20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by Nishant V
CB.EN.U4CYS21049
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

AMRITA
VISHWA VIDYAPEETHAM

Outline

Aspect-Oriented

Aspect-Oriented Modules:

- Aspect-oriented modules are units of code that contain aspects, pointcuts, and advice. These modules encapsulate cross-cutting concerns.
- Purpose: Organizing code into aspect-oriented modules enhances modularity, making it easier to manage and maintain.

Aspect-Oriented Programming Languages:

- Aspect-oriented programming languages, such as AspectJ, AspectC++, and Spring AOP, provide constructs and syntax for implementing AOP concepts.
- Purpose: These languages offer dedicated features to express aspects, pointcuts, and advice, facilitating the implementation of AOP principles..

Aspect-Oriented Design Patterns:

- Aspect-oriented design patterns are recurring solutions to common problems encountered in software design, applying AOP principles.
- Purpose: These patterns help developers apply aspect-oriented concepts effectively and promote best practices in designing modular and maintainable systems.

- **Aspect Declarations:**
  AspectC++ introduces the aspect keyword to declare aspects. An aspect encapsulates cross-cutting concerns.
- **Pointcuts and Join Points:**
  Pointcuts define the set of join points where aspects will be applied. Join points are specific points in the program execution, such as method calls or field access.
- **Advice:**
  Advice contains the code that will be executed at specified join points. It defines how the cross-cutting concern will be woven into the existing code.
- **Introduction:**
  AspectC++ allows the introduction of new members (fields, methods) to existing classes, enhancing the flexibility to add functionality.
- **Weaving:**
  Weaving is the process of integrating aspect code with the base code. AspectC++ supports both compile-time and link-time weaving.

AspectC++ Working

Printing "Hello World"

~/Downloads/aspectc++/examples/helloworld/world.ah - Mousepad

File Edit Search View Document Help

```
1 #ifndef __WORLD_AH__
2 #define __WORLD_AH__
3
4 #include <iostream>
5
6 aspect World {
7
8   advice execution("void hello()") : after() {
9     //print "World" after execution of the 'hello()' function
10    std::cout << "World" << std::endl;
11  }
12
13 };
14
15 #endif
16
```

Figure: Creation of World.ah

AspectC++ Working

```
#ifndef __HELLO_H__
#define __HELLO_H__

#include <iostream>

void hello(){
  std::cout << "Hello" << std::endl;
}

#endif
~
~
```

Figure: Creation of hello.ah

AspectC++ Working

```
#include "hello.h"

int main(){
  hello(); //print "Hello"
  return 0;
}
~
~
~
~
~
~
~
~
```

Figure: Main Program

```
┌──(nishant㉿nishant)-[~/Downloads/aspectc++/examples/helloworld]
└─$ ls
Makefile  hello.h  main.cc  world.ah

┌──(nishant㉿nishant)-[~/Downloads/aspectc++/examples/helloworld]
└─$ make
Compiling main.cc
Linking helloworld
```

Figure: Compiling Hello World

Working in Threads

~/Downloads/aspectc++/examples/threads/SynchronizedLog.ah - Mousepad

File Edit Search View Document Help

```
1 #ifndef __SynchronizedLog_ah__
2 #define __SynchronizedLog_ah__
3
4 #include <pthread.h>
5
6 aspect SynchronizedLog {
7   // introduce a mutex into the Log class
8   // → there is one mutex instance per log object
9   advice "Log": slice struct {
10    pthread_mutex_t _mutex;
11  };
12
13  // initialize the mutex upon Log construction
14  advice construction("Log") : before () {
15    pthread_mutex_t helper = PTHREAD_MUTEX_INITIALIZER;
16    tjp→that ()→_mutex = helper;
17  }
18
19  // protect the critical section
20  advice execution("% Log::print( ... )") : around () {
21    pthread_mutex_lock(&tjp→that ()→_mutex);
22    tjp→proceed ();
23    pthread_mutex_unlock(&tjp→that ()→_mutex);
24  }
25 };
26
27 #endif // __SynchronizedLog_ah__
28
```

Figure: Creation of synchronized-log.ah

AspectC++ Working

```
--#ifndef __ThreadSafeLogging_ah__
#define __ThreadSafeLogging_ah__

#include <stdio.h>
#include <pthread.h>

#include "Log.h"

// This example illustrates how to use the aspectof() function
// to implement thread-local aspects with the pthread library
// or others that support thread-local storage in a similiar way.

// The beginning looks quite normal ...
aspect ThreadSafeLogging {

  int _execs;    // thread local function execution counter
  int _indent;   // per thread indentation of the log

  advice execution ("% TestClass::%( ... )") : before () {
    _execs++;
    Log::instance ().print (_indent, true, JoinPoint::signature ());
    _indent++;
  }

  advice execution ("% TestClass::%( ... )") : after () {
    _indent--;
    Log::instance ().print (_indent, false, JoinPoint::signature ());
  }

```

Figure: Creation of ThreadSafeLogging.ah

AspectC++ Working

```
#ifndef __Log_h__
#define __Log_h__

#include <stdio.h>

class Log {
public:
  void print (int indent, bool enter, const char *text) {
    for (int i = 0; i < indent; i++) printf (" ");
    if (enter) printf ("> "); else printf ("< ");
    printf ("%s\n", text);
  }

  static Log &instance () {
    static Log log;
    return log;
  }
};

#endif //  __Log_h__
~
```

Figure: Creation of log file

```
┌──(nishant㉿nishant)-[~/Downloads/aspectc++/examples/threads]
└─$ ls
Log.h  Makefile  SynchronizedLog.ah  ThreadSafeLogging.ah  main.cc

┌──(nishant㉿nishant)-[~/Downloads/aspectc++/examples/threads]
└─$ make
Compiling main.cc
main.cc:24:1: warning: non-void function does not return a value [-Wreturn-type]
}
^
main.cc: In function ‘void* thread_start(void*)’:
main.cc:24:1: warning: no return statement in function returning non-void [-Wreturn-type]
   24 | }
      | ^
Linking threads
```

Figure: Linking Threads

Concurrent

Independence of Execution:

- Concurrent programs consist of multiple independent tasks or processes that can execute simultaneously.
- Each task operates independently, and the order of execution is not predetermined.

Communication and Coordination:

- Concurrency involves communication and coordination between concurrent tasks.
- This is typically achieved through mechanisms such as message passing, shared memory, or synchronization primitives.

Parallelism:

- Parallelism is a key aspect of concurrency, where tasks are executed simultaneously to achieve improved performance.
- Concurrency provides a way to express parallelism in a program.

Handling Concurrent Access:

- Dealing with shared resources and avoiding race conditions are essential aspects of concurrent programming. Locks, semaphores, and other synchronization mechanisms are used to manage access to shared data.

Erlang

- **Lightweight Processes:**
  Erlang processes are lightweight and can be created and managed efficiently. This is in contrast to operating system threads, making it feasible to spawn thousands or even millions of Erlang processes.
- **Message Passing:**
  Concurrency in Erlang is based on message passing between processes. Processes communicate by sending and receiving messages. This explicit communication model simplifies concurrent programming and avoids shared state issues.
- **Immutability:**
  Erlang encourages immutability, meaning that once a data structure is created, it cannot be modified. Immutability helps prevent data races and simplifies reasoning about concurrent code.
- **Selective Receive:**
  Erlang supports selective receive, allowing a process to choose which messages to process based on their content. This enables more controlled and specific handling of messages.

Printing Hello Erlang

```
File  Actions  Edit  View  Help
 GNU nano 7.2
% my_file.erl
-module(my_file).
-export([hello/0]).

hello() ->
    io:format("Hello, Erlang!~n").
```

Figure: Code Snippet

```
┌──(nishant㉿nishant)-[~/Downloads/erlaang]
└─$ nano my_file.erl

┌──(nishant㉿nishant)-[~/Downloads/erlaang]
└─$ erlc my_file.erl
```

Figure: Compiling the code

Concurrent-Erlang Working

```
┌──(nishant㉿nishant)-[~/Downloads/erlaang]
└─$ erl
Erlang/OTP 25 [erts-13.2.2.5] [source] [64-bit] [smp:2:2] [ds:2:2:10] [async-threads:1] [jit]

Eshell V13.2.2.5  (abort with ^G)
1> my_file:hello().
Hello, Erlang!
ok
```

Figure: Output

Concurrent-Erlang Working

```
-module(conc).
-export([start/1, square/1, print_squares/2, main/0]).

start(N) ->
    Pids = [spawn(?MODULE, square, [X]) || X <- lists:seq(1, N)],
    print_squares(N, Pids).

square(X) ->
    Square = X * X,
    io:format("~w squared is ~w~n", [X, Square]).

print_squares(0, _) ->
    ok;
print_squares(N, [Pid | Rest]) ->
    Pid ! print,
    print_squares(N - 1, Rest).

main() ->
    N = 5,
    start(N).
```

Figure: Code Snippet for Concurrency

Concurrent-Erlang Working

```
┌──(nishant㉿nishant)-[~/Downloads/erlaang]
└─$ erl
Erlang/OTP 25 [erts-13.2.2.5] [source] [64-bit] [smp:2:2] [ds:2:2:10] [async-threads:1] [jit]

Eshell V13.2.2.5  (abort with ^G)
1> c(conc).
{ok,conc}
2> conc:main().
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
ok
3>
```

Figure: Output

Aspect-Oriented Programming (AOP) vs. Concurrent Programming Similarities:

- **Modularity:**
  Both AOP and Concurrent Programming aim to improve modularity. AOP achieves modularity by isolating cross-cutting concerns into aspects, while Concurrent Programming emphasizes modular design to manage independent processes or threads.
- **Cross-Cutting Concerns:**
  AOP is designed to address cross-cutting concerns such as logging, security, and error handling. Concurrent programming often involves handling shared resources, synchronization, and communication, which can be considered cross-cutting concerns.
- **Enhanced Abstraction:**
  Both paradigms aim to provide enhanced abstractions for dealing with complex scenarios. AOP introduces new constructs like aspects, join points, and advice, while concurrent programming introduces lightweight processes, message passing, and distribution.

Differences

- **Focus and Goal:**
  AOP focuses on modularizing and encapsulating cross-cutting concerns to improve code modularity and maintainability. Concurrent programming focuses on managing and executing tasks simultaneously to improve system performance and responsiveness.
- **Programming Constructs:**
  AOP introduces specific constructs like aspects, join points, and advice to separate concerns. Concurrent programming introduces concepts like lightweight processes, message passing, and synchronization primitives.
- **Language Support:**
  While AOP can be implemented in various languages (AspectJ in Java, AspectC++ in C++), it is not a standalone language. Concurrent programming, as seen in Erlang, is often deeply integrated into the language itself, providing specific features to handle concurrency.

Differences

- **Concurrency Model:**
  AOP does not inherently provide a concurrency model. It focuses on concerns that span multiple modules. Concurrent programming, as in Erlang, provides a clear and robust concurrency model based on the Actor model, lightweight processes, and message passing.
- **Use Cases:**
  AOP is often employed in scenarios where cross-cutting concerns can be cleanly separated, such as logging or security. Concurrent programming, particularly in Erlang, is employed in systems requiring high concurrency, fault tolerance, and distribution, such as telecommunication and distributed systems.
- **Adoption and Popularity:**
  Concurrent programming, especially in languages like Erlang, has gained popularity in specific domains. AOP, while powerful, might not be as widely adopted, and its usage depends on specific needs and preferences.

References

1. ChatGPT
2. https://stackoverflow.com/questions/49963072/how-to-correctly-use-concurrency-in-erlang
3. https://stackoverflow.com/questions/4200183/aspect-oriented-programming-in-c-current-supported-alternatives
4. https://www.aspectc.org/Documentation.php
5. https://www.erlang.org/doc/getting_started/conc_prog.html