CSE 1325: Object-Oriented Programming

Lecture 24

Concurrency and Hyperthreading
Deployment Diagrams
Anti-Patterns

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup
www.stroustrup.com/Programming

ERB 402
Office Hours:
Tuesday Thursday 11 - 12
Or by appointment

Lecture 23
Quick Review

- Briefly explain the following data types and the reason why they are often good for embedded systems.
  - Arrays (and not vectors) – Fixed size, no overhead, but offers methods
  - Stacks – Grow and shrink from top only, no fragmentation, constant time
  - Globals – Allocated at startup, fixed size
  - Pools – Allocate fixed size objects in any order, no fragmentation, constant time
- Bit manipulation is usually managed through the **bitset** class.
  - In addition to the usual size() method, identify and briefly define 4 other bit manipulation methods. set/reset/flip changes one or all bits, operator[]/test checks a bit, count() returns number of 1s
- Briefly explain the bit manipulation operators:
  & (AND), | (OR), ^ (Exclusive OR), ~ (1's complement),
  << and >> shift left and right with 0 fill

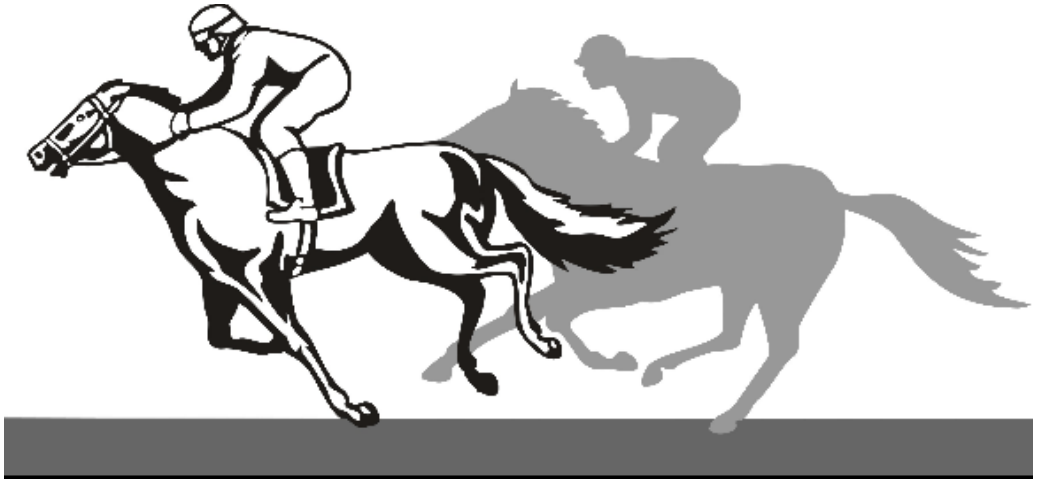
Lecture 23

Quick Review

- A **state** is the cumulative value of all relevant stored information to which a system or subsystem has access.
- A **state diagram** documents the states, permissible transitions, and activities of a system.
- An **event** is a type of occurrence that potentially affects the state of the system
- A **guard** is a Boolean expression that enables a state transition when true and disables it when false, e.g., [power == on]
- The **State Design Pattern** supports full encapsulation of unlimited states within a scalable context. It does not, however, support events as is. It is closely related to the **Strategy Pattern**.
- A **Mealy** state machine defines outputs based on current state and current inputs, while a **Moore** state machine defines outputs based on current state only.

Overview: Concurrency, Adapter Pattern, & Deployment Diagrams

- Introduction to Concurrency
  - Brief history
  - Uses
  - C++ Support
    - Thread class
    - Sleep
    - Conflicts / Race
    - Mutex



- Deployment Diagrams
- Anti-Patterns

Concurrency and Hyperthreading

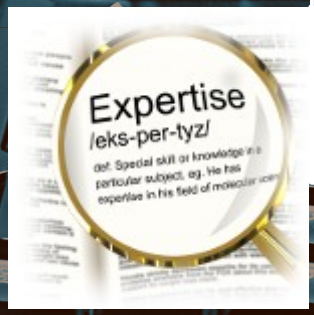
A Brief History of Concurrency

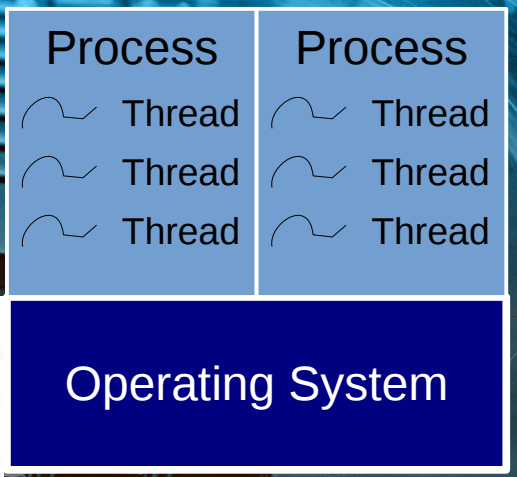
- **Moore's Law** (paraphrased): Computer tech (originally transistor density) doubles every 2 years
  - CPU speed, transistor and memory density, disk capacity, etc.
- By the 21st century, Moore's Law began to crack
  - Processor speeds topped out around 4 GHz (2.8 – 3.4 common)
  - Transistor density continued for some time – but how to best use?
- **Multi-Core Processor** – a chip with multiple **cores**, or ALU/register sets, each running a separate thread
  - Intel worked out use of one ALU with 2 register sets, interleaving 2 threads of execution – **Hyperthreading**
- Recently deployed 24 hyperthreaded core machines – but how to utilize so many cores?

Concurrency

Concurrency

- “I do one thing, I do it very well, and then I move on” – Dr. Charles Emmerson Winchester III
- “Move on, Chaaarles” – Hawkeye
- **Concurrency** – Performing 2 or more algorithms (as it were) simultaneously
- **Process** – A self-contained execution environment including its own memory space.
- **Thread** – An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space.





Conceptual Model

https://gcc.gnu.org/wiki/cauldron2015?action=AttachFile&do=get&target=Torvald+Riegel_+Modern+concurrent+code+in+C.pdf

Good Uses for Concurrency

- Perform background processing independent of the user interface, e.g., communicating the game state to apps
- Programming logically independent program units, e.g., the behavior of each non-player character in a game
- Processing small, independent units of a large problem, e.g., calculating the shaded hue of each pixel in a rendered photograph – or rendering an entire movie frame by frame!
- Periodic updating of a display or hardware unit, e.g., updating the second hand of a clock
- Periodic collection of data, e.g., capturing wind speed from an anemometer every 15 seconds

Creating a C++ Thread via Thread

- Class std::thread represents a thread of execution
  - Each thread has a unique thread ID (.get_id())
  - Each initialized thread can be “joined” back to the main thread

```
#include <string>
#include <iostream>
#include <thread>
using namespace std;

// The function we want to execute on the new thread
void task1(string msg) {
    cout << "task1 says: " << msg << endl;
}

int main() {
    // Constructs the new thread and runs it. Does not block execution.
    thread t1(task1, "Hello");
    cout << "Thread 1 ID is " << t1.get_id() << endl;

    // Makes the main thread wait for the new thread to finish execution
    t1.join();
}
```

Creating a C++ Thread via Thread

- Class std::thread represents a thread of execution
  - Each thread has a unique thread ID (.get_id())
  - Each initialized thread can be “joined” back to the main thread (a non-initialized thread can't)

```
#include <string>
#include <iostream>
#include <thread>
using namespace std;

// The function we w
void task1(string ms
    cout << "task1 s
}

int main() {
    // Constructs the new thread and runs it. Does not block execution.
    thread t1(task1, "Hello");
    cout << "Thread 1 ID is " << t1.get_id() << endl;

    // Makes the main thread wait for the new thread to finish execution
    t1.join();
}
```

```
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 threadid.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
terminate called after throwing an instance of 'std::system_error'
  what():  Enable multithreading to use std::thread: Operation not permitted
Aborted (core dumped)
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 -pthread threadid.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
Thread 1 ID is 7f6211307700
task1 says: Hello
ricegf@pluto:~/dev/cpp/201608/threads$
```

Must compile with -pthread

C++ Offers a *Hint* at HW Support

- thread:hardware_concurrency() returns a rough estimate of the number of *concurrent* threads
  - This may represent cores or hyperthreaded half-cores
  - This may return 0 or nothing relevant at all

```
#include <string>
#include <iostream>
#include <thread>

using namespace std;

int main()
{
    // Show rough approximation of thread capacity
    cout << "Hardware concurrency is " << thread::hardware_concurrency() << endl;
}
```

```
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 -pthread hw_concurrency.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
Hardware concurrency is 4
task1 says: Hello
ricegf@pluto:~/dev/cpp/201608/threads$
```

Threads can be Confusing

- Both main() and t1() execute independently
  - Threads can switch execution *between microprocessor instructions* (not C++ lines) at any time
  - This can garble output

Ungarbled output

```
ricegf@pluto:~/dev/cpp/threads$ g++ -std=c++0x -pthread threadid.cpp
ricegf@pluto:~/dev/cpp/threads$ ./a.out
Thread 1 ID is 140301458634496
task1 says: Hello
ricegf@pluto:~/dev/cpp/threads$
```

Garbled output

```
ricegf@pluto:~/dev/cpp/threads$ ./a.out
Thread 1 ID is 1404682637125 12task1 says:
Hello
ricegf@pluto:~/dev/cpp/threads$
```

  - We'll see how to avoid this in a bit

Sleeping a Thread

- It's tempting to pause a thread using a “busy loop”

```
for (int i = 0; i < 100000; ++i) { } // Wait a while
```

- This is *very* problematic
  - Compilers are very smart nowadays, and may optimize away the useless loop
  - Processor speeds vary widely, so timing is uncertain
  - If it runs the instructions, it's burning valuable CPU cycles that could be used by other threads
- Instead, use this_thread::sleep_for

```
this_thread::sleep_for(chrono::milliseconds(2000); // or seconds(2)
```

Sleeping 3 Threads Randomly

```
#include <string>
#include <iostream>
#include <thread>
#include <chrono>
#include <time.h>
using namespace std;

void task1(string msg) {
    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    cout << "task1 says: " << msg << endl;
}

int main() {
    // Randomize the pseudorandom number generator
    srand(time(NULL));

    // Construct new threads
    thread t1(task1, "Hello");
    thread t2(task1, "Bonjour");
    thread t3(task1, "Hola");

    // Join all threads back
    t1.join();
    t2.join();
    t3.join();
}
```

Sleeping 3 Threads Randomly

```
#include <string>
#include <iostream>
#include <thread>
#include <chrono>
#include <time.h>
using namespace std;

void task1(string msg) {
    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    cout << "task1 says: " << msg << endl;
}

int main() {
    // Randomize the pseudorandom number generator
    srand(time(NULL));

    // Construct new threads
    thread t1(task1, "Hello"
    thread t2(task1, "Bonjou
    thread t3(task1, "Hola")

    // Join all threads back
    t1.join();
    t2.join();
    t3.join();
}
```

```
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 -pthread 3thread.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
task1 says: Bonjour
task1 says: Hola
task1 says: Hello
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
task1 says: Hello
task1 says: Hola
task1 says: Bonjour
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
task1 says: Hola
task1 says: Hello
task1 says: Bonjour
ricegf@pluto:~/dev/cpp/201608/threads$
```

Tasks may start and run *in any order*

Applying Threads to Serious Work (Race Horses)

horse.h

```
#include <string>

using namespace std;

class Horse {
    public:
        Horse(string name, int speed)
      : _name{name}, _speed{speed}, _position{30} { }
        string name();
        int position();
        int speed();
        int move();
    protected:
        string _name;
        int _position;
        int _speed;
};
```

horse.cpp

```
#include "horse.h"

using namespace std;

string Horse::name() {return _name;}
int Horse::position() {return _position;}
int Horse::speed() {return _speed;}
int Horse::move() {if (_position > 0) --_position;}
```

Utility functions

horserace.cpp

```
#include <string>
#include <iostream>
#include <thread>
#include <chrono>
#include <time.h>
#include "horse.h"

using namespace std;

// Our three competitors, assigned random speeds (smaller is faster)
Horse horse1("Legs of Spaghetti", 100 + rand() % 100);
Horse horse2("Ride Like the Calm", 100 + rand() % 100);
Horse horse3("Duct-taped Lightning", 100 + rand() % 100);

// The gallop threads, which decrement position after a random delay
void gallop(Horse& horse) {
    while (horse.position() > 0) {
        this_thread::sleep_for(
            chrono::milliseconds(horse.speed() + rand() % 200));
        horse.move();
    }
}

// Utility function to print the horse track
void view(int position) {
    for (int i = 0; i < position; ++i) cout << (i%5 == 0 ? ':' : '.');
}                                          // Continued next page
```

main()

horserace.cpp

```
int main() {
    // Randomize the pseudorandom number generator
    srand(time(NULL));

    // Construct the new threads (horses) and run them
    thread t1{gallop, ref(horse1)};
    thread t2{gallop, ref(horse2)};  // ref( forces the parameter to pass as
    thread t3{gallop, ref(horse3)};  // a reference rather than a value

    // Display the horse track as the race runs
    while (horse1.position() > 0
       &&  horse2.position() > 0
       &&  horse3.position() > 0) {
        cout << endl << endl << endl << endl;
        view(horse1.position());
        cout << " " << horse1.name() << endl;
        view(horse2.position());
        cout << " " << horse2.name() << endl;
        view(horse3.position());
        cout << " " << horse3.name() << endl;
        this_thread::sleep_for(chrono::milliseconds(100));
    }

    // Join the threads
    t1.join();
    t2.join();
    t3.join();
}
```

The Obligatory Makefile

Makefile

```
CXXFLAGS += -std=c++11 -pthread

all: main

debug: CXXFLAGS += -g
debug: main

rebuild: clean main

main: horserace.o horse.o
        g++ -o horserace $(CXXFLAGS) horserace.o horse.o
horserace.o: horserace.cpp horse.h
        g++ $(CXXFLAGS) -c horserace.cpp
horse.o: horse.cpp horse.h
        g++ $(CXXFLAGS) -c horse.cpp
clean:
        -rm -f *.o *~ horserace
```

```
:.. Ride Like the Calm
:.... Duct-taped Lightning


:....: Legs of Spaghetti
:.. Ride Like the Calm
:.... Duct-taped Lightning


:....: Legs of Spaghetti
:.. Ride Like the Calm
:... Duct-taped Lightning


:.... Legs of Spaghetti
:.. Ride Like the Calm
:... Duct-taped Lightning


:.... Legs of Spaghetti
:. Ride Like the Calm
:... Duct-taped Lightning


:.... Legs of Spaghetti
:. Ride Like the Calm
:.. Duct-taped Lightning


:.... Legs of Spaghetti
: Ride Like the Calm
:.. Duct-taped Lightning
```

Running the Race

(It's a lot easier to follow live!)

Concurrency is Harder than Single-Threaded

- Non-reentrant code can lose data
  - **Reentrant** – An algorithm can be paused while executing, and then safely executed by a different thread
  - Non-reentrant code can experience **Thread Interference**
- Methods that aren't **thread safe** enable Threads to corrupt objects
  - Thread A updates a portion of the object's data, while Thread B updates a dependent portion, leaving the object in an inconsistent state – the bug's impact occurs much later!
- Memory Consistency Errors
  - Because variables may be cached, one thread's change may never be incorporated by a different thread's algorithm
- Concurrent bugs tend to appear only when multiple threads happen to align, thus appearing to be both rare and random
  - Nightmare debugging scenario

The "Garbled Output" Problem Revisited

- Here's a (Java) example of a thread sync problem

(`javap -c Counter.class` disassembles bytecode)

```
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public void decrement() {
        count--;
    }

    public int getCount() {
        return count;
    }
}
```

```
public void increment();
  Code:
     0: aload_0
     1: dup
     2: getfield      #2
          // Field count:I
     5: iconst_1
     6: iadd
     7: putfield      #2
          // Field count:I
    10: return
```

Thread A

Paused here!

```
public void decrement();
  Code:
     0: aload_0
     1: dup
     2: getfield      #2
          // Field count:I
     5: iconst_1
     6: isub
     7: putfield      #2
          // Field count:I
    10: return
```

Thread B runs decrement

Thread A resumes

The change made by decrement in Thread B is overwritten by the obsolete data in Thread A. $+1-1 \stackrel{?}{=} 1$

Hint: Immutable Classes!

- If the data can't change, synchronization issues can't crop up *in that class*
  - Create and use immutable classes when practical
  - In an immutable class, mark all data fields as const to indicate those fields won't change
  - This isn't always possible...
  -

General C++ Solution: The Mutex

- Assume a crowd of people want to use an old-fashioned phone booth. The first to grab the door handle gets to use it first, but must hang on to the handle to keep the others out. When finished, the person exits the booth and releases the door handle. The next person to grab the door handle gets to use the phone booth next.

- A **thread** is: Each person
  The **mutex** is: The door handle
  The **lock** is: The person's hand
  The **resource** is: The phone

Hat tip to Nav on Stack Overflow for this analogy

The Mutex in Code

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std

mutex m;
int i = 0;

void makeACallFromPhoneBooth() {
    m.lock(); // person grabs the phone booth door and locks it, all others wait
    cout << i << " talks on phone" << endl;
    i++; //no other thread can access variable i until m.unlock() is called
    m.unlock(); // person releases the door handle and unlocks the door
}

int main() {
    thread person1(makeACallFromPhoneBooth); // Despite appearances, as we saw
    thread person2(makeACallFromPhoneBooth); // earlier, it is not clear who will
    thread person3(makeACallFromPhoneBooth); // reach the phone booth first!

    person1.join(); // person1 finished their phone call and joins the crowd
    person2.join(); // person2 finished their phone call and joins the crowd
    person3.join(); // person3 finished their phone call and joins the crowd
}
```

```
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 -pthread phone.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
0 talks on phone
1 talks on phone
2 talks on phone
ricegf@pluto:~/dev/cpp/201608/threads$
```

Testing the Mutex

```
#include <iostream>
#include <thread>
using namespace std;

static const int num_threads = 50;
static const int num_decrements = 5000;

int counter = num_threads * num_decrements;

// This is the code to be run as threads
void decrementer() {
    for (int i=num_decrements; i > 0; --i) {
        --counter;
    }
}
int main() {
    //Launch a group of threads
    thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) t[i] = thread(decrementer);

    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) t[i].join();

    cout << "This should be 0: " << counter << endl;
    return counter;
}
```

Allow 5000 chances for thread interference per thread – and we see 10s of thousands! (Your machine may vary.)

```
ricegf@pluto:~/dev/cpp/201708/21$ ./no_mutex
This should be 0: 18067
ricegf@pluto:~/dev/cpp/201708/21$ ./no_mutex
This should be 0: 24515
ricegf@pluto:~/dev/cpp/201708/21$ ./no_mutex
This should be 0: 135590
ricegf@pluto:~/dev/cpp/201708/21$ ./no_mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$
```

Testing the Mutex

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

static const int num_threads = 50;
static const int num_decrements = 5000;

int counter = num_threads * num_decrements;

mutex m;

// This is the code to be run as threads
void decrementer() {
    for (int i=num_decrements; i > 0; --i) {
        m.lock(); --counter; m.unlock();
    }
}
int main() {
    //Launch a group of threads
    thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) t[i] = thread(decrementer);

    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) t[i].join();

    cout << "This should be 0: " << counter << endl;
    return counter;
}
```

The mutex solves the problem.

```
ricegf@pluto:~/dev/cpp/201708/21$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$
```

Warning

- This just scratches the surface of concurrency
  - Avoiding race conditions is exceptionally tricky
  - Other dangers lurk, e.g., priority inversion
  - Bugs in concurrent systems are often catastrophic, but appear only once in a blue moon
- This lecture gives you just enough knowledge to get in trouble... or to motivate you to learn much more about a field growing in importance
  - Your decision...

Applying Patterns

- Our Digital Signal Processing (DSP) application is designed to receive and process data from a QC-107 quadcopter drone.
- However, we could only afford the QC-53½ quadcopter drone, which has a different interface library not supported by our DSP application.
- This problem is a good candidate for the _____ pattern.

Applying Patterns

- Our company is expanding to 10 different sites around the world, each with its own local computing resources.
- When an employee seeks to launch an instance of a tool, we want to ensure that the instance is on computing resources local to that employee.
- This problem is a good candidate for the _____ pattern.

Applying Patterns

- Your team is developing a next-generation big data analysis tool that will revolutionize your industry, crush your competition, and possibly result in a pay raise.
- However, the tool's algorithms are dependent on the Goggles data reduction library, which is complex and difficult to learn and use. Only one team member has experience with this library.

- The tool needs only a small subset of the Goggles library functionality.
- This problem is a good candidate for the _____ pattern.

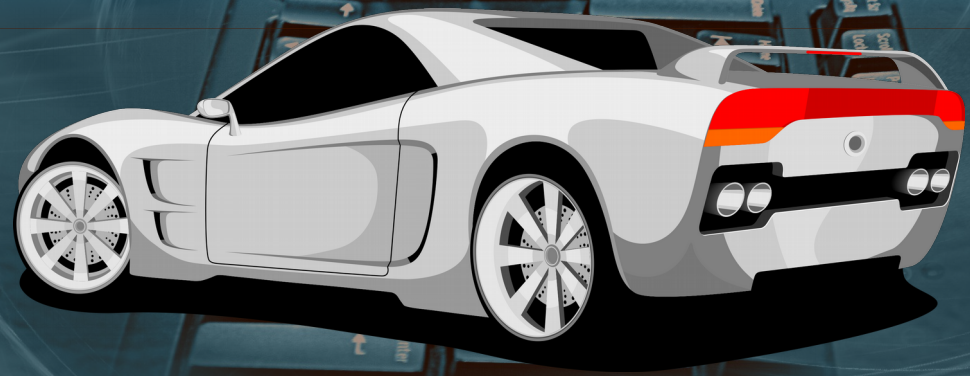
Applying Patterns

- After 42 sprints, your team is ready to deploy your new iPhone app to your eager worldwide fan base.

- To ensure your fan base doesn't become your plaintiffs, you want to collect data about app crashes and operational problems so you can proactively address any issues. One approach is for your data collection server to register to be notified when an installed app detects any anomalies.
- This problem is a good candidate for the _____ pattern.

Applying Patterns

- Your new self-driving car software will dominate the coming auto-Uber market by steering each wheel *individually* for optimal maneuvering.
- A central controller program determines the optimal angle of each wheel. However, it's rather important that only one central controller coordinate all 4 wheels, otherwise Bad Things might happen.
- This problem is a good candidate for the _____ pattern.



Applying Patterns

- Your Internet of Things (IoT) application will build an ad hoc network out of the outlets in a building. Your product backlog specifies that your application must try to establish the network via wifi, bluetooth, power line net, infrared, and audio comm technologies, and then use the one that works best in that specific building.
- This problem is a good candidate for the _____ pattern.

Applying Patterns

- Which pattern(s) weren't covered?
- Give an example scenario where each would be appropriate to resolve design issues.

Review

Patterns

A **software design pattern** is a general reusable algorithm solving a common problem

- Patterns are **discovered**, not invented
- Patterns are documented and validated by the software development community at large - **meritocracy**

Patterns: What To Do

Types of Patterns

- Creational Patterns
  - These patterns address creating objects from classes via mechanisms more flexible than “new”
- Structural Patterns
  - These patterns address class and object composition, which is in general favored over inheritance
- Architectural Patterns
  - These patterns address design of loosely coupled subsystems of objects
- Behavioral Patterns
  - These patterns address communication between objects
- We covered 9 of the 23 original GoF patterns
  - It's good to know others, but only these 9 will be on the exam!

Patterns We Covered

- Creational
  - Singleton – restricts a class to a single instance
  - Factory – creates new objects based on specified criteria
- Structural
  - Decorator – dynamically adds new functionality to a class
  - Adapter – implements a bridge between 2 classes
  - Façade – provides a simple interface to a complex class or package
- Architectural
  - MVC – separates the business logic from the user interface
- Behavioral
  - Observer – notifies dependent objects of events of interest
  - Strategy – enable algorithm behavior to be modified at runtime
  - State Design – supports full scalable encapsulation of unlimited states

Anti-Patterns

- Anti-Patterns are common errors reflected in software systems
  - Common enough to be given a name
  - One or more strategies are known to exist
- This is a small subset of the most common / irksome (IMHO) anti-patterns
  - Wikipedia actually has an extensive list
    - https://en.wikipedia.org/wiki/Anti-pattern
  - However, only these will be on the exam!

Anti-Patterns: What Not To Do

Anti-Pattern: Comment Inversion and Documentation Inversion

- **Comment Inversion**: Adding obvious code comments (“adds two variables”) and omitting useful comments such as why those variables should be added in the first place!
- **Documentation Inversion**: Explaining the system in terms of itself (“select 'New Foo' to create a new Foo”) rather than documenting what a Foo is and why the user may want one
- In both cases, empathy is power. Consider the reader and what they are likely to want to know, rather than just writing down what you happen to know
  - User-testing documentation is helpful to learn empathy

Anti-Pattern: Error Hiding

- Catching an error and either doing nothing with it or displaying a meaningless message
  - May also refer to destroying the evidence, e.g., overwriting the stack trace or disposing of problematic input
  - May also refer to addressing the error inappropriately, e.g., printing an error message in a library routine
- Catch exceptions only for specific reasons for which reasonable remediation is well-understood

Anti-Pattern: The “God Class”

- Functionality migrates upward in the class hierarchy until most methods are at the root
- This is a symptom that inheritance *may be* less appropriate for your design than composition
  - Use UML to understand your current class hierarchy
  - Test multiple redesigns with UML to more properly partition and encapsulate your data and allocate responsibilities to managable units of code.

AntiPattern: The Big Ball of Mud

- A software system lacking any discernible architecture, and is thus a “haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle” (Wikipedia).
  - Often result of a project with high programmer turnover and a long life of unstable requirements, bad managers, and tight budgets
  - Sometimes called “legacy code”
- *Usually* avoid temptation to rewrite – instead, try to *refactor*
  - Start by writing good test code that captures correct behavior
  - Next, define a clear and appropriate architecture for the system
  - Replace portions of the system one-by-one with well-designed code, using your test code to verify equivalent functionality
  - Continue until the Big Ball of Mud becomes maintainable enough

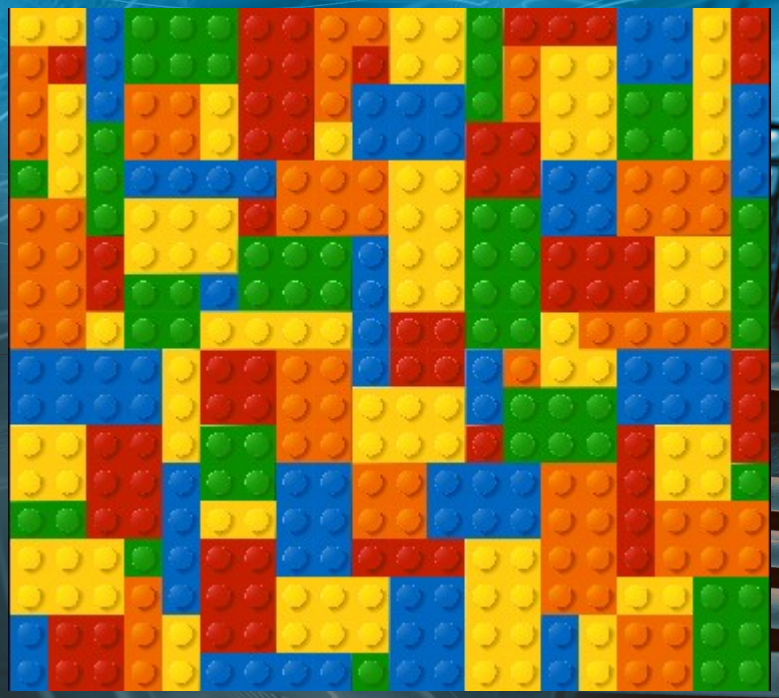
AntiPattern: Not Invented Here (NIH) Syndrome

- NIH is reimplementing an existing solution, e.g., a standard class library, due to the mistaken belief that code “not invented here” is inferior
  - If the longevity of the existing solution is in question, delegate to the Buyer to escrow or otherwise ensure access
  - If licensing concerns exist, discuss with Legal
  - If the code doesn't fit the problem precisely, engineer an adapter or façade

AntiPattern: Cargo Cult Programming

- Inclusion of code, language features, data structures, or (ahem) patterns without really understanding why
  - Often occurs when an inexperienced programmer copies code without understanding it
  - Lack of understanding results in redundant code, subtle errors, and unnecessary or misleading comments
- Complete due diligence and thoroughly understand EVERYTHING you include in your designs, code, and documentation
  - Ask questions! That what Sr. Programmers are for!

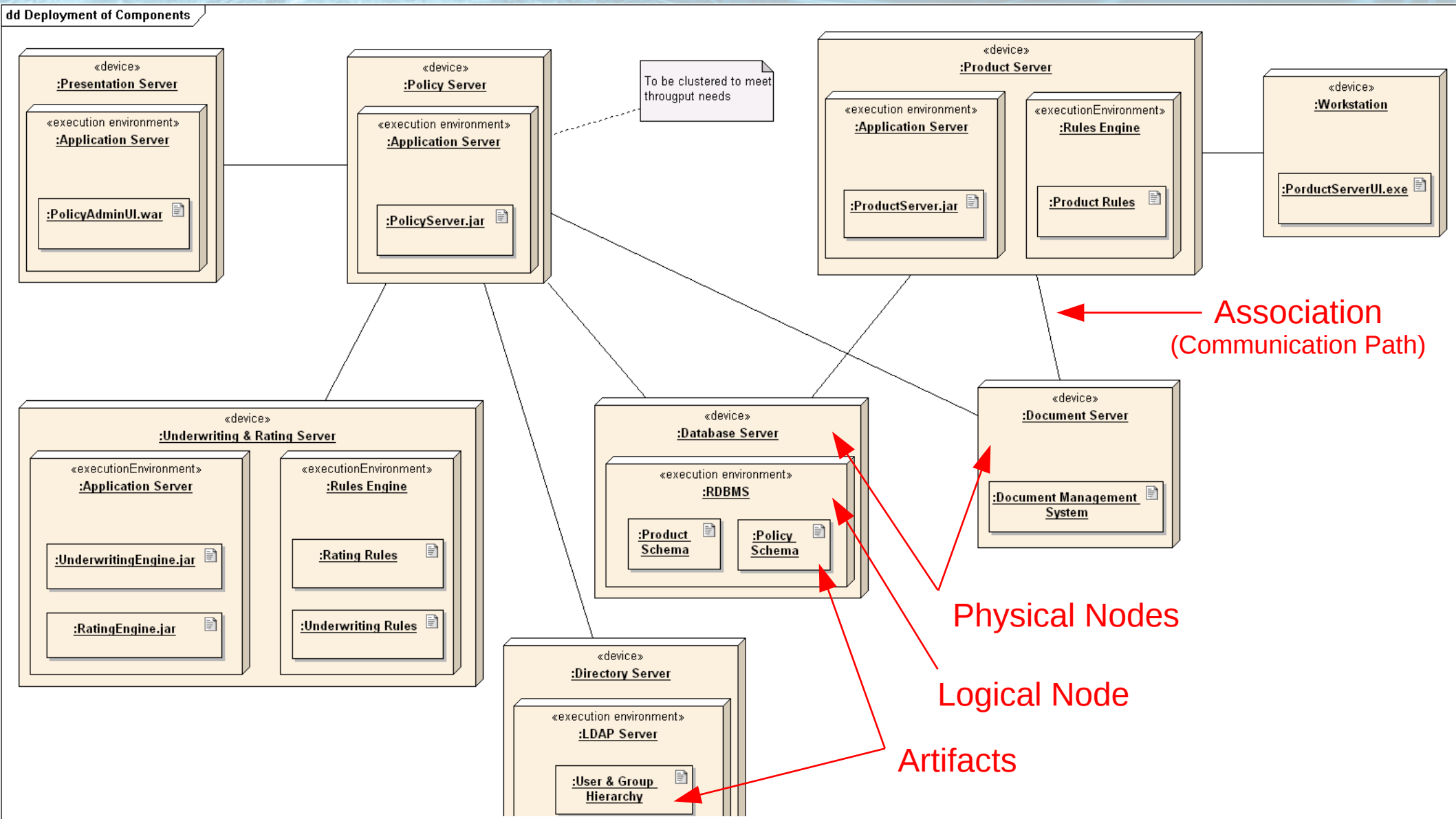
Deployment Diagrams



UML Deployment Diagrams

- A Deployment Diagram models the physical deployment of **artifacts** on **nodes**.
  - An **artifact** is the result of the software development process, e.g., an executable file, media file, script, database table, office document, or archive file.
    - An artifact may be defined recursively, e.g., an archive file that contains executables, resources, and installer.
  - A **node** is a physical or logical computational resource
    - A physical node (**device**) is a computer, and may be recursively defined (e.g., x64 processors consisting of multiple cores each)
    - A logical node (**execution environment**) is software that supports the asset, e.g., an operating system, virtual machine, or cloud server cluster

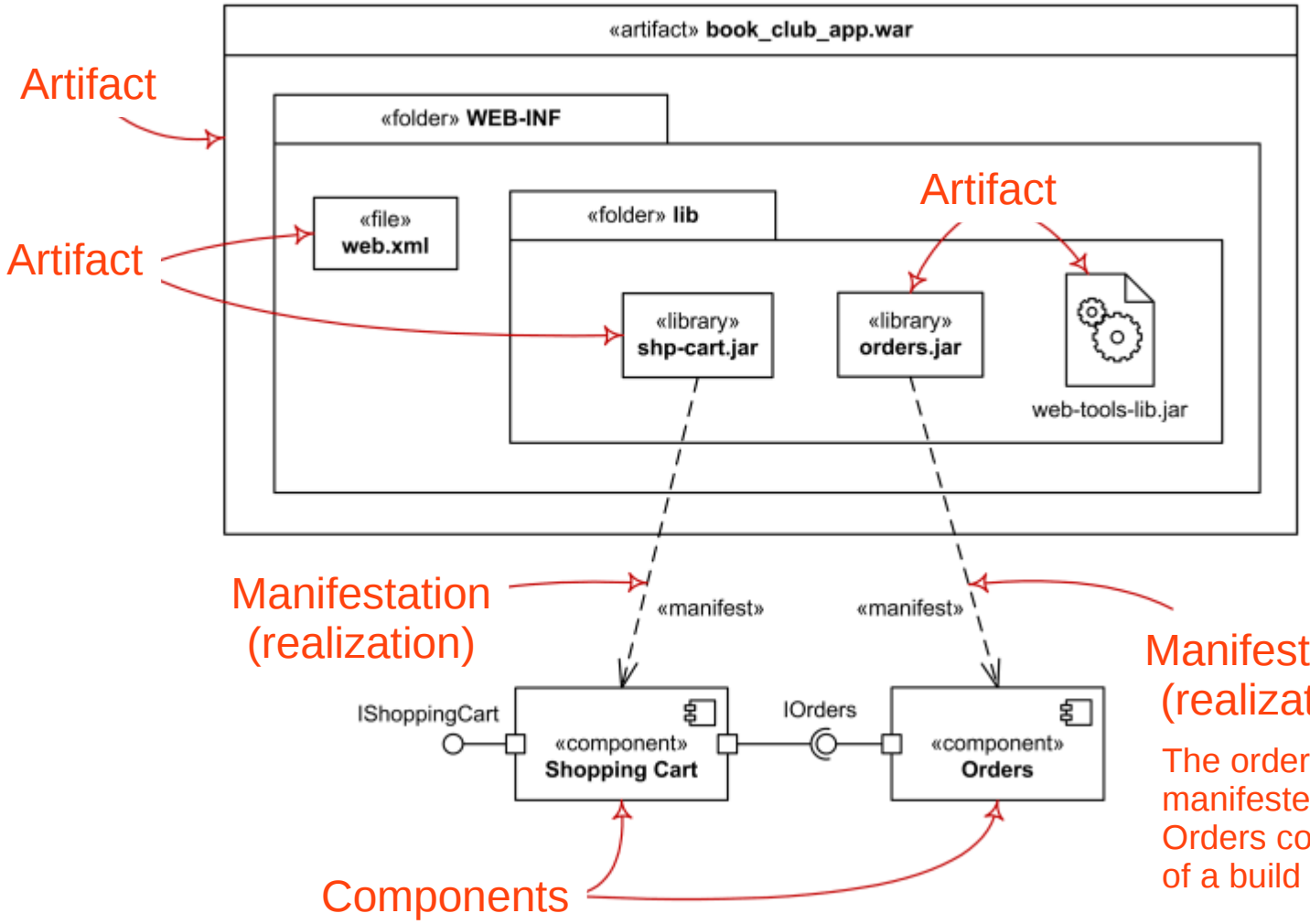
Example Deployment Diagram



4 Types of Deployment Diagrams

- Implementation of Components by Artifacts
  - Shows the manifestation (realization) of components (the result of the build) into artifacts (the configuration of components that can be deployed)
  - May also model the nested folder structure inside an artifact
- Specification-Level Deployment
  - Overview of deployment by artifact and node *types* (i.e., classes), without reference to specific instances
  - Often used to describe workstation or virtual server configurations
- Instance-Level Deployment
  - Shows deployment of specific *instances* (i.e., objects) of artifacts, e.g., by file name, to specific nodes, e.g., by server name
- Network Architecture
  - Represents connection of nodes via communication paths
  - Often the nodes are stereotyped and represented by more meaningful icons than the UML standard “3D cube” icon

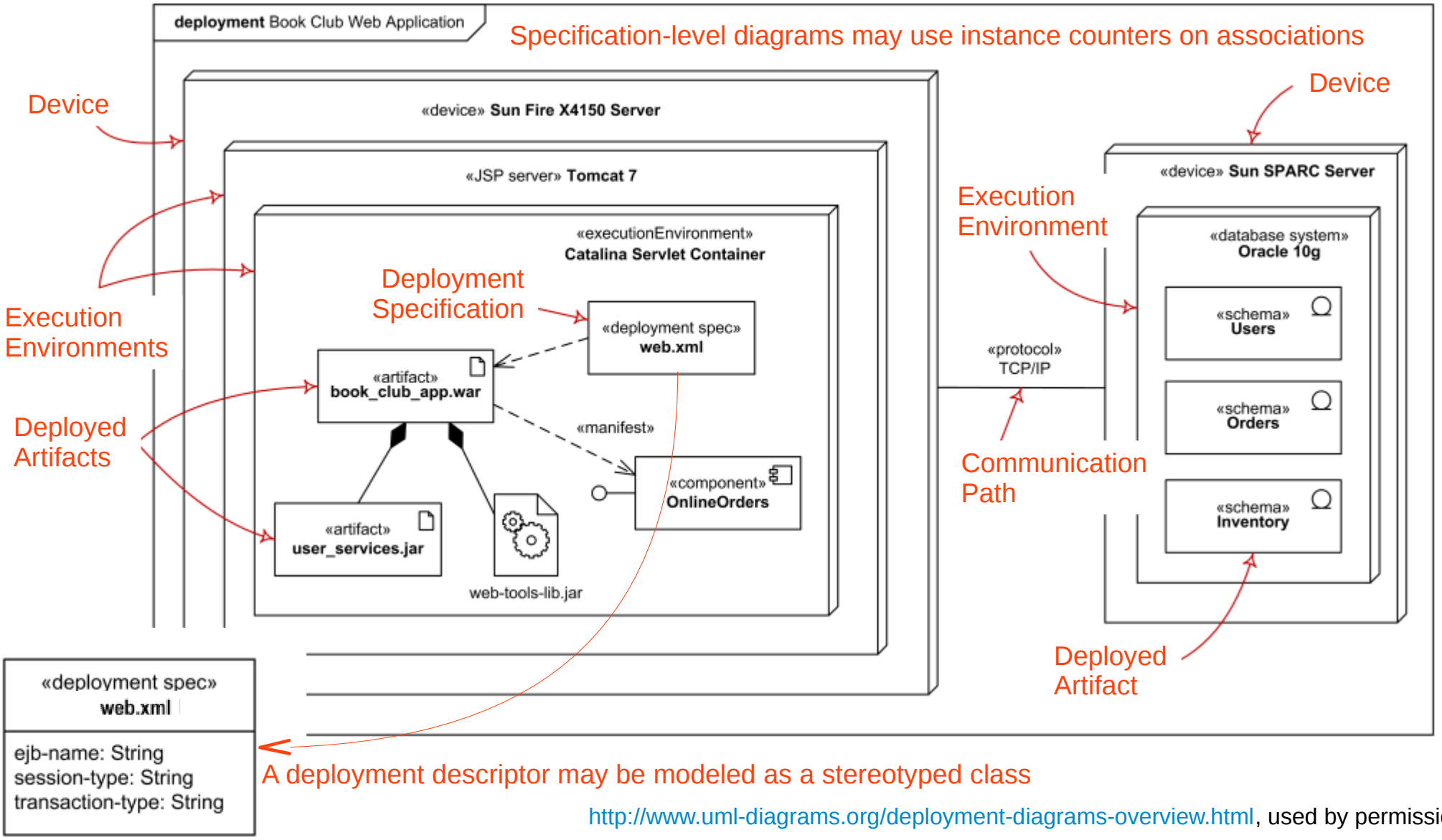
Manifestation of Components by Artifacts



The orders.jar artifact is manifested (built) from the Orders component (the result of a build process)

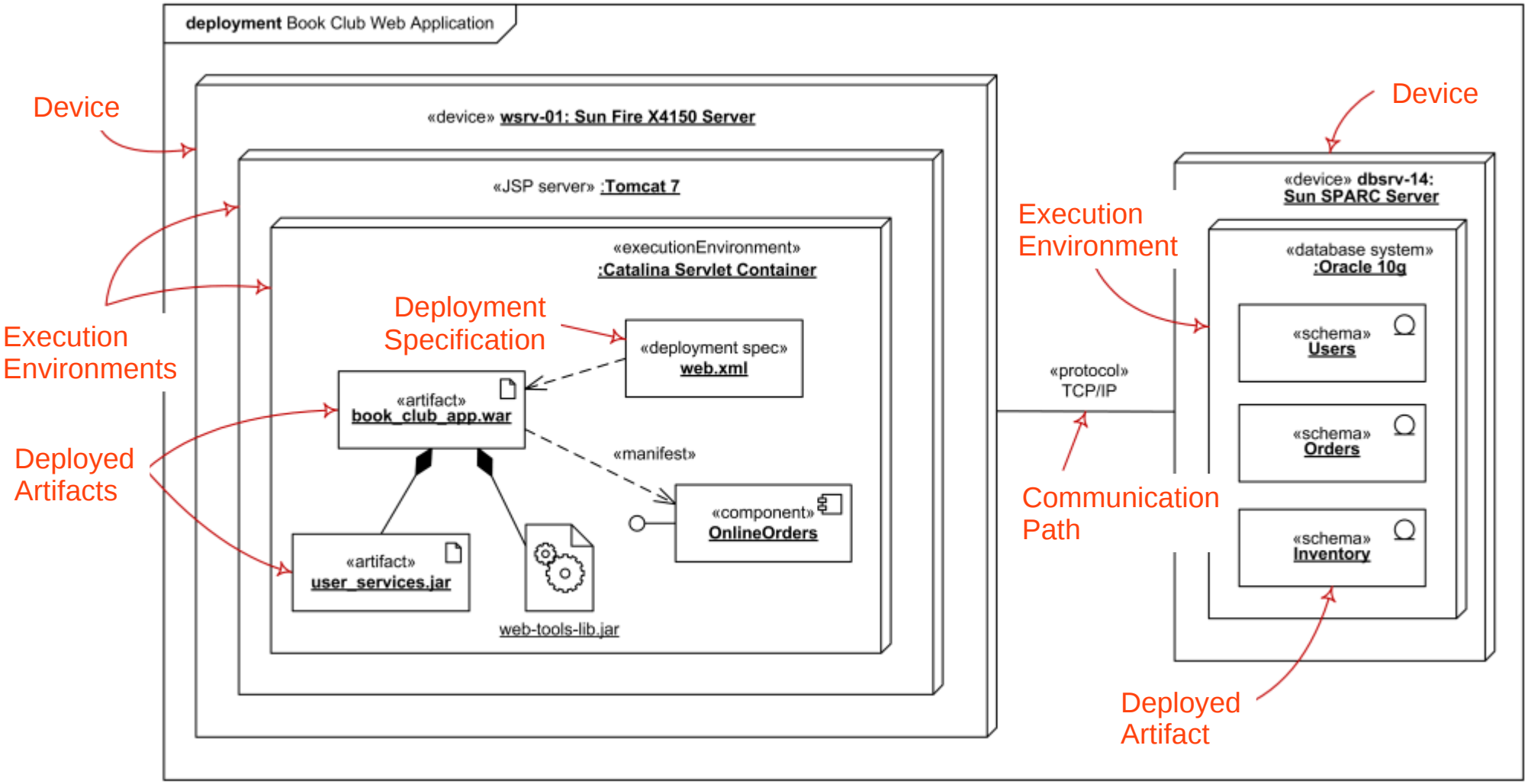
http://www.uml-diagrams.org/deployment-diagrams-overview.html, used by permission

Specification-Level Deployment



http://www.uml-diagrams.org/deployment-diagrams-overview.html, used by permission

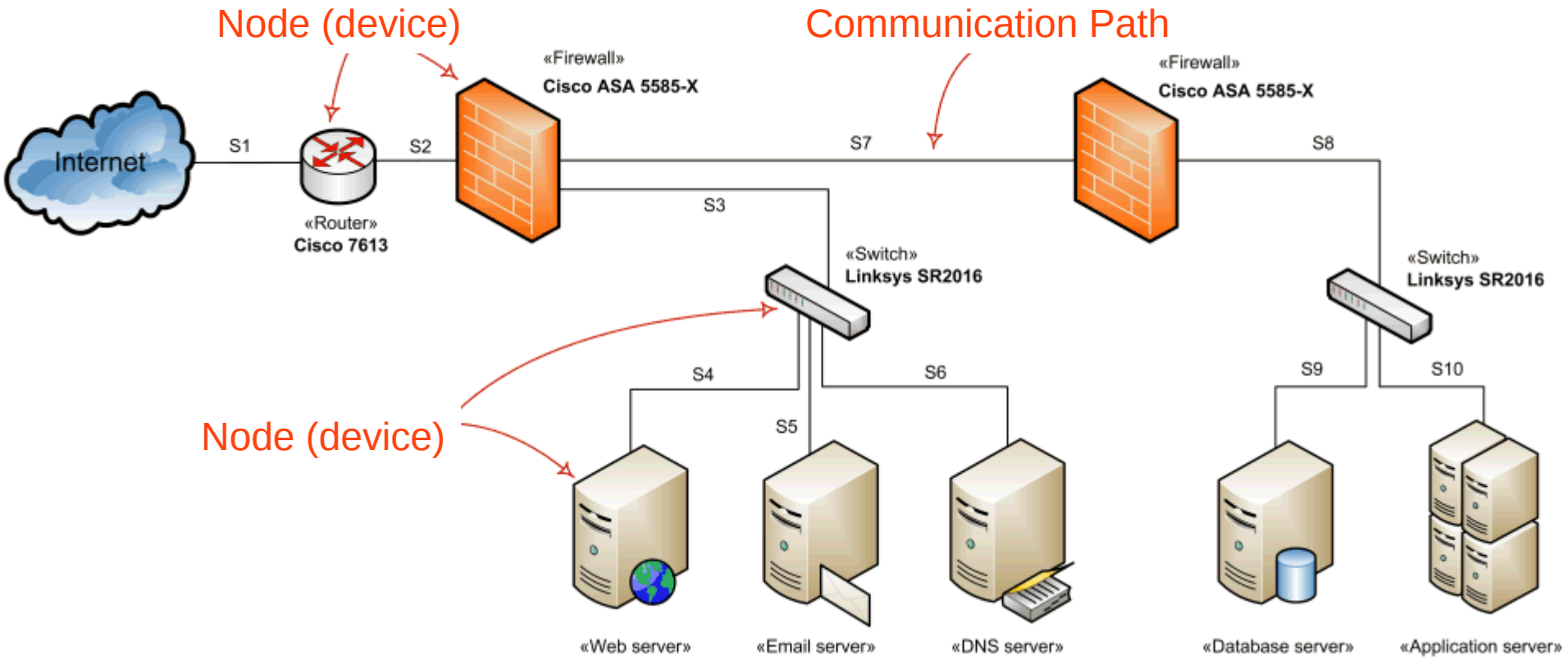
Instance-Level Deployment



Note the ":" and specific server names indicating instances rather than classes

http://www.uml-diagrams.org/deployment-diagrams-overview.html, used by permission

Network Architecture



Note the custom iconography for the nodes,
providing a richer visual representation of the network

http://www.uml-diagrams.org/network-architecture-diagrams.html, used by permission

Quick Review

- _____ is one ALU/register set on a CPU, which runs one thread at a time. _____ is when one ALU has 2 register sets, interleaving 2 threads.
- Match the definition to either process, thread, or concurrency.
  - Performing 2 or more algorithms (as it were) simultaneously
  - A self-contained execution environment including its own memory space
  - An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space.
- List 3 problems with busy loops. What's the better alternative?
- What does joining a thread accomplish?
- Describe the “garbled output” (sync) problem. How do invariant classes help? How does a mutex solve the problem for the general case?

Quick Review

- The _____ _____ creates new objects without exposing the creation logic to the client. List some capabilities and advantages of this pattern and situations where it may apply.
- A _____ _____ models the physical deployment of artifacts on nodes.
  - An _____ is the result of the software development process
  - A ____ is a physical or logical computational resource
  - A physical node (device) is typically a _____. Give an example.
  - A logical node (execution environment) is _____ that supports the asset. Give an example.
- Which of the follow are deployment diagram types?
  (A) Network Architecture (B) Military (C) .cpp and .h
  (D) Manifestation of Artifacts by Components (E) EXE
  (F) Specification-Level Deployment (G) Instance-Level Deployment
- The _____ _____ dynamically adds new functionality to an object without altering its structure

Quick Review

- Match Creational Pattern, Structural Pattern, and Behavioral Pattern to their definitions
  - These patterns address class and object composition, which is in general favored over inheritance
  - These patterns address communication between objects
  - These patterns address creating objects from classes via mechanisms more flexible than “new”
- Match the pattern name to the application and identify its type

Observer	1	Creates new objects without exposing the creation logic to the client
Decorator	2	Enables an algorithm behavior to be modified at runtime
Strategy	3	Implements a bridge between two classes with incompatible interfaces
Singleton	4	Restricts a class to instancing a single object
Adapter	5	Dynamically adds new functionality to an object without altering its structure
Factory	6	Implements a simplified interface to a complex class or package
MVC	7	Notifies dependent objects when the observed object is modified
State Design	8	Separates business logic from data visualization and human or machine user
Façade	9	Supports full scalable encapsulation of unlimited states

Quick Review

- Match the anti-pattern name to its definition

Documentation Inversion	1 Adding obvious code comments while omitting useful comments
The “God Class”	2 Explaining the system in terms of itself
Comment Inversion	3 Catching an error and ignoring it or displaying a meaningless message
Not Invented Here (NIH) Syndrome	4 Migrating functionality upward in the class hierarchy until most methods are at root
Error Hiding	5 A software system lacking any discernable architecture
The Big Ball of Mud	6 Reimplementing an existing solution because “our code is always better”
Cargo Cult Programming	7 Inclusion of code, language features, data structures or patterns with understanding why

Quick Review

- Suggest a pattern to address each of the following concerns
  - “I want to create an object based on subjective criteria”
  - “I want to switch to a more precise algorithm as my self-driving car approaches another vehicle”
  - “I want my software to be notified every time the user clicks the left mouse button”
  - “I want to extend a set of Boolean methods so that each one, when called, keeps re-running until it returns true”
- Define and suggest strategies to overcome each of the following anti-patterns
  - The “God Class”
  - Not Invented Here (NIH) Syndrome
  - The Big Ball of Mud
  - Comment and Documentation Inversion
  - Error Hiding
  - Cargo Cult Programming

For Next Week

- No reading – we finished the book!
- **Sprint #5 now in progress: Prep for delivery!**
- **Tuesday**: Final exam review
  - Study sheet is available on Blackboard
  - Practice exam is available or will be shortly
- **Thursday**: Project finals and contract award!
  - Come prepared – finalists announced in class
  - Presentations are not optional, but may be as simple as running your program through a few use cases
  - Class will vote for contract winner among finalists

Unsolicited Career Advice

(1) Integrity matters most. Never lie, mislead, or claim undue credit for any reason.

(2) Make your team and your boss look good. Credit others for success, but accept responsibility for your own mistakes quickly. **Focus on the solution**, not the blame. **Be a teacher** and help others grow and succeed, then cheer their success.

(3) Network broadly. Remember peoples' names. Cherish and practice loyalty.

(4) Being the person who knows who knows the answer is often more valuable than actually knowing the answer.

(5) Prefer "Yes". Consider no task beneath you, and do what is necessary for the team to win. But say "no" when necessary, e.g., to optimize your work load.

(6) Code early and often. A software manager or architect who doesn't code is clueless.

(7) Keep an engineering log. Write down everything, especially solutions and reasons.

(8) Learn new languages and tools often. **Automate everything.**

(9) Start stuff. Show initiative. Motivate your team, even if you're the junior member. Make good things happen. It's contagious.

(10) Keep a personal task backlog. Keep it updated. Work it by priority.