CSE 219
COMPUTER SCIENCE III

DESIGN REVIEW

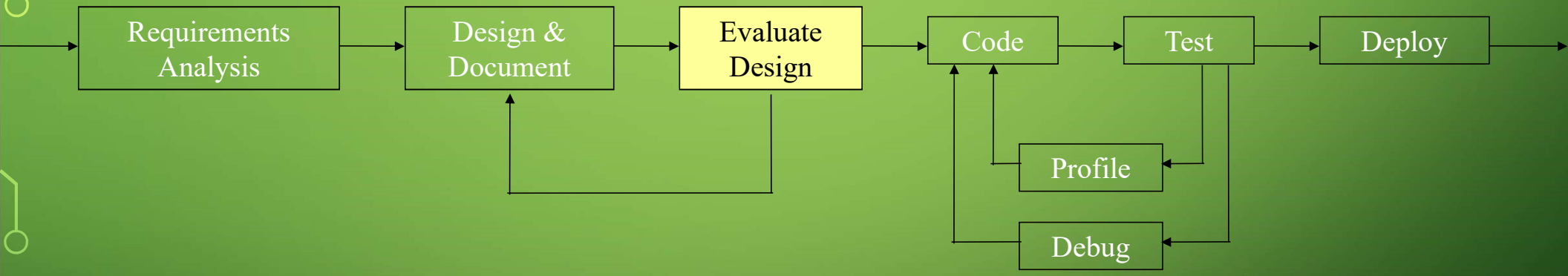
SLIDES COURTESY:

PROF. RICHARD MCKENNA

STONY BROOK UNIVERSITY

Software Development Life Cycle

- We're making progress



Evaluating a Design

- During the design of a ***large program***, it is worthwhile to step back periodically & attempt a comprehensive evaluation of the design so far
  - called a *design review*

Design Reviews are not just for Software

Who performs the design review?

- Design review committee
- Members should include:
  - varied perspectives
  - some from the project
  - some external to the project
- All should be familiar with the design itself

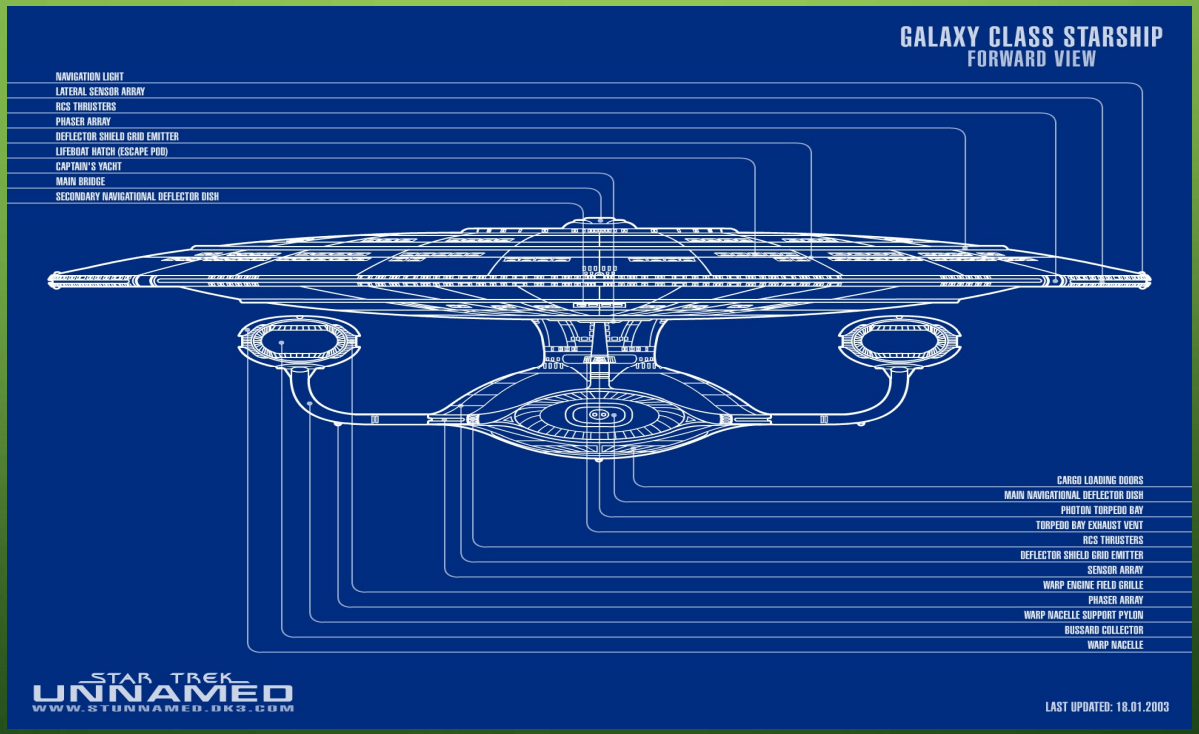
Write up a Testing Process Specification (TPS) report for management

- The Test Procedures are developed from both the Test Design and the Test Case Specification. The document describes how the tester will physically run the test, the physical set-up required, and the procedure steps that need to be followed.
- The IEEE829 standard defines ten procedure steps that may be applied when running a test.

One World Trade Center

There is no perfect design

- Is the design adequate?
- Will do the job with adequate performance & cost?



Critical Design Issues

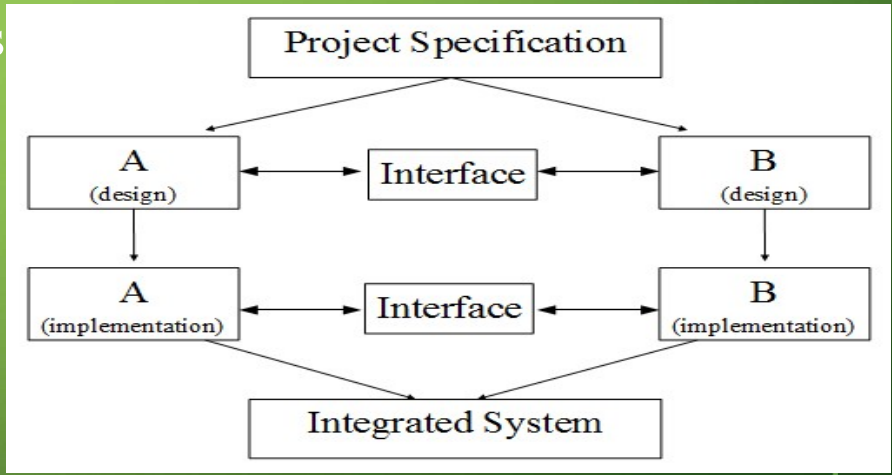
- Is it correct?
  - Will all implementations of the design exhibit the desired functionality?
- Is it efficient?
  - Are there implementations of the design that will be acceptably efficient?
- Is it testable & maintainable?
  - Does the design describe a program structure that will make implementations reasonably easy to build, test and maintain?
- Is it modifiable, extensible, & scalable?
  - How difficult will it be to enhance the design to accommodate future modifications?

Other Considerations

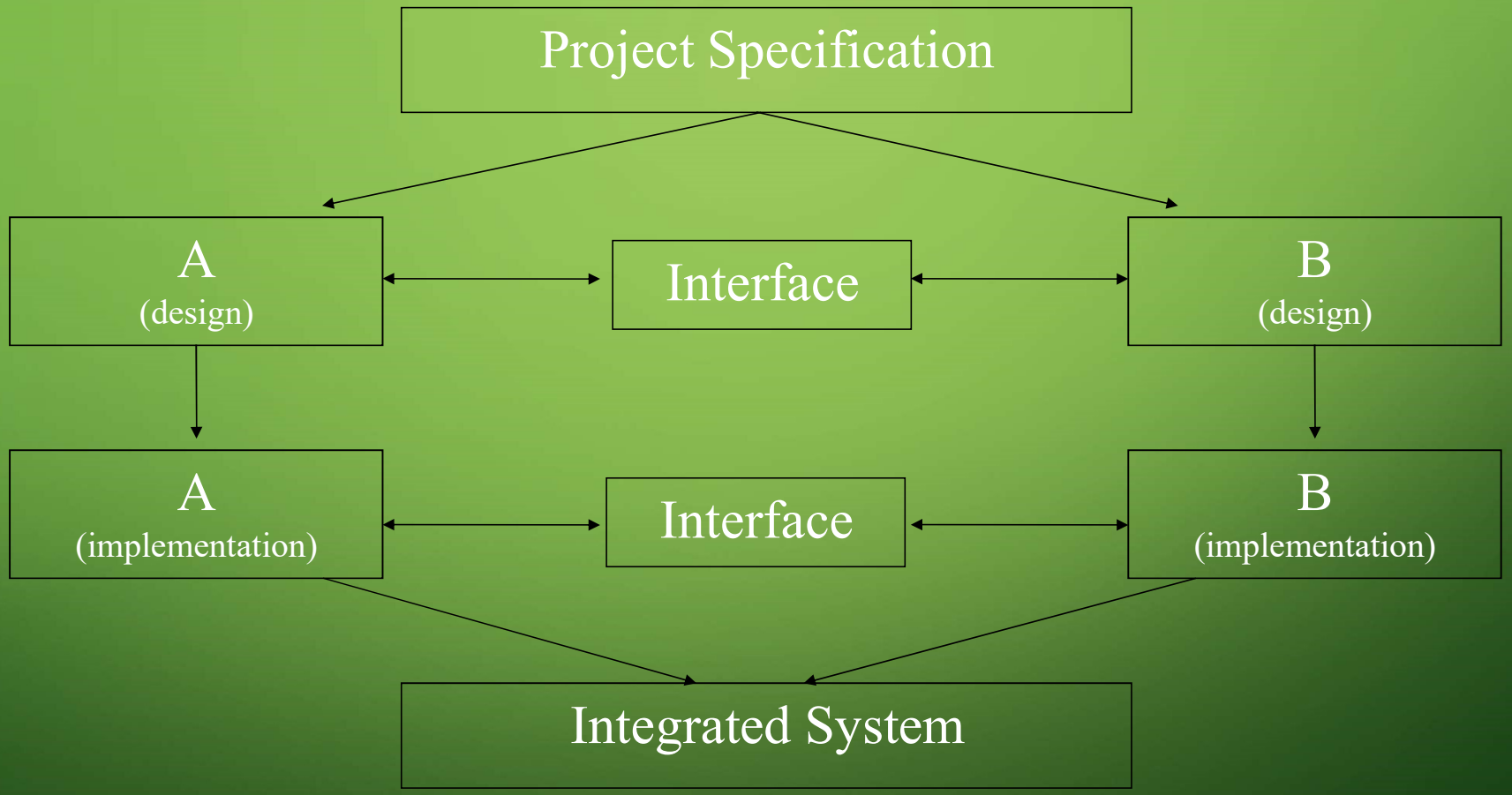
- Are the classes independent?
- Is there redundancy?
- Do they manage & protect their own data?
- Can they be tested individually?
- Do they promote code reuse?
- Is data and control flow clear or complex?

Modular Design Methodology

- Large software projects are divided up into separate modules
  - i.e. groups of related classes
- **Decompose**
  - large programming problems into smaller ones
  - i.e. sub-problems
- **Solve**
  - the sub-problems independently
  - modules solve sub-problems
- **Assemble**
  - the modules to build full system
  - called system integration
    - scariest parts of software development
    - serious design flaws can be exposed



Modular Design

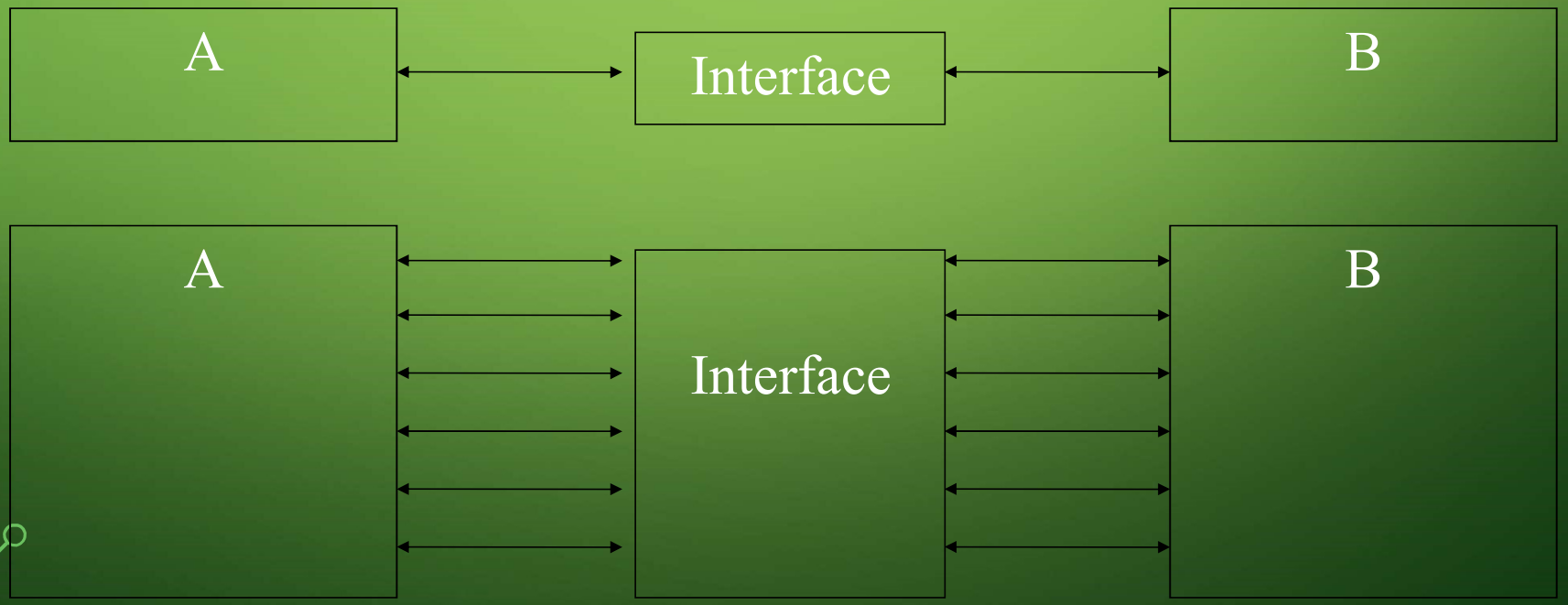


What makes a good *modular* design?

- Connections between modules are ***explicit***
- Connections between modules are ***minimized***
  - called narrow interfaces
- Modules use ***abstraction*** well
- Implementation of modules can be done ***independently***
  - modules avoid duplication of effort

More on Narrow Interfaces

- A module should have access to only as much information as it needs to work
  - less chance of misuse
  - less coordination needed between team members
    - fewer meetings necessary (YAY!)



Design is Difficult

- Where do you begin?
- When is the design complete?

Good Design comes with Experience

- It takes time to become an expert

POSITION INFORMATION

- **Company:** AVID Technical Resources
- **Location:** Woodbury, NY
- **Status:** Full Time, Employee
- **Job Category:** IT/Software Development
- **Work Experience:** 10+ to 15 Years
- **Occupations:** Software/System Architecture Usability/Information Architecture Web/UI/UX Design
- **Salary/Wage:** $0.00 - $210,000.00 /year

POSITION DESCRIPTION

Java Architect

Please only reply with Java Architects that have created white papers that Developers have followed.

Must have a car to get to my client in Woodbury, Long Island.

Contract will last at least 1 year.

1st is a phone screen for 30 minutes, then a face to face for 2 hrs with the VP, and the other Architect.

The goal is to design for the best performance and create processes to be followed.

Create the white papers that will be followed by the developers.

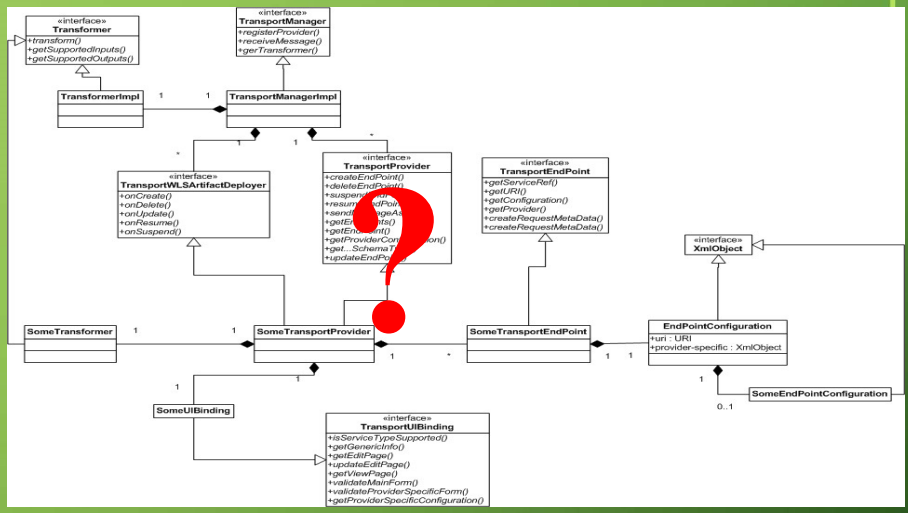
The consultant must be on-site every day, no telecommuting.

One architect is there right now, the other is retiring.

Performance monitoring, reporting, and tuning of Oracle databases.

How can a design be reviewed for correctness?

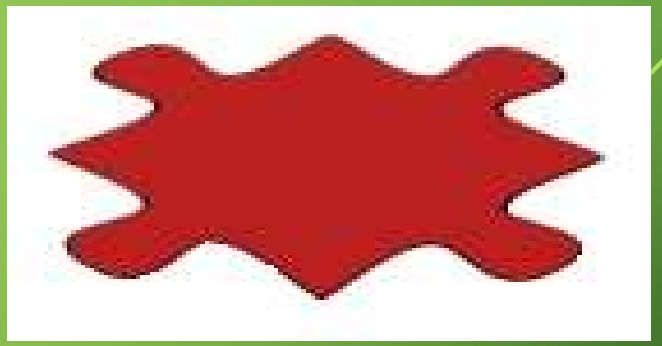
- Testing is not possible
- Verification is not possible
  - unless it uses a formal language
  - typically not practical



- Use proven, systematic procedure
  - examine both local & global properties of the design

Local Properties

- Studying individual modules



- Important local properties:
  - consistency
    - everything designed was as specified
  - completeness
    - everything specified was designed
  - performance
    - running time
    - storage requirements

Global Properties

- Studying how modules fit together
  - after examining local properties

- Is all the data accounted for?
  - from original SRS
    - exists properly in a module
    - rules are properly enforced

Reviewing Design Structure

- Two key questions:
  - Is there an abstraction that would lead to a better modularization?
  - Have we grouped together things that really do not belong in the same module?

- Structural Considerations
  - Coherence of procedures and types
  - Communication between modules
  - Reducing dependencies

Coherence of procedures and types

- A procedure (method) in a design should represent a single, coherent abstraction

- Examine each method to see how crucial it is for the data type
  - does it need to access instance or static variables of the class

- Move irrelevant methods out to another location

- Common with static functions

Communication between Modules

- Careful examination can uncover important design flaws
  - think of handing your HW 3 Design to another student for inspection
    - Do these pieces really fit together?
  - to improve any design:
    - act like a jerk when examining your own design
    - ask questions that a jerk would ask
    - make sure your design addresses these jerky questions

Reducing Dependencies

- A design with fewer dependencies is generally better than one with more dependencies.

- What does this mean?
  - Make the design of each component dependent on as few other components as necessary
  - Example of bad framework design:
    - Every class in your framework uses every other class in your framework in one way or another
      - This would be terribly complex to test & modify

Look for Antipatterns

- Common patterns in programs that use poor design concepts
  - make reuse very difficult
  - source: http://www.antipatterns.com/

- Ex:
  - The Blob
  - Spaghetti Code

Development AntiPattern:

The Blob

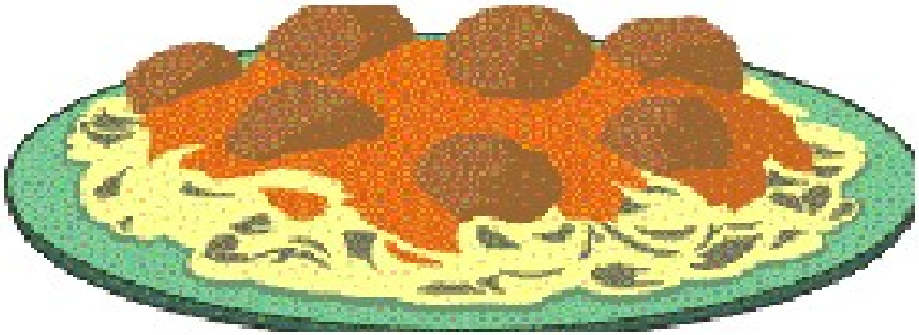
- **Symptoms**
  - Single class with many attributes & operations
  - Controller class with simple, data-object classes.
  - Lack of OO design.
  - A migrated legacy design
- **Consequences**
  - Lost OO advantage
  - Too complex to reuse or test.
  - Expensive to load

MITRE

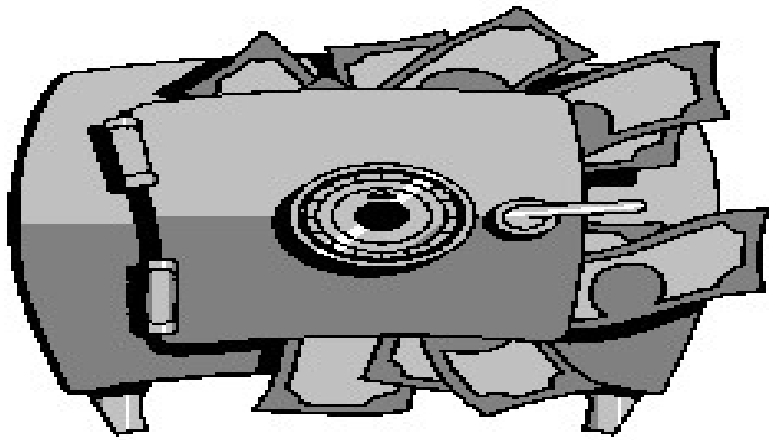
Development AntiPattern:

Spaghetti Code

☞ **spa·ghet·ti code** [Slang] an undocumented piece of software source code that cannot be extended or modified without extreme difficulty due to its convoluted structure.



Un-structured code
is a liability



Well structured code
is an investment.

MITRE

Design Principles

*A **design principle** is a basic tool or technique that can be applied to designing or writing code to make that code more maintainable, flexible, or extensible.*

- Cohesion
- The Open-Closed Principle
- The Don’t Repeat Yourself Principle
- The Single Responsibility Principle
- The Liskov Substitution Principle

A cohesive class does

one thing

really well and

does not try to

do

or be

something else.

Cohesion. Cohesion measures the degree of connectivity among the elements of a single module, class, or object. The higher the cohesion of your software is, the more well-defined and related the responsibilities of each individual class in your application. Each class has a very specific set of closely related actions it performs.

The Open-Closed Principle (OCP)

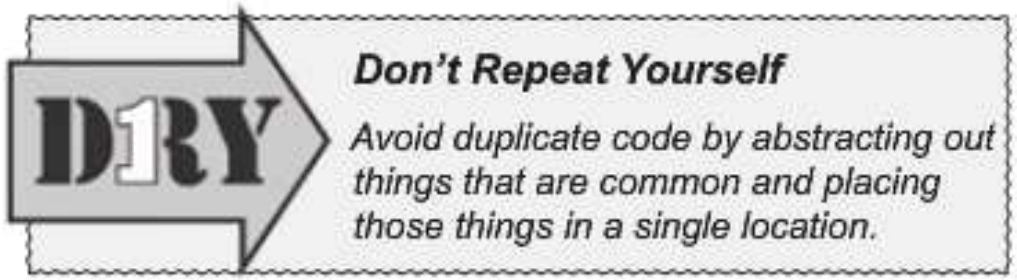
Our first design principle is the OCP, or the Open-Closed principle. The OCP is all about **allowing change**, but doing it **without requiring you to modify existing code**. Here's how we usually define the OCP:

Open-Closed Principle

Classes should be open for extension, and closed for modification.

The Don't Repeat Yourself Principle (DRY)

Next up is the Don't Repeat Yourself principle, or DRY for short. This is another principle that looks pretty simple, but turns out to be critical in writing code that's easy to maintain and reuse.



DRY is about having each piece of information and behavior in your system in a single, sensible place.

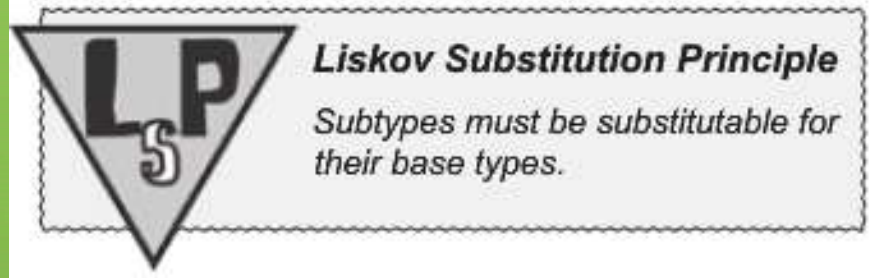
The Single Responsibility Principle (SRP)

The SRP is all about responsibility, and which objects in your system do what. You want each object that you design to have just one responsibility to focus on—and when something about that responsibility changes, you'll know exactly where to look to make those changes in your code.

Single Responsibility Principle

Every object in your system should have a single responsibility, and all the object's services should be focused on carrying out that single responsibility.

The Liskov Substitution Principle (LSP)



The LSP is all about well-designed inheritance. When you inherit from a base class, you must be able to substitute your subclass for that base class without things going terribly wrong. Otherwise, you've used inheritance incorrectly!

One more principle

Don't create problems to solve problems.

These principles may lead to making a framework

Application A

-pushTransactionOnStack(…

-execAddItem(…

-moveStackCounter(…

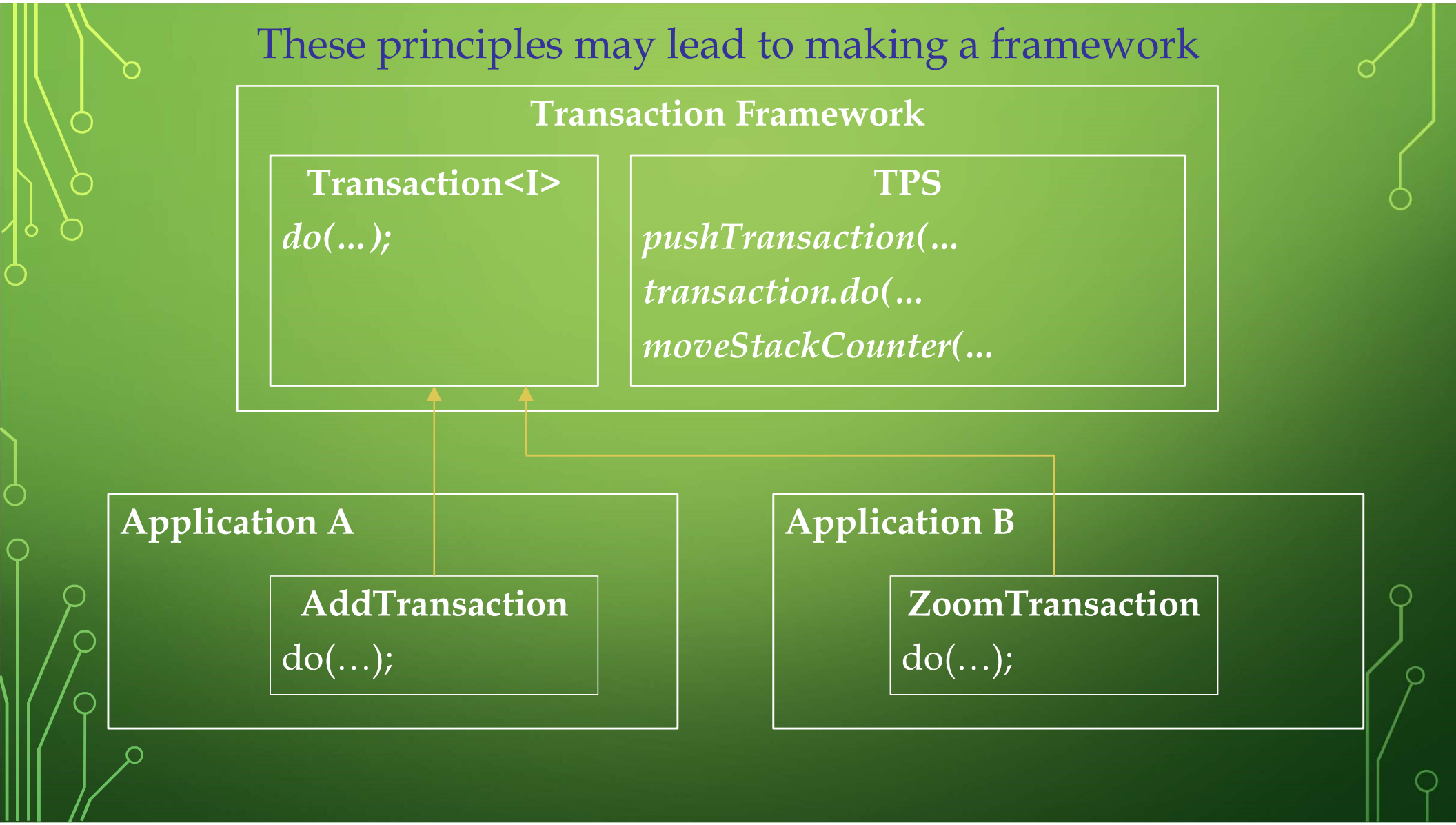
Application B

-pushTransactionOnStack(…

-execZoomTransaction(…

-moveStackCounter(…

These principles may lead to making a framework



So what's left?

- Designing with Exceptions
- Test Driven Design
- Implementation Strategies
- Profiling
- Design Patterns