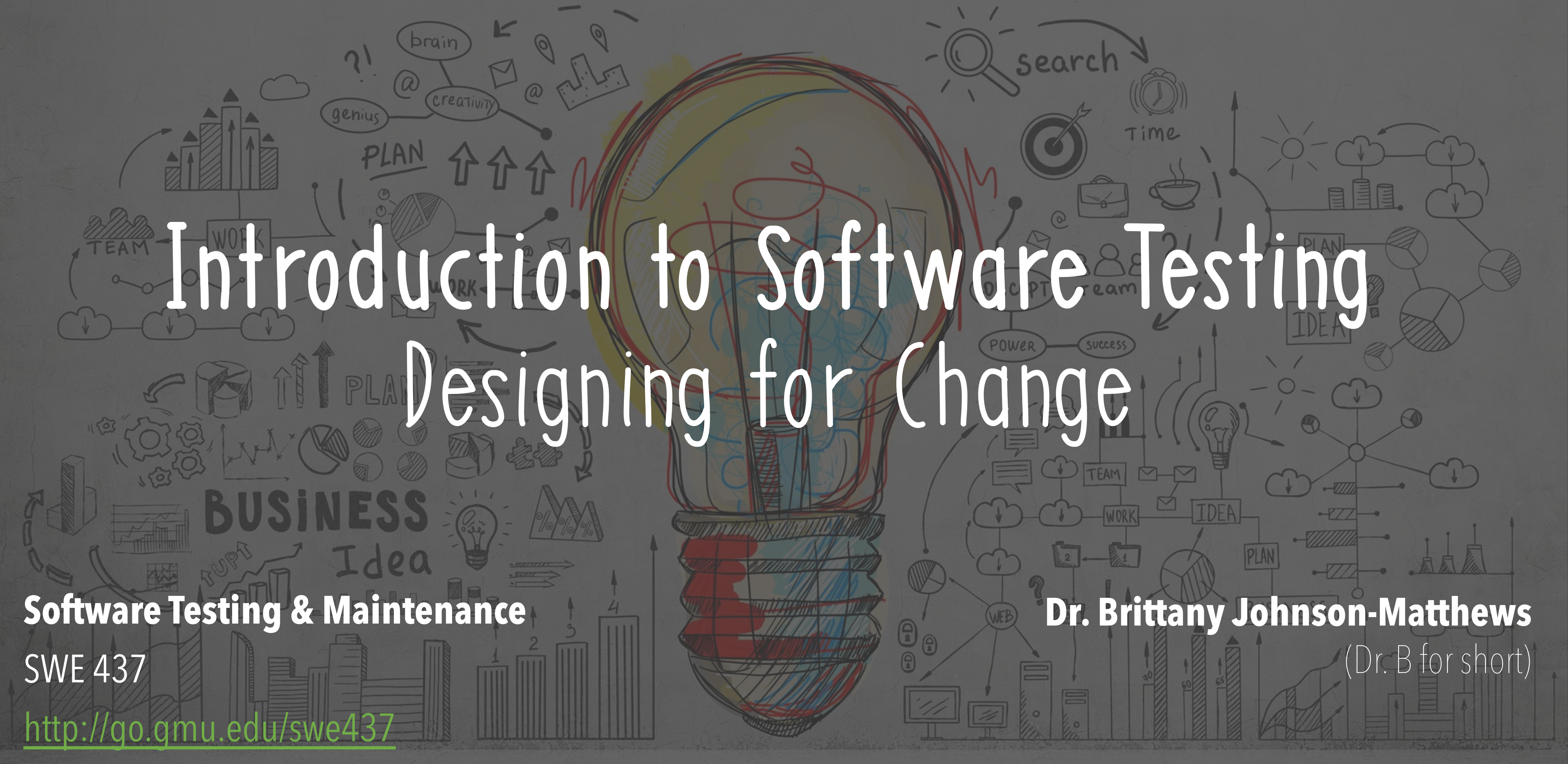


Introduction to Software Testing

Designing for Change

Software Testing & Maintenance

SWE 437

http://go.gmu.edu/swe437

Dr. Brittany Johnson-Matthews

(Dr. B for short)

Designing for maintainability

1. Integrating software components
2. Sharing data and message passing
3. Using design patterns to integrate

Designing for maintainability

1. **Integrating software components**
2. Sharing data and message passing
3. Using design patterns to integrate

Modern software is connected

Modern programs **rarely** live in isolation

- they **interact** with **other programs** on the same computer
- they use **shared library** modules
- They **communicate** with programs on **different computers**
- Data is **shared** among multiple computing devices

Web applications communicate across a network

Mobile applications live in a complex ecosystem

Web services connect **dynamically** during execution

Distributed computing is now common

Why is integration hard?

Networks are **unreliable**

Networks are **slow**

- multiple orders of magnitude slower than a function call

Programs on different computers are **diverse**

- different languages, operating systems, data formats…
- connected through diverse hardware and software applications

Change is **inevitable** and **continuous**

- programs we connect with change
- host hardware and software changes

Distributed software must use extremely low coupling

Coupling explained

Tight coupling: dependencies encoded in logic

- changes in A may require changing logic in B
- This used to be common

Loose coupling: dependencies encoded in the structure and data flows

- changes in A may require changing data uses in B
- goal of data abstraction and object-oriented concepts

Extremely loose coupling (ELC): dependencies encoded only in the data **contents**

- changes in A only affects the contents of B's data
- motivating goal for distributed software and web apps

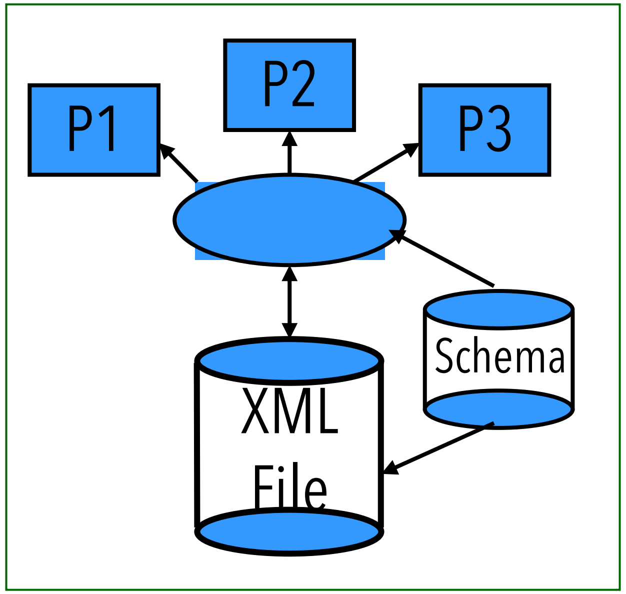
The issues are about how we share data…

XML supports ELC

Data is **passed directly** between components

Components must agree on **format, types,** and **structure**

XML allows data to be **self-documenting**



```
<book>
   <author>Steve Krug</author>
   <title>Don’t Make Me Think</title>
</book>
<book>
   <author>Don Norman</author>
   <title>Design of Every Day Things</title>
</book>
```

P1, P2, and P3 can see the **format**, **contents**, and **structure** of the data

Free parsers are available

Discussion

Discuss in groups

Explain coupling to each other

Have you used tight coupling?

Have you used loose coupling?

Have you used extremely loose coupling?

Designing for maintainability

1. Integrating software components
2. **Sharing data and message passing**
3. Using design patterns to integrate

Sharing data

1. **Transferring files**

    - one program **writes** to a file that another later **reads**
    - both programs need to **agree** on:

        file name, location, and format

        timing for when to read and write it

2. **Sharing a database**

    - replace a file with a database
    - most decisions are **encapsulated** in the **table design**

Sharing data

3. **Remote procedure invocation**

   - one program **calls a method** in another application
   - communication is **real-time** and **synchronous**
   - Data are passed as **parameters**

4. **Message passing**

   - one program sends a message to a common **message channel**
   - other programs read the messages at a later time
   - programs must **agree** on the channel and message format
   - communications is **asynchronous**
   - **XML** is often used to implement encoded messages

Message passing

Message passing is asynchronous and very loosely coupled.

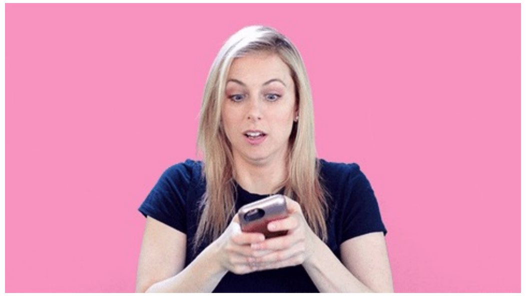
Telephone calls are **synchronous**

This introduces **restrictions**:

- other person must be there
- communication must be real time

Voicemail and texts are **asynchronous**

- messages left for **later retrieval**
- **real-time** aspects less important



Benefits of message passing

Message-based software is **easier to change** and **reuse**

- better **encapsulated** than shared database
- more **immediate** than file transfer
- more **reliable** than remote procedure invocation

Software components **depend less** on each other

Several **engineering** advantages:

- **reliability**
- **maintainability** & changeability
- security
- scalability

Drawbacks of message passing

Programming model is different – and complex

- **universities** seldom teach event-driven software (SWE 432)
- **logic** is distributed across several software components
- **harder** to develop and debug

Sequencing is harder

- **no guarantees for when** messages will arrive
- messages sent in one sequence may arrive **out of sequence**

Some programs require applications to be **synchronized**

- shopping requires users to **wait** for responses
- most web apps are synchronized
  - **Ajax** allows asynchronous communications

Message passing is **slower**, but good middleware helps

Discussion

Discuss in groups

Have you used message passing?

Have you learned about message passing?

If yes, describe to other members of the group

If not, do you understand message passing?

Designing for maintainability

1. Integrating software components
2. Sharing data and message passing
3. **Using design patterns to integrate**

Enterprise applications

Enterprise systems contain hundreds or thousands of separate applications

- custom-built, third party vendors, legacy systems…

- multiple tiers with different operating systems

Enterprise systems often ***grow*** from disjoint pieces

- just like a town or **city** grows together and slowly integrates

Companies want to buy the **best package** for each task

- then **integrate** them!

Thus, integrating diverse programs into a coherent enterprise application will be a challenge for years to come.

Information portals

Information portals aggregate information from multiple sources into a single display to avoid making the user access multiple systems.

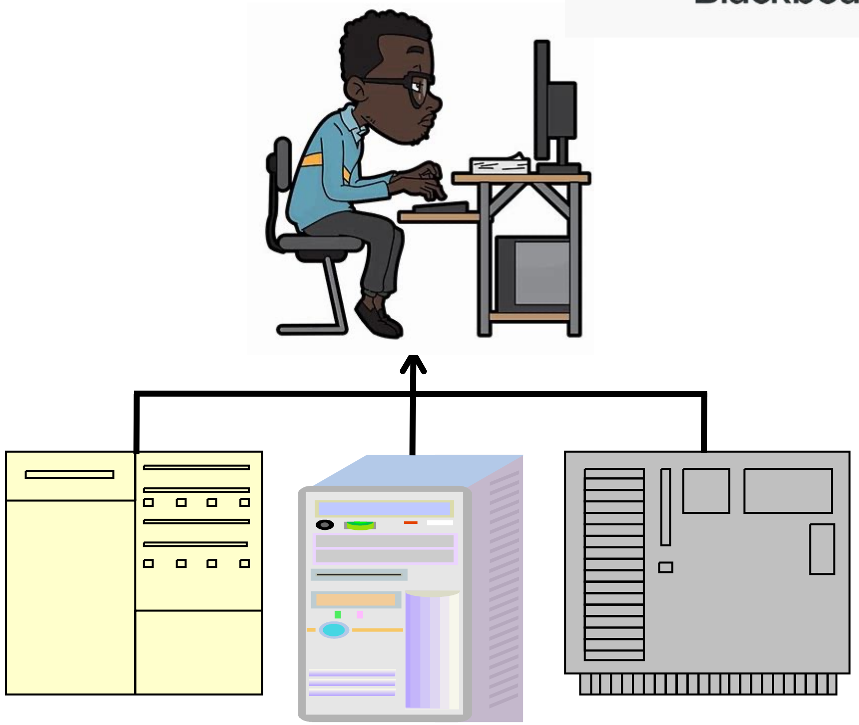
myMASON Blackboard

Answers are pulled from different places

- e.g., grade sheets, syllabus, transcript…

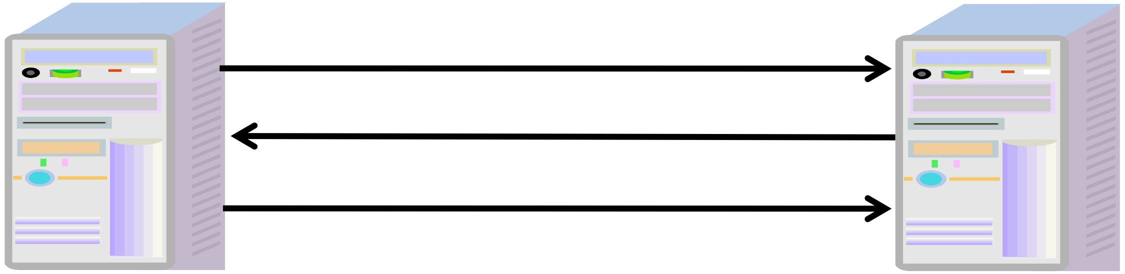
Information portals divide the screen into different zones

They should make it easy to **move data** between zones



Business-to-business integration

Integration between two separate businesses.



Business functions are available from outside suppliers or business partners

- e.g., online travel agents use **credit card** service

Integration may occur "**on-the-fly**"

- a customer may seek the **cheapest price** on a given day

Standardized data formats are critical

Data replication (observer)

Making data needed by multiple applications available where it's needed.

Multiple business systems often need the **same data**

- e.g., student **email address** is needed by professors, registrar, department, IT…

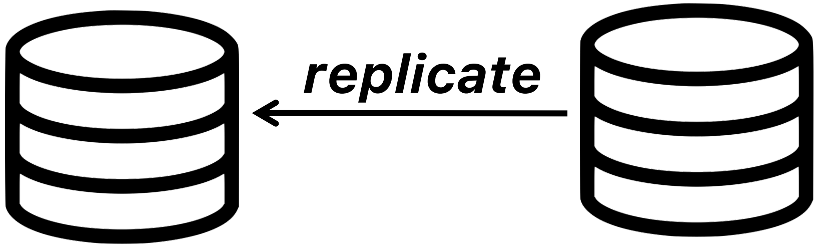
- when email is **changed** in one place, all copies must change

Data replication can be implemented in many ways

- built into the **database**

- **export** data to files, re-import them to other systems

- use **message-oriented** middleware



Shared business functions (builder)

Same functions used by several applications.

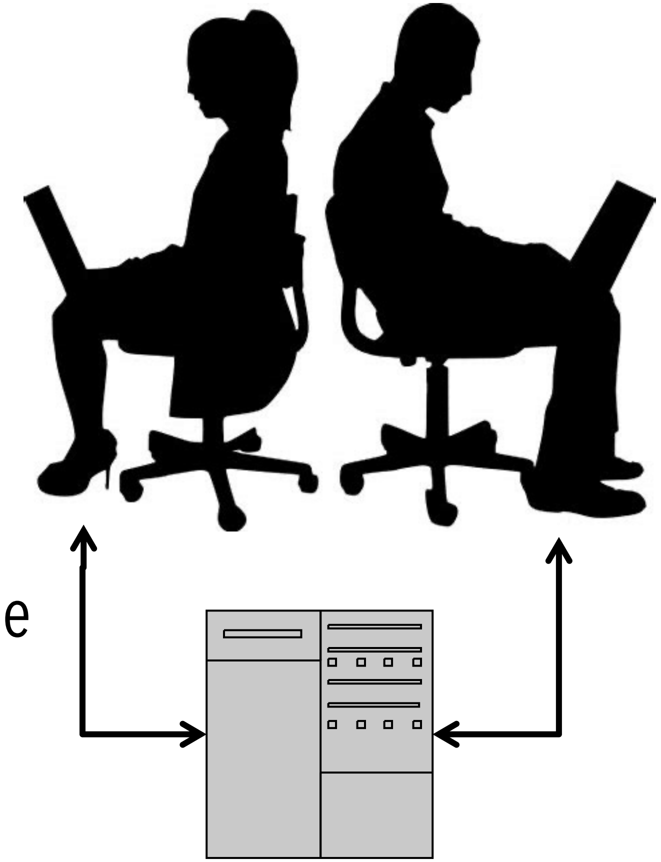
Multiple users need the same **function**

- e.g., whether a **particular course** is taught this semester
- student, instructor, admins

Each function should only be **implemented once**

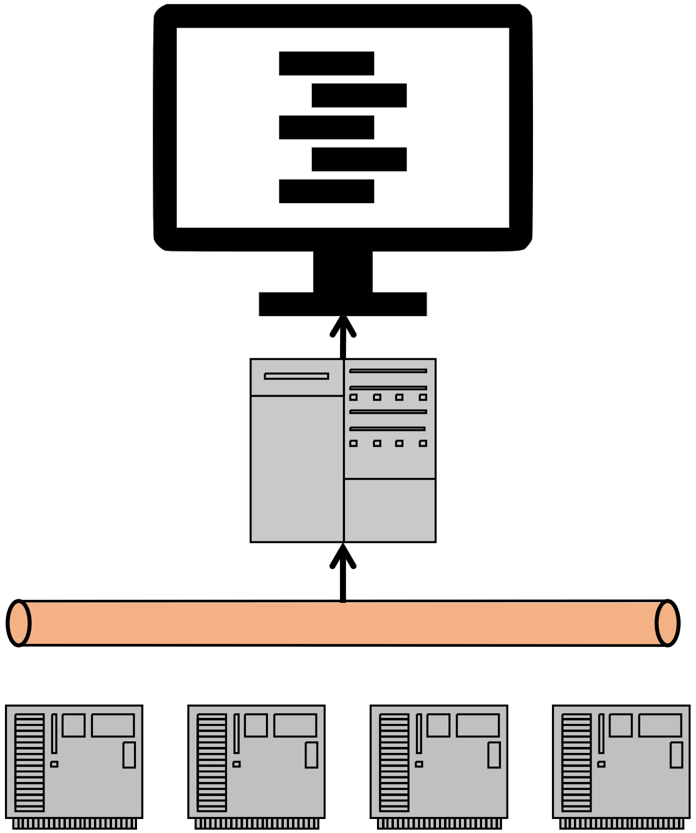
If the function only **accesses data** to return result, duplication is simple

If function **modifies data**, race conditions can occur



Service-oriented architecture (SOA)

A service is a well-defined function that is available from anywhere.



Managing a collection of useful services is a **critical function**

- service **directory**
- each service needs to describe its **interface** in a generic way

A mixture of **integration** and **distributed** application

Patriot web

A Self Service Web Site for Students, Faculty, and Staff

Other common design patterns

Abstract Factory

provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Singleton

ensures a class has only one instance, and provide a global point of access to it.

Visitor

represents an operation to be performed on the elements of an object structure.
lets you define a new operation without changing the classes of the elements on which it operates.

https://sourcemaking.com/design_patterns

Summary: coupling, coupling, coupling

We have always known coupling is important.

Goal is to **reduce the assumptions** about exchanging data

- loose coupling means fewer assumptions

A local **method call** is very **tight** coupling

- same language, same process, typed params, return value

Remote procedure call has **tight** coupling, but with the complexity of distributed processing

- the **worst of both** worlds
- results in systems that are **hard to maintain**

Message passing has extremely loose coupling

Message passing systems are easy to maintain.