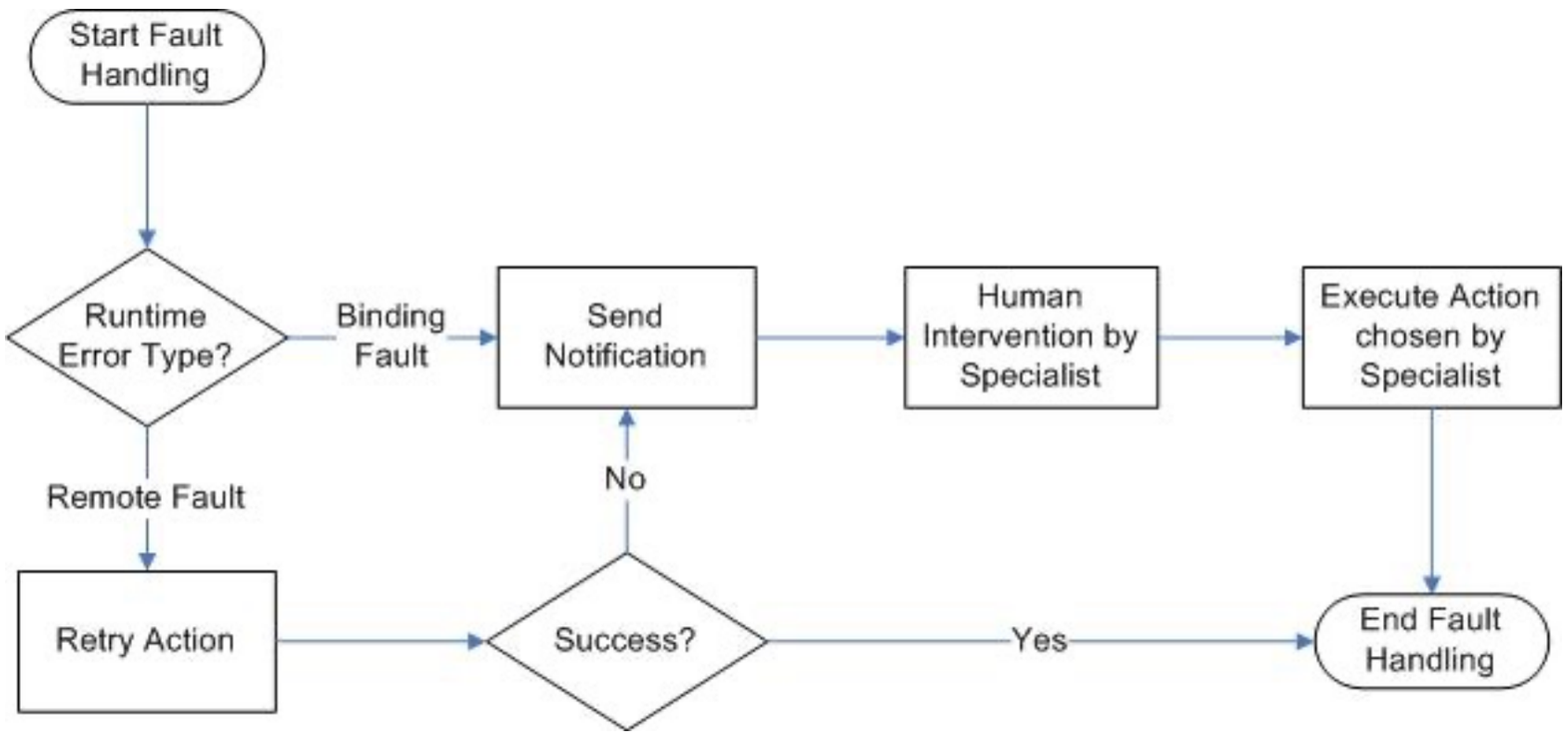
# Exception Handling: Chaining

```
try{
    // setting up some connection
    // with lots of risky I/O operations!
}
catch (java.io.IOException rootCause){
    throw new ConnectionException(rootCause);
}
```

- Stack trace readability
- Throwable.getCause()
- Custom exception handling

```
public class ConnectionException extends java.io.IOException{
    public ConnectionException(Throwable cause){
        super(cause)
    }
}
```

# Exception Handling: Chaining




https://technology.amis.nl/amis/extending-the-oracle-bpel-error-hospital-with-custom-java-actions/

# Error Handling: FMEA

**FMEA: Failure Mode and Effects Analysis:** A systematic process of identifying potential failure modes of parts of a system, effects of those failures, and actions to prevent or mitigate failures

1. How could each part of the system fail?
2. What are the effects of each failure?
3. What is the risk and severity of each failure?
4. How likely is each failure to be detected?
5. Rank by highest risk and severity, least likely to be detected
6. Reduce the severity, occurrence, or improve detection
7. Reevaluate

# Error Handling: FMEA

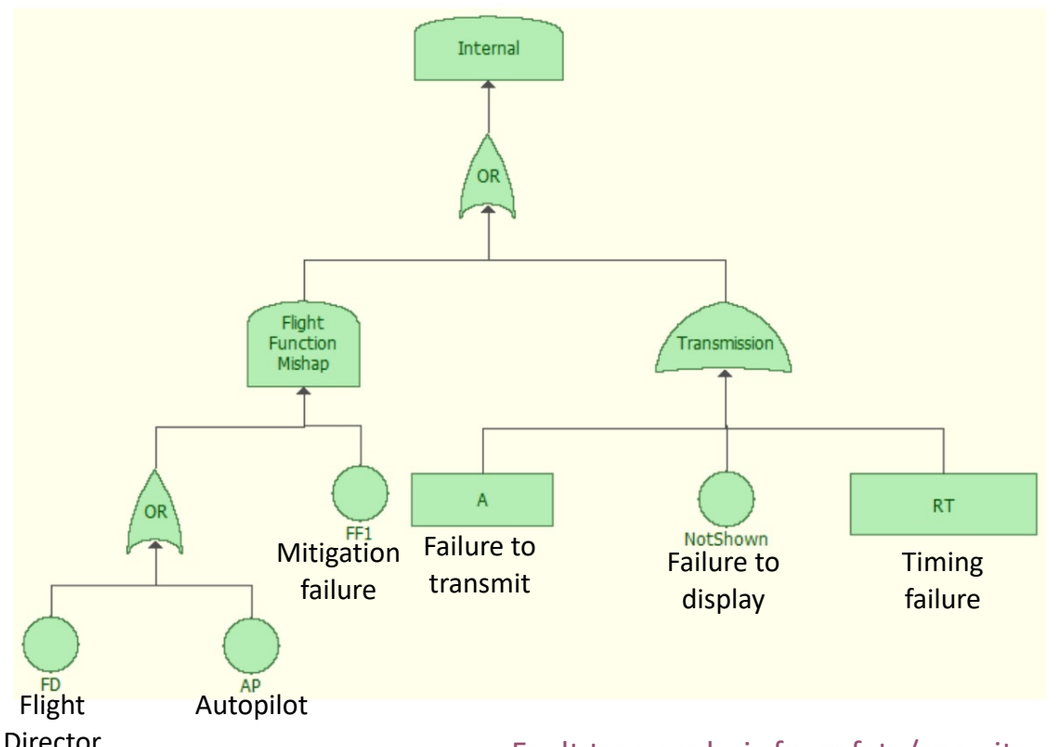
1. Identify components of the system and their functions and failure modes
   - How could each module/class/variable “fail”?
   - Failure to execute, incomplete execution, wrong timing, incorrect value
2. What are the effects of failure on other components and the system?
   - For each failure mode
3. What is the risk (probability) of occurrence and severity of each failure?
   - What would cause each failure? How likely is each cause?
   - Relative ranking
4. How likely is the failure to be detected by current/planned controls?
   - Detect causes or failure
   - Relative ranking, higher is **less likely** to be detected
5. Rank by the Risk Priority Number (RPN)
   - RPN = severity x occurrence x detection
6. Take action to reduce the severity, occurrence, or detection rankings of the highest RPN
7. Reevaluate

https://peer.asee.org/applying-fmea-to-software

# Error Handling: FTA

**Fault Tree Analysis:** Identify top-level failure modes first, backtrack to list possible causes and chains of events, combine with logic AND, OR

- Top-down approach




Fault tree analysis for safety/security verification in aviation software

# Error Handling: Summary

- Anticipate likely errors, try to avoid them in the first place
- Handle errors in context, preferably in the same place they were detected
  - May need to pass up to the caller instead
- Weigh the risk of not handling vs. cost of handling exceptions
  - Evaluate probability/severity of errors
- Handle errors systematically/consistently across the system
  - Have a planned strategy