COMP 3522

Object Oriented Programming in C++

Week 10 Day 2

Agenda

1. Coupling/Cohesion
2. Refactoring
3. Design idioms
    1. Law of Demeter
    2. Liskov substitution principle
    3. Open-closed principle

COMP 3522

DESIGN IDIOMS

You've heard of design patterns…

- But we're going to start by talking about design idioms
- Design patterns tell us how to design systems
- **Design idioms** tell us how many (sometimes most) developers solve issues of:
  - Message passing
  - Communication
- Some developers call them **design principles**

The big questions

1. How do we assemble classes?
2. We know how to implement each class, but how should they interact? How deeply should a class reach into another class, for example?
3. How do we implement relationships between classes?

- Good software architecture begins with clean code
- **If the bricks aren't well made, the architecture of the building doesn't mean much.**

Criteria

- We want code that
  - **Behaves** correctly
  - **Tolerates change** aka it's soft aka it's easy to change
- When stakeholders change a feature, the difficulty in making a change should be proportional only to the **scope** of the change, not its **shape**
- There's no such thing as a system that is impossible to change, but there are systems that are practically impossible to change, i.e., cost of change > benefit of change

Benefits of structured OOP

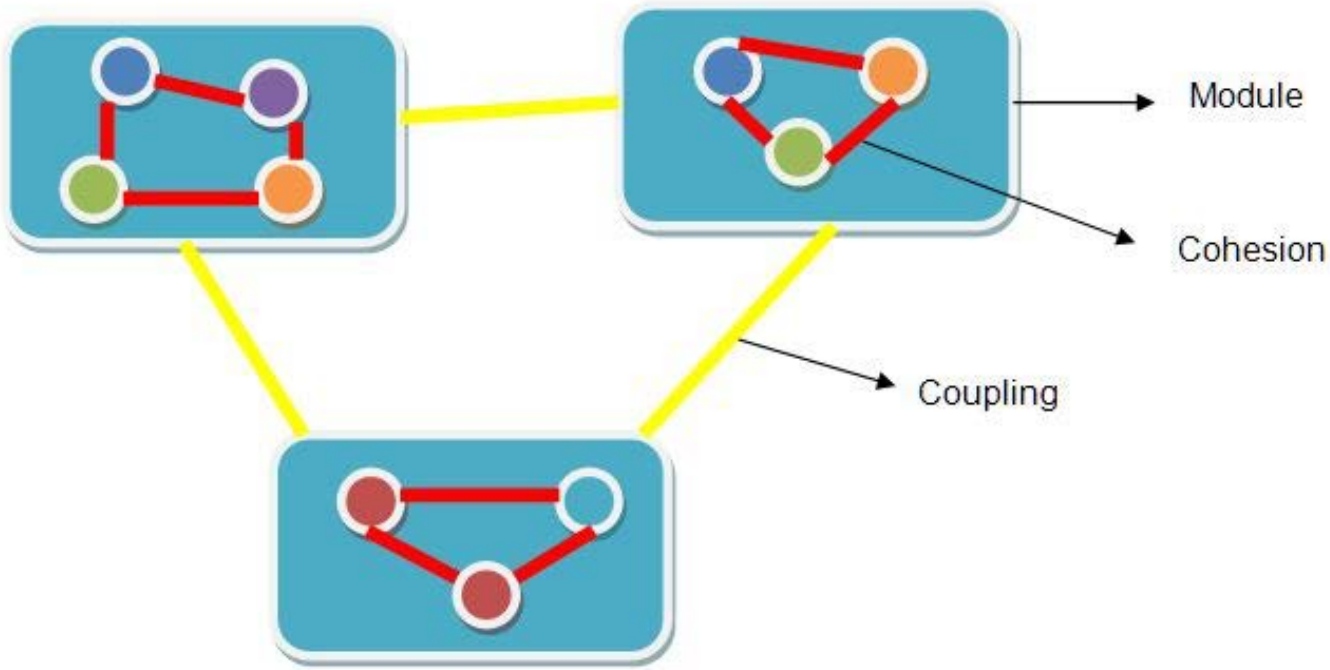
- Decompose a large-scale problem statements into modules and components
  - **Independent** developability
  - **Independent** deployability
- Plugin architecture
  - Modules that contain high-level policies are **independent** of modules that contain low-level details
- **To achieve this goal we have principles, patterns, and heuristics. Lots.**

Software changes (and changes and…)

- Software is **not written once**, like a novel
- Software is extended, corrected, maintained, ported, adapted, updated…
- The work is done by different people over time (often decades)
- **Software is either**
  - Maintained
  - Abandoned

Code and design quality

- If we are to be critical of code quality, we need some solid evaluation criteria
- Two important concepts for assessing the quality of code are:
  1. Coupling
  2. Cohesion



COUPLING AND COHESION

Coupling

- Coupling refers to links between separate units of a program
- If two classes depend closely on many details of each other, we say they are tightly coupled
- **We aim for loose coupling between modules**
- A class diagram provides (limited) hints at the degree of coupling

The strength of the connection between modules

Loose coupling

- We **aim** for loose coupling
- Loose coupling makes it possible to:
  - **Understand** one class without reading others
  - **Change** one class with little or no effect on other classes.
- Loose coupling increases maintainability

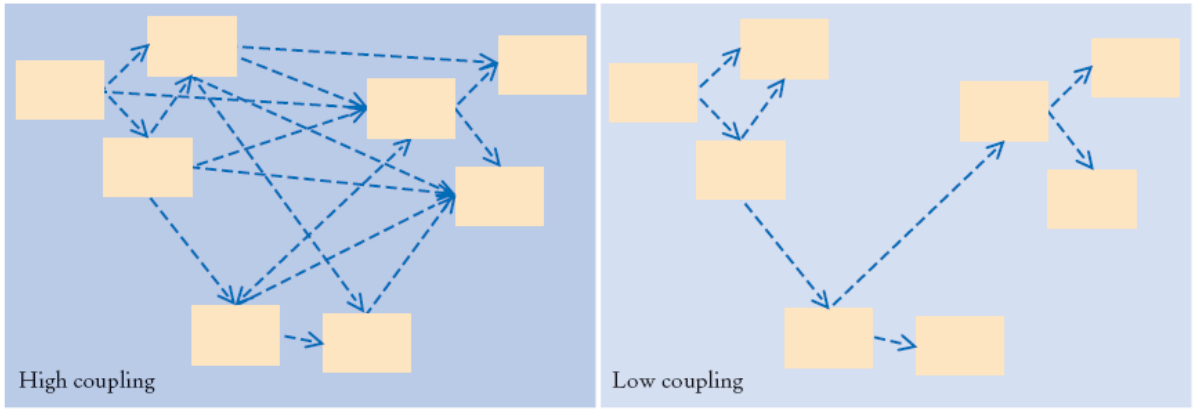
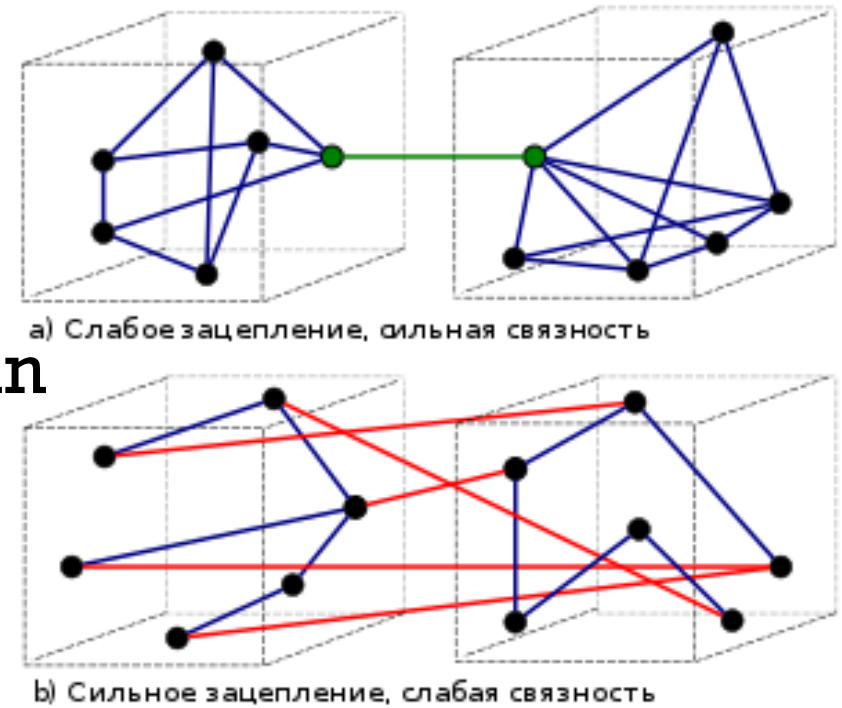


Figure 2 High and Low Coupling Between Classes

Tight coupling is bad

- We want to **avoid** tight coupling
- Changes to one class bring a **cascade of changes** to other classes
- Classes are **harder to understand** in isolation
- Flow of control between objects of different classes is **complex**
- **Difficult to test** classes in isolation



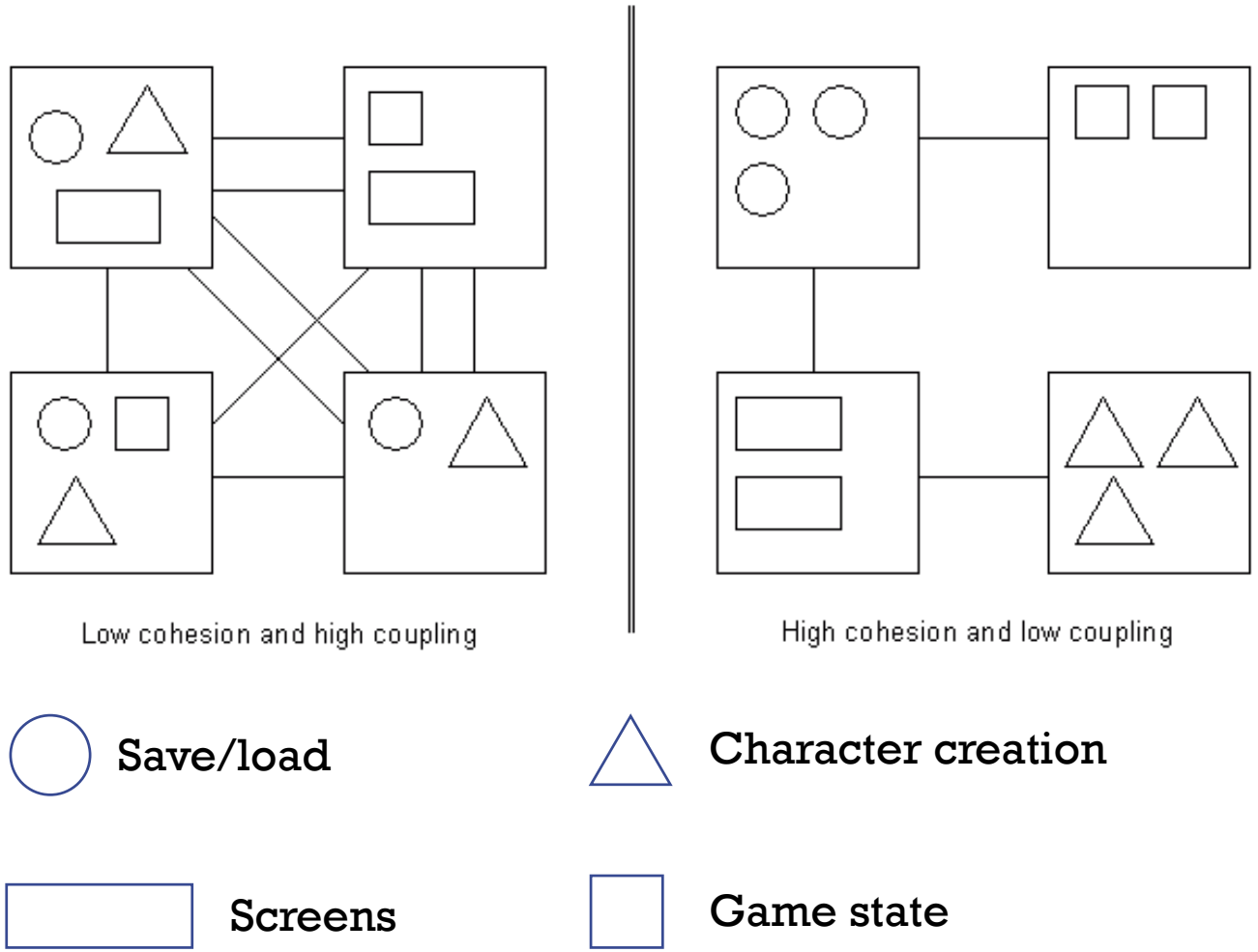
Cohesion

- Cohesion refers to the number and diversity of tasks that a single unit is responsible for
- If each unit is responsible for one single logical task, we say it has high cohesion
- **We aim for high cohesion inside modules**
- 'Unit' applies to functions, classes, and modules

The glue that holds a module together

High cohesion

- High cohesion makes it easier to:
  1. Understand what a class or method does
  2. Use descriptive and accurate names for variables, methods, and classes
  3. Reuse classes and methods (we love this!)



High cohesion

Class level:

- Classes should represent one single, well defined **entity**

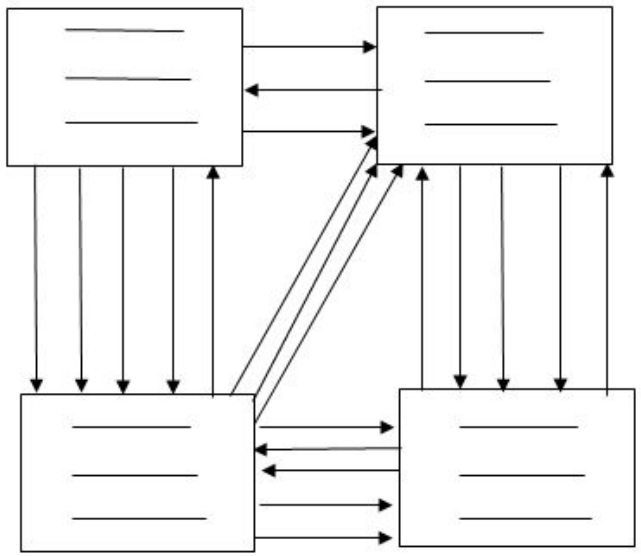
Method level:

- A method should be responsible for one and only one well defined **task**

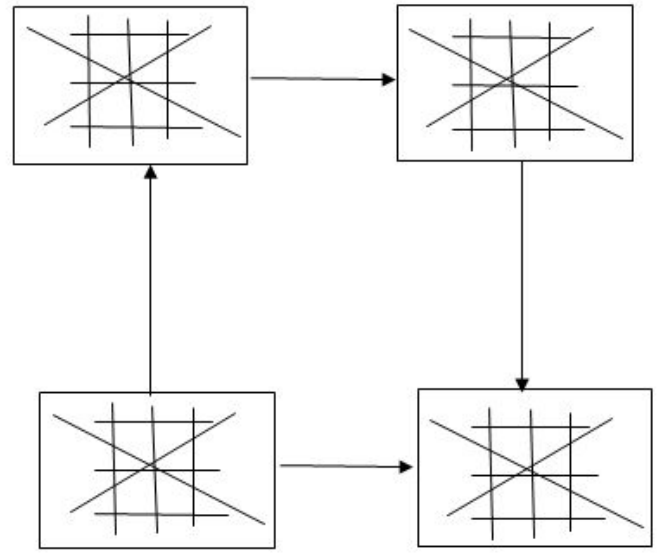
- We **avoid** loosely cohesive classes and methods
  - Methods perform multiple tasks # BAD
  - Classes have no clear identity # BAD

Design Principle – Coupling and Cohesion

Examples of Coupling and Cohesion



High Coupling
Low Cohesion



Low Coupling
High Cohesion

Which one is better from a software design point of view and why?

Symptom: code duplication

- Code duplication
  - is an indicator of **bad design**
  - makes maintenance harder
  - can lead to introduction of errors during maintenance
- One aim of reducing coupling and code duplication is to localize change
- When a change is needed, as few classes (and functions) as possible should be affected

Responsibility-Driven Design (RDD)

- Question: where should we add a new method (which class)?
- Self governance: each class should be responsible for manipulating its own data
- The **class that owns the data should be responsible for processing it**
- RDD leads to low coupling

Refactoring

- When classes are maintained, often code is added
- Classes and methods tend to become longer
- Every now and then, classes and methods should be refactored to maintain cohesion and low coupling
- Refactoring is the process of **restructuring existing computer code – changing the factoring – without changing its external behavior**
- Creates a more expressive internal architecture or object model to improve extensibility

Refactoring

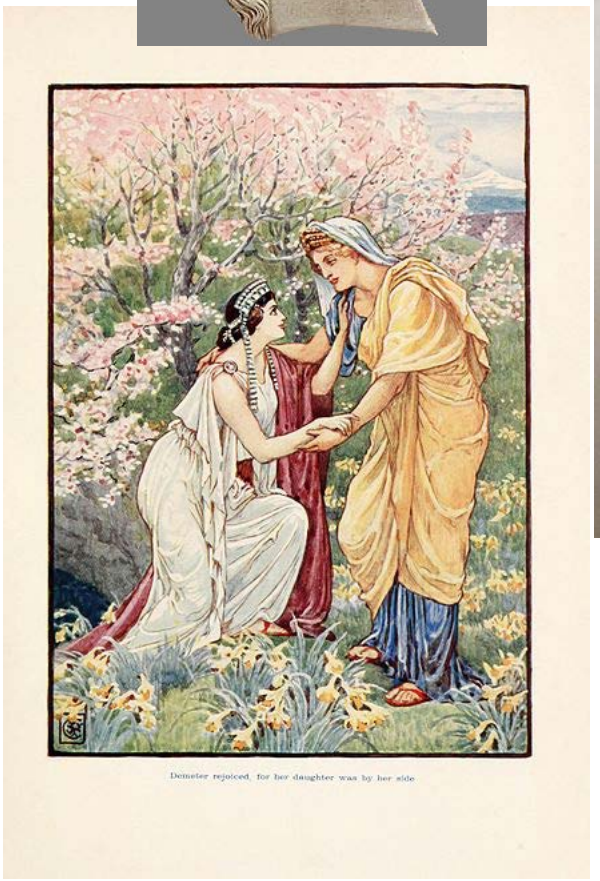
- Improves nonfunctional attributes of the software
  - improves readability
  - reduces complexity
  - improves maintainability
- When refactoring code, separate the refactoring from making other changes:
  - **First do the refactoring**, without changing the functionality
  - **Test** before and after refactoring to ensure that nothing was broken

LAW OF DEMETER

*pronunciation: di me’tr. The stress is on the middle, which is pronounced like e in equal

Demeter

- Greek mythology
- Goddess of the Harvest (grain, agriculture, harvest, growth, nourishment, grains, fertility)
- Responsible for the cycle of life and death
- Her daughter with Zeus, king of the gods, was Persephone
- Persephone was abducted by Hades
- During Demeter's grief, the earth went barren (winter)



Law of Demeter

- Also known as the Principle of Least Knowledge
- Supports loose coupling
- Proposed in 1987 by Dr. Ian Holland:
  1. Each unit should only have limited knowledge about units "closely" related to the current unit
  2. Each unit should talk only to its friends; don't talk to strangers
  3. **Only talk to your immediate friends.**

Why Demeter

"The Demeter project was named after Demeter because we were working on a hardware description language Zeus and we were looking for a tool to simplify the implementation of Zeus. We were looking for a tool name related to Zeus and we chose a sister of Zeus: Demeter.

Later we promoted the idea that Demeter-style software development is about growing software as opposed to building software. We introduced the concept of a growth plan which is basically a sequence of more and more complex UML class diagrams. Growth plans are useful for building systems incrementally."

* http://www.ccs.neu.edu/home/lieber/what-is-demeter.html

Description

- **An object should avoid invoking methods of a member object returned by another method**
- A given object should assume as little as possible about:
  - the structures or properties of other objects
  - its subcomponents.
- A corollary to the principle of least privilege: a module possesses only the knowledge and resources for its legitimate purpose
- Named after Demeter because it signifies a bottom-up philosophy of programming.

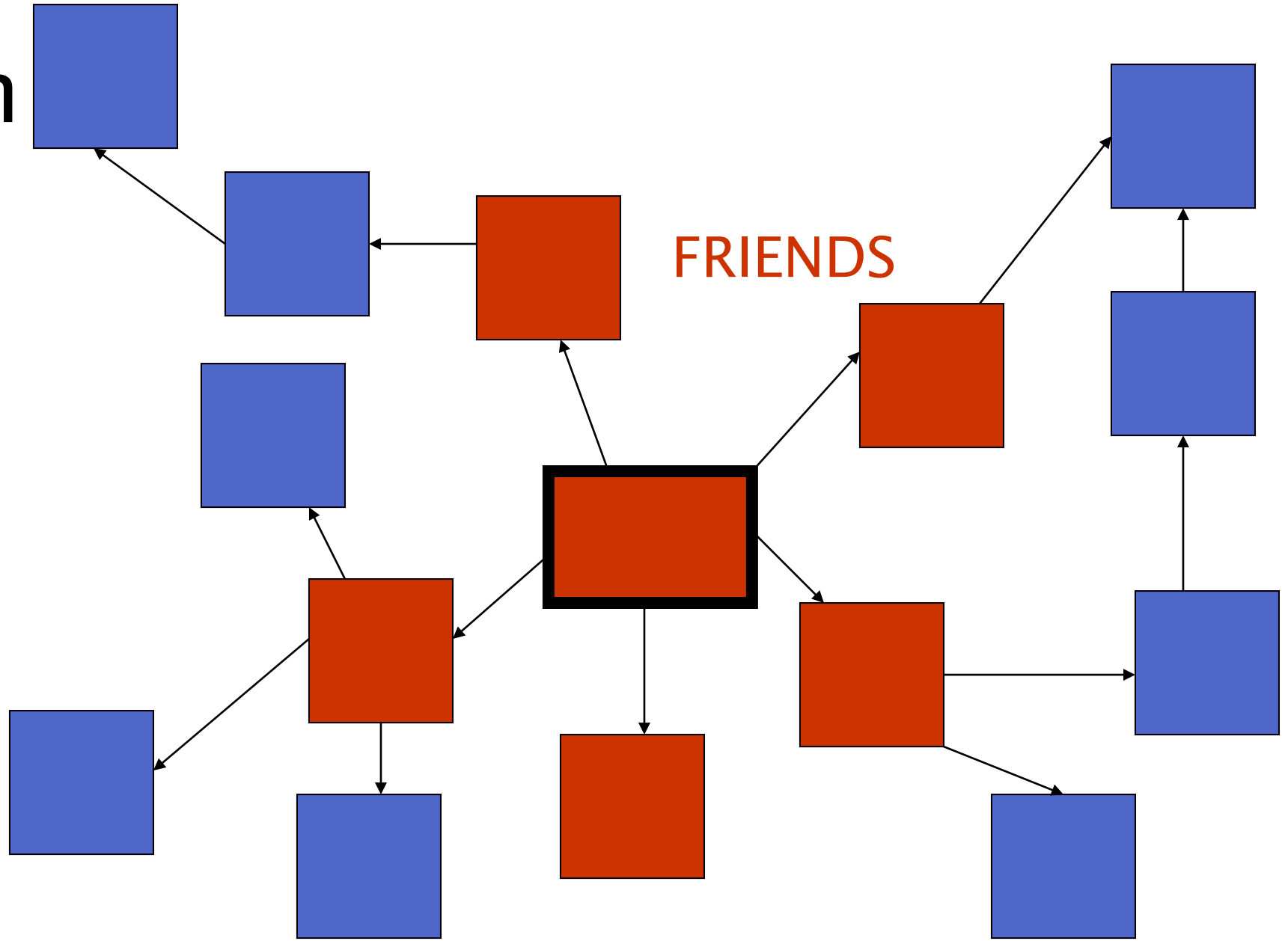
At the code level…

- For an object O which contains a method m, we say that the method m may only invoke the methods of the following kinds of objects:
  1. O itself
  2. m's parameters
  3. Any object created or instantiated in m
  4. O's direct component objects
  5. A global variable, accessible by O, in the scope of m.

ELI5

- **Only talk to your immediate friends**
- An object A can request a service from object B, but object A should not reach through object B to access yet another object C to request C's service
  - This would mean A must know B's internal structure, which is bad (it's private!)
  - An object should avoid invoking member methods of a member object returned by another object
- **When we want a dog to walk, we ask the dog to move its legs, we don't move its legs directly.**

Diagram



Example

```
struct myClass {
int myAttribute;
class1 memberClass1;

void myClassfunction(class1 arg1, class2 arg2) {
    //ok to access own member variables/functions
    cout << myAttribute;

    //ok to access parameter's variables/functions
    cout << arg1.attribute << arg2.attribute;
```

Example

```
struct myClass {
int myAttribute;
class1 memberClass1;

void myClassfunction(class1 arg1, class2 arg2) {
    //ok to access locally instantiated objects's variables/functions
    class1 tempClass1;
    cout << tempClass1.attribute;

    //ok to access member variable's/functions of own member variables
    cout << memberClass1.attribute;
```

Example

```
int globalAttribute;
struct myClass {
int myAttribute;
class1 memberClass1;

void myClassfunction(class1 arg1, class2 arg2) {
   //ok to access member variable's/functions of global variable
   cout << globalAttribute;
   }
};
```

Example

- Wallet example

Example

```
class classA{
    void function();
    classP p(); //returns object p
    classB b;
};
class classB {
    classC c;
};
class classC {
    void foo();
};
```

```
class classP {
     classQ q(); //returns object q
};
class classQ {
    void bar();
};

void classA::function()
{
    this.b.c.foo(); // NO!
    this.p().q().bar(); // NO!
}
```

Motivation

- Control information overload
  - We can only keep a limited set of items in short-term memory
- Software is more maintainable and manageable
- Looser coupling via better encapsulation
- Objects are less dependent on one another
- Robust to changes

- Disadvantage: we may have to write wrapper functions in an object that ‘wrap’ calls to its constituents

I have a paper for you to read

- Please read **The Paperboy, The Wallet, and The Law of Demeter**
  1. Only 8 pages long
  2. Confirms most of what I've said to you
  3. Has a great tangible code example inside.

https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf

LISKOV SUBSTITUTION PRINCIPLE

Robert C Martin*

1. **S**RP Single responsibility principle
2. **O**CP Open-closed principle
3. **LSP Liskov substitution principle**
4. **I**SP Interface segregation principle
5. **D**IP Dependency inversion principle

* Co-authored the agile manifesto

Why have SOLID, other heuristics?

- Make code that is:
  1. Understandable
  2. Flexible
  3. Maintainable
  4. (Dare we say) pleasant to work with!
- SOLID can form a core philosophy for methodologies like agile development or adaptive software development

Liskov substitution principle

- Developed by Dr Barbara Liskov at MIT in 1988
  - Let $\phi(x)$ be a property provable about objects x of type T
  - Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Liskov substitution principle (in English)

- To build software systems from interchangeable parts, those parts must adhere to a contract that permits the parts to be substituted for one another
- We create inheritance hierarchies of base classes and derived classes and use things like polymorphism
- Liskov tells us that a good design will always accept an instance of a derived class where an instance of the base class is expected

Liskov coding in real terms

1. No new **exceptions** should be thrown by methods of the derived class
2. **Preconditions** cannot be strengthened in a subtype
3. **Postconditions** cannot be weakened
4. **Invariants** of a base class must be preserved in the subclass
5. Objects are regarded as being modifiable only through their method (**encapsulation**)
6. Because subtypes may introduce methods that are not present in the supertype, the introduction of these methods may allow state changes in the subtype that are not permissible in the supertype.

We call this variance in programming

- Variance describes how subtyping between types relates to the subtyping between their components
- If Cat is derived from Animal, then an expression of type Cat can be used wherever an expression of type Animal can be used
- A **covariant** return type of a method is one that can be replaced by a "narrower" version in the derived class
- A **contravariant*** parameter type of a method is one that can be replaced by a "wider" version in the derived class

* C++ does not permit this

Covariant return type example

```
// Classes used as return types:
class A { }
class B extends A { }
// "Class B is more narrow than class A”
// Classes demonstrating method overriding:
class C {
    A getFoo() { return new A(); }
}
class D extends C {
    //Overriding getFoo() in parent class C
    B getFoo() { return new B(); }
}
```

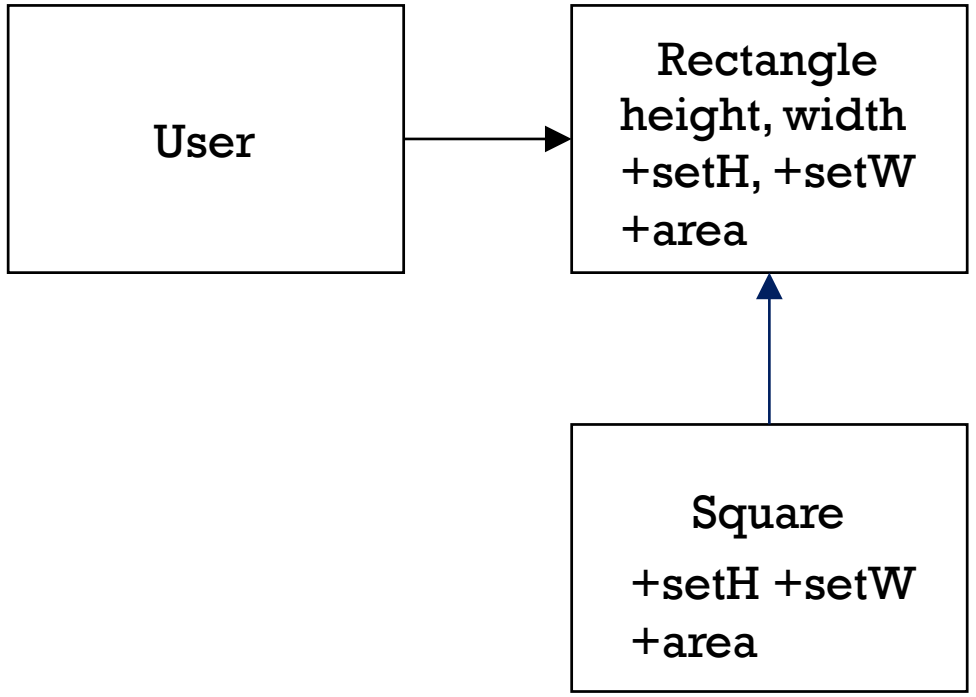
Covariant: **wide to narrow**. That is, the return type of D.getFoo( ) is a B, which is a narrower, more specialized version of A, which is returned by C.getFoo( )

Contravariant argument type example

- Suppose we write a class that models an AnimalShelter
  - member function called adopt( ) that returns an Animal
  - member function called add_animal( ) that accepts an Animal at the shelter
- What happens if we create a subtype of AnimalShelter called CatShelter?
- A language like C++ does **NOT** allow contravariant argument widening:
  - We cannot override add_animal( ) in CatShelter so that it accepts an instance of the Animal's superclass (LivingEntity)
  - C++ interprets that as an overloaded function
  - The argument type is **INVARIANT** in C++

LiskovExample.cpp

Canonical example



- Fails Liskov Substitution Principle
- Square is not a proper subtype of Rectangle
- The height and width of rectangle are independently mutable
- In contrast, the height and width of square must change together
- The user will get confused because they will think they are dealing with a rectangle

```
//passing in square
void area(Rectangle &r)
{
  r.setW(5); //sets height & width 5
  r.setH(2); //sets height & width 2
  Assert(r.area() == 10); // Error area = 4
}
```

RectangleSquare.cpp

OPEN-CLOSED PRINCIPLE

Open-closed principle

"You can either have software quality, or you can have pointer arithmetic, but you cannot have both"

Bertrand Meyer, French academic

- software entities (classes, modules, functions, etc.) should be open for extension but closed for modification
- The behavior of a software entity should be extendible, without having to modify its source code

How do we do this?

- We accomplish this by:
  1. Partitioning our software system into components
  2. Arranging the components into a dependency hierarchy that protects higher-level components from changes in lower level components
  3. Avoiding modifying source code in classes that are already in use
  4. Using abstract base classes in inheritance hierarchies as interfaces
- Our goal is (as always) to make a system easy to extend without incurrent a high impact of change

So when is a module open or closed?

- **A module is open if** it is still available for extension if it is possible to:
  - Add fields to the data structures it contains
  - Add new elements to the set of functions it performs.
- **A module is closed if** it is available for use by other modules
- A closed module is one that may be compiled and stored in a library, for others to use
- Closing a module simply means having it approved by management, adding it to the project's official repository of accepted software items (often called the project _baseline_), and publishing its interface for the benefit of other module designers

Software changes

- We don’t want to modify code that’s in use
- But sometimes we have to make changes
- We **should extend the original object**:
  - Add facets of behaviours
  - *Leave original source code untouched*.
- Sounds like polymorphism doesn’t it!
- Challenges:
  - We have to anticipate changes
  - Designer must be aware of possible variations in current functionality, consider a technical interface that will scale.

Example

- Imagine we're asked to write code to calculate area of a rectangle

```
struct Rectangle {
    double width;
    double height;
};
```

```
double area(Rectangle *r) {
    return r->width * r->height;
}
```

Example

- Asked to now find area of rectangle and circle

```
struct Shape {
};

struct Rectangle : Shape {
   double width;
   double height;
};

struct Circle : Shape {
   double radius;
}
```

```
double area(Shape *s) {
   if(typeid(s).name() == typeid(Rectangle).name())
   {
      Rectangle *r = (Rectangle*)s;
      return r->width * r->height;
   }
   else
   {
      Circle *c = (Circle*)s;
      return c->radius * c->radius * 3.14;
   }
}
```

Example

- Asked to now find area of rectangle and circle, and triangle…

```
struct Shape {
};

struct Rectangle : Shape {
   double width;
   double height;
};

struct Circle : Shape {
   double radius;
}

struct Triangle : Shape {
//triangle code
```

```
double area(Shape *s) {
   if(typeid(s).name() == typeid(Rectangle).name())
   {
      Rectangle *r = (Rectangle*)s;
      return s->width * s->height;
   }
   else if (typeid(s).name() == typeid(Circle).name())
   {
      Circle *c = (Circle*)s;
      return c->radius * c->radius * 3.14;
   }
   else …
   //calculate triangle area
```

Example - Solution

- Abstract out shape and have subtypes have their own area functions

```
struct Shape {
   virtual double area() = 0;
};

struct Rectangle : public Shape{
   double width;
   double height;
   double area() override () {return width * height;}
};

struct Circle : public Shape{
   double radius;
   double area() override () {return radius * radius *
3.14;}
};
```

```
double area(Shape *s) {
   return s.area();
}
```

Example - Solution

- Abstract out shape and have subtypes have their own area functions
- With this new approach we can add as many shapes as we like
  - OPEN for extension – add new shapes without modifying Shape or area function
  - Shape and area functions are considered CLOSED