PW SKILLS

Logging and Debugging

Lesson Plan

1. Introduction

Logging and debugging are essential practices in software development. Logging provides a record of events that occur during the execution of an application, while debugging involves investigating and fixing code issues. Together, they help developers ensure that software runs smoothly and issues are addressed promptly.

2. Importance of Logging

Logging plays a critical role in monitoring and troubleshooting applications. It provides visibility into the application's behavior and helps in diagnosing problems. Here's why logging is important:

2.1. Monitoring and Analysis

Logging helps track the application's performance and behavior over time. For instance, a web server might log every request it handles, which can be analyzed to identify performance trends or unusual activity.

Example:

```
import logging

# Configure logging
logging.basicConfig(filename='app.log', level=logging.INFO)

# Log an informational message
logging.info('Application has started')
```

2.2. Error Diagnosis

When errors occur, logs provide context that helps in diagnosing the problem. For example, if a web application crashes, the logs can reveal what the application was doing at the time.

Example:
python

Copy code

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    logging.error('An error occurred: %s', e)
```

2.3. Audit Trails

Logs can serve as an audit trail, recording user actions and system changes. This is especially important in applications that handle sensitive data.

Example:
python

Copy code

```
def user_login(username):
    logging.info('User %s logged in', username)
```

3. Effective Logging Practices

To make the most of logging, follow these best practices:

3.1. Log Levels

Use different log levels to categorize messages by importance:

- **DEBUG –** It is used to provide detailed information and only use it when there is diagnosing problems.
- **INFO –** It provides the information regarding that things are working as we want.
- **WARNING –** It is used to warn that something happened unexpectedly, or we will face the problem in the upcoming time.
- **ERROR –** It is used to inform when we are in some serious trouble, the software hasn't executed some programs.
- **CRITICAL –** It specifies the serious error, the program itself may be incapable of remaining executing.

The above levels are sufficient to handle any types of problems. These corresponding numerical values of the levels are given below.

Level	Numeric Values
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Example:

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

3.2. Log Formats

Choose a consistent format for logs to make them easier to analyze. Common formats include:

- **Plain Text:** Simple and human-readable.
- **JSON:** Structured and suitable for automated processing.
- **XML:** Provides a hierarchical structure but is more verbose.

Example:

```
import logging

# Configure logging to output JSON format
import json_log_formatter
formatter = json_log_formatter.JSONFormatter()
handler = logging.StreamHandler()
handler.setFormatter(formatter)

logger = logging.getLogger()
logger.addHandler(handler)
logger.setLevel(logging.INFO)

logger.info('User login', extra={'user': 'john_doe'})
```

3.3. Log Rotation

To manage log file sizes and prevent disk space issues, implement log rotation. This involves archiving old logs and creating new ones.

Example:

```
import logging
from logging.handlers import RotatingFileHandler

# Create a rotating file handler
handler = RotatingFileHandler('app.log', maxBytes=2000,
backupCount=5)
logger = logging.getLogger()
logger.addHandler(handler)
logger.setLevel(logging.INFO)

logger.info('This is a log message')
```

3.4. Sensitive Information

Avoid logging sensitive information such as passwords or personal data. Mask or exclude such details to prevent security breaches.

Example:

```
def login(user, password):
    logging.info('User %s attempted to log in', user)
    # Avoid logging the password
```

3.5. Performance Considerations

Be aware that logging can impact application performance. Use asynchronous logging or adjust log levels to balance performance and logging needs.

Example:

```
import logging
import logging.handlers

# Asynchronous logging
queue_handler =
logging.handlers.QueueHandler(logging.Queue())
logger = logging.getLogger()
logger.addHandler(queue_handler)
logger.setLevel(logging.INFO)

logger.info('Asynchronous logging example')
```

What is Debugging?

In the development process of any software, the software program is religiously tested, troubleshot, and maintained for the sake of delivering bug-free products. There is nothing that is error-free in the first go.

So, it's an obvious thing to which everyone will relate that as when the software is created, it contains a lot of errors; the reason being nobody is perfect and getting error in the code is not an issue, but avoiding it or not preventing it, is an issue!

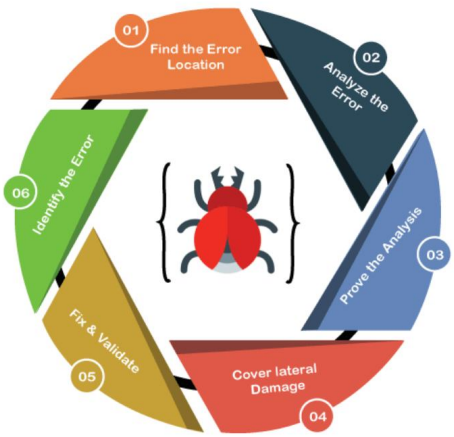
All those errors and bugs are discarded regularly, so we can conclude that debugging is nothing but a process of eradicating or fixing the errors contained in a software program.

Debugging works stepwise, starting from identifying the errors, analyzing followed by removing the errors. Whenever a software fails to deliver the result, we need the software tester to test the application and solve it.

Since the errors are resolved at each step of debugging in the software testing, so we can conclude that it is a tiresome and complex task regardless of how efficient the result was.

Steps involved in Debugging

Following are the different steps that are involved in debugging:



1. **Identify the Error:** Identifying an error in a wrong may result in the wastage of time. It is very obvious that the production errors reported by users are hard to interpret, and sometimes the information we receive is misleading. Thus, it is mandatory to identify the actual error.
2. **Find the Error Location:** Once the error is correctly discovered, you will be required to thoroughly review the code repeatedly to locate the position of the error. In general, this step focuses on finding the error rather than perceiving it.
3. **Analyze the Error:** The third step comprises error analysis, a bottom-up approach that starts from the location of the error followed by analyzing the code. This step makes it easier to comprehend the errors. Mainly error analysis has two significant goals, i.e., evaluation of errors all over again to find existing bugs and postulating the uncertainty of incoming collateral damage in a fix.
4. **Prove the Analysis:** After analyzing the primary bugs, it is necessary to look for some extra errors that may show up on the application. By incorporating the test framework, the fourth step is used to write automated tests for such areas.
5. **Cover Lateral Damage:** The fifth phase is about accumulating all of the unit tests for the code that requires modification. As when you run these unit tests, they must pass.
6. **Fix & Validate:** The last stage is the fix and validation that emphasizes fixing the bugs followed by running all the test scripts to check whether they pass.

4. Debugging Techniques

Effective debugging helps identify and fix issues in software. Here are some common techniques:

4.1. Reproducing the Issue

Try to reproduce the issue consistently to understand the conditions under which it occurs.

Example:
If a bug appears intermittently, create a test case that simulates the conditions under which the bug occurs.

4.2. Using Debuggers

Debuggers allow you to set breakpoints, step through code, and inspect variables.

Example:
In Python, using a debugger like pdb:

```
import pdb

def divide(a, b):
    pdb.set_trace()  # Set a breakpoint
    return a / b

divide(10, 0)
```

4.3. Print Statements

Use print statements to check the state of variables or application flow.

Example:

```
def process_data(data):
    print('Processing data:', data)
    # Further processing
```

4.4. Analyzing Core Dumps

Core dumps capture the state of an application at the time of a crash and can be analyzed to understand what went wrong.

Example:
Use tools like gdb to analyze core dumps:
gdb ./myapp core

4.5. Profiling

Profiling tools measure performance and identify bottlenecks.

Example:

```
Using Python's cProfile:
import cProfile

def main():
    # Code to profile
cProfile.run('main()')
```

5. Tools and Best Practices

5.1. Logging Tools

- **Log4j:** A logging library for Java.
- **ELK Stack:** For searching and analyzing log data.
- **Splunk:** For searching, monitoring, and analyzing data.

5.2. Debugging Tools

- **GDB:** GNU Debugger for various languages.
- **LLDB:** Part of the LLVM project.
- **Visual Studio Debugger:** Integrated in Visual Studio IDE.

5.3. Best Practices

- **Document Procedures:** Ensure that logging and debugging procedures are well-documented.
- **Review Logs Regularly:** Periodically review logs to identify and address issues.
- **Keep Debugging Sessions Focused:** Avoid making extensive changes during debugging to prevent new issues.

6. Conclusion

Effective logging and debugging are vital for maintaining high-quality software. By following best practices in logging and employing various debugging techniques, developers can improve their ability to monitor, diagnose, and fix issues in their applications. Continuous improvement in these areas contributes to more reliable and maintainable software.