CMPSCI 220 Programming Methodology

Libraries

Based on Head First Design Patterns

Objectives

Libraries

- Learn what is a software library
- Learn about how different languages support libraries
- Learn how to always program for libraries
- Learn to focus on small reusable component libraries

What is a Software Library?

Well defined, controlled, and restricted **access** to a software artifact and its implementation.

Provides services to other software libraries or tools without any assumptions about its use.

Designed and implemented in isolation.

Isolation and Reusability

Libraries facilitate the construction of isolated reusable components that can be easily contained and distributed to software systems and other libraries that use them.

Libraries should be separately maintained and evolve with strict adherence to how they are accessed from other libraries.

Libraries should not impact other libraries that use them.

Testable and Verifiable

Libraries are easily testable and can be verified independent of the software system that depend on them.

This allows libraries to be more easily reasoned about and easier to verify and debug.

Organized and Orthogonal

Libraries are organized and define a clear and strict *interface* that is orthogonal to other libraries and tools that use them.

Leaking internal operations and data structures that depend on library implementation breaks orthogonally and causes dependencies on library internals.

UNIX Philosophy

One of the fundamental concepts of UNIX is creating new tools from smaller, dedicated tools. Rather than large monolithic tools that attempt to solve every problem, UNIX provides a board range of simple tools that are combined though pipes.

```
$ grep –v nobody /etc/password | awk –F: ‘{print $3}’ | sort –n | tail -1
```

This is one of the great reasons why UNIX is flexible!

Closed for Modification / Open for Extension

A good library will be closed for modification.
The internal representation does not need to be modified to extend its functionality.

A good library will be open for extension.
The library can be used through a well defined interface allowing it to be used and extended into other domains.

Why are Libraries Important?

There are a variety of reasons why libraries are important in building large-scale software systems.

Access to Software Libraries

Language Dependent

How libraries are implemented and accessed is dependent on the programming language.

Programming Language Support

Programming languages support different mechanisms for implementing libraries and different levels of granularity for controlling access to them.

Levels of Granularity

Course Grained Access

Some languages provide course grained access to libraries – operations and data are either accessible or they are not.

Fine Grained Access

Other languages provide fine grained access to libraries – operations and data may be accessible to a group of libraries, classes, operations, or data but not others.

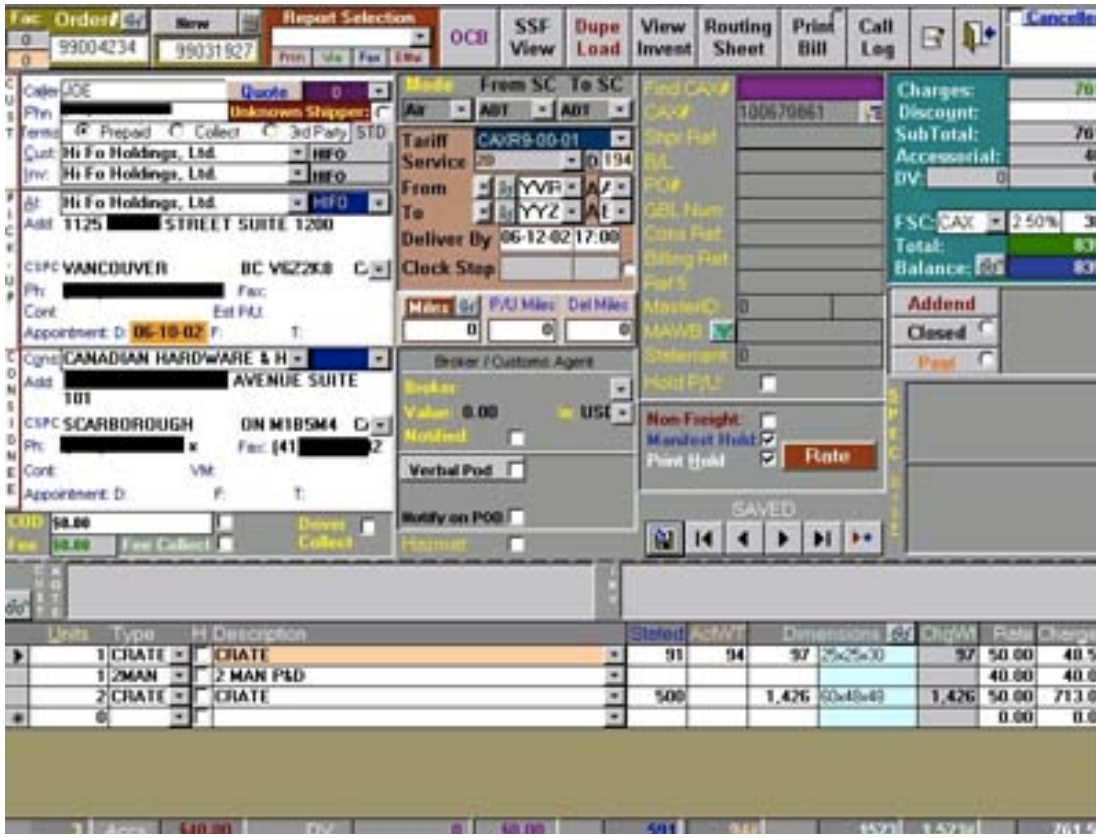
Well Defined Interfaces

Libraries (like user interfaces) provide access to functionality through well defined interfaces.

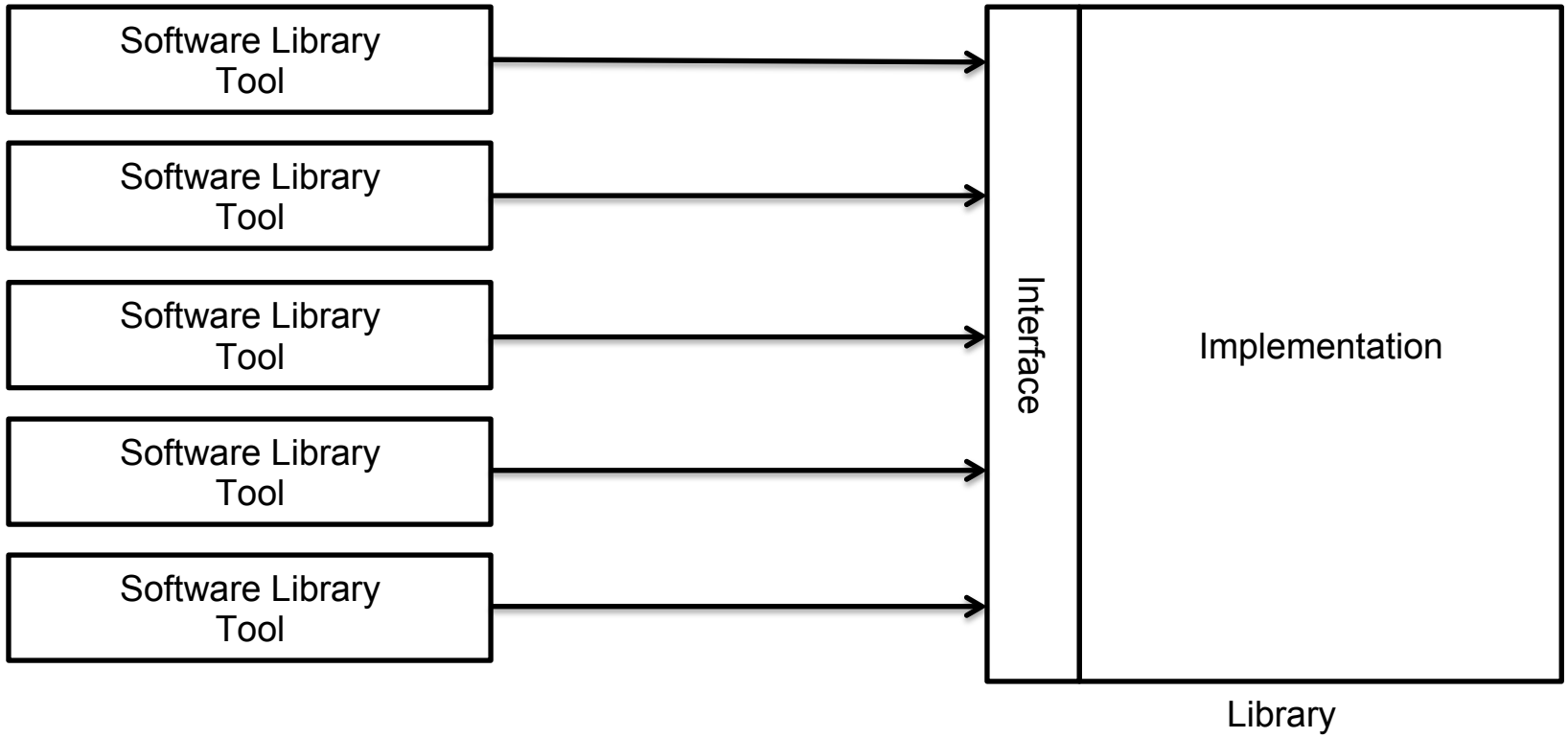
Well Defined Interfaces

Libraries (like user interfaces) provide access to functionality through well defined interfaces.

A poorly designed interface (confusing, to restrictive, overly permissive) will lead to libraries that are not useful, difficult to use and impact the performance and reliability of the software system using it.



Access Through Interface



Library Dependencies: Lasagna

Paying attention to what your library depends on and how it interacts with other libraries is important.

Lasagna is made from bottom to top where each layer depends on the layer below it. It takes a little longer to put together, but it is nicely organized and provides a rich flavor.

This is a good analogy for how libraries are organized, implemented, and used in a software system.

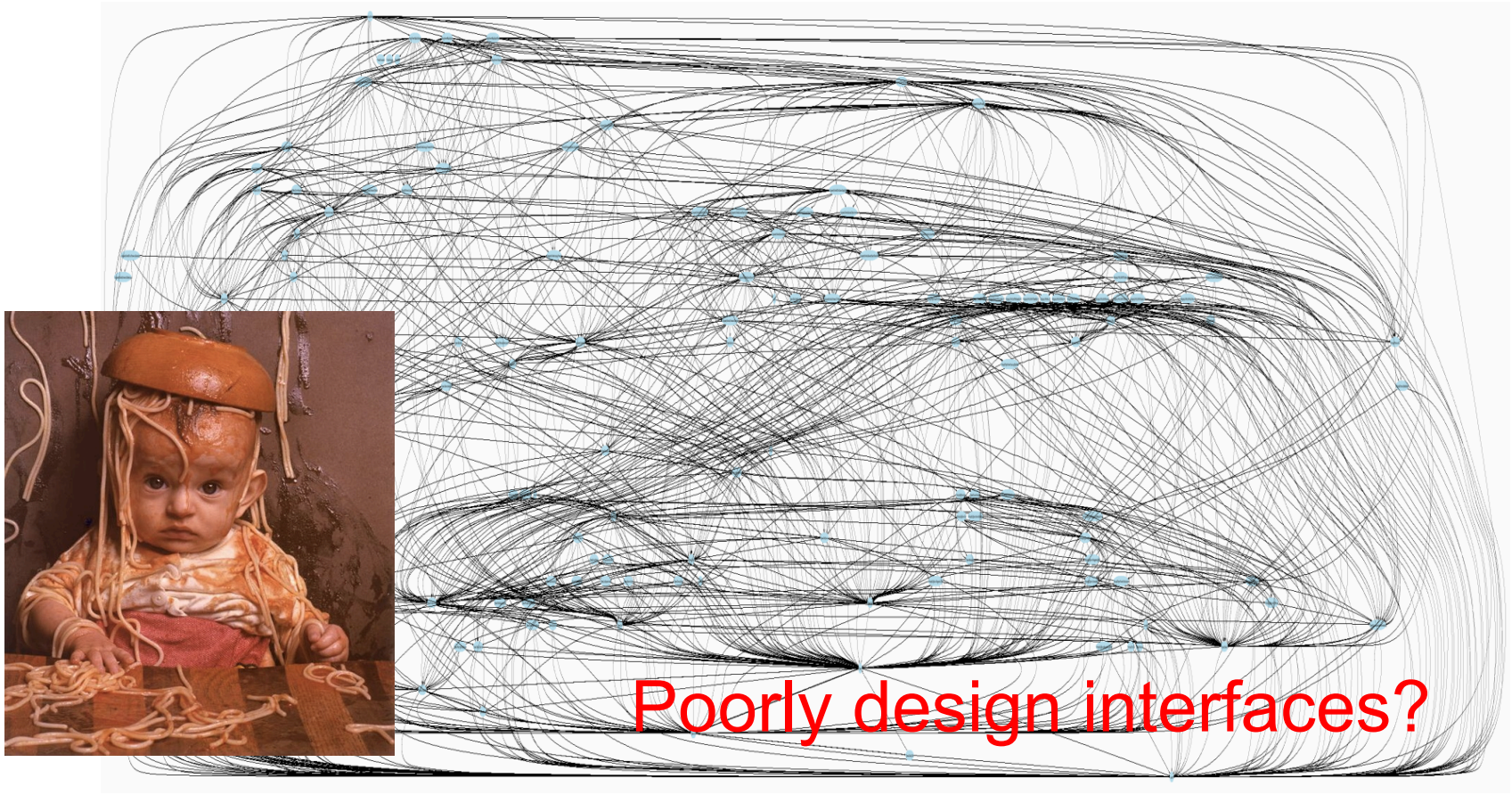
Library Dependencies: Spaghetti

Paying attention to what your library depends on and how it interacts with other libraries is important.

Spaghetti might be quick to make and have good taste – but, it can be a sloppy mess to eat!

Creating libraries quickly without a clear understanding of their interface can lead to code that is disorganized and monolithic.

Dependency Mess



Library Dependency Directionality

One-Way Dependency

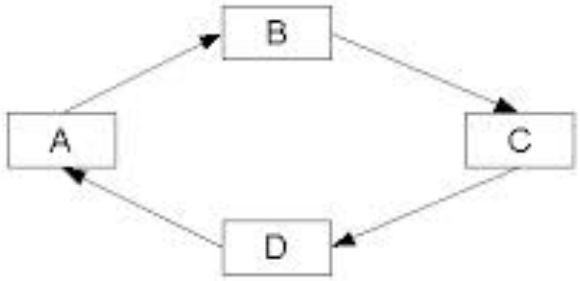
Good library design and usage are those that promote a one-way dependency. That is, a library at a lower level should not depend on a library “above” it and a library “below” it.

Library Dependency Directionality

Circular Dependency

Bad library design and usage are those that exhibit **circular** dependencies.

That is, a library A that depends on the library B which depends on the library A or some transitive relationship amongst a collection of libraries.



Languages and Libraries

Programming languages provide different facilities to support the construction of reusable libraries.

This includes: code organization, data structure constructs, types, information hiding, and interface definition.

Languages and Libraries: Browser JavaScript

Despite JS being the most ubiquitous language supported by every platform it still has a poor mechanism for library construction – in fact, it has **no language support** at all!

Libraries are constructed from global objects with properties that are dynamically constructed with little to no access restrictions!

Languages and Libraries: Browser JavaScript

```
// Library: people.js:
var PeopleLibrary = (function () {
  function Person(name) {
    this.name = name;
  }
  return {
    newPerson: function (name) {
      return new Person(name);
    }
  };
})();

// Use:
PeopleLibrary.newPerson('Mia');
```

Languages and Libraries: Server JavaScript

```
// people.js
function Person(name) {
  this.name = name;
}

exports.newPerson = function (name) {
  return new Person(name);
};

// Use:
var people = require('people');
var mia = people.newPerson('Mia');
```

Languages and Libraries: Python

```
// people.py
class _Person:
    def __init__(self, name):
        self.name = name

def newPerson(name):
    _Person(name)

// Use:

import people
mia = people.newPerson(“Mia”)
```

Languages and Libraries: Java

```
// cs220.people.Person
public class Person {
  private String name;
  private Person(String name) {
    this.name = name;
  }
  String getName() { return name; }

  public static Person newPerson(String name) {
    return new Person(name);
  }
}
// Use:
import cs220.people.Person;
Person.newPerson(“Mia”);
```

Languages and Libraries: Scala

```
// cs220.people.Person
private [people] class Person (private val name: String) {
  protected [people] def getName: String = name
}
class Student private (private val name: String,
                       private val id: Int)
  extends Person(name) {
  def getFirstName: String = getName
  def getID: Int = id
}
object Student { def newPerson(n,id): Person = new Student(n,id) }

// Use:
import cs220.people.Student._;
newPerson(“Mia”, 1234)
```