COMP 3522

Object Oriented Programming in C++

Week 5

Agenda

1. Exceptions
2. Design idioms
3. Coupling and cohesion

COMP 3522

EXCEPTIONS

Typical Error Situations

- Implementing a class or method **incorrectly**
- Failing to meet the specification
- Making an **inappropriate** object request
  - invalid index
- Generating an **inconsistent** or inappropriate object state
  - arising through class extension

Not ALWAYS Programmer Error*

- Errors often arise from the environment:
  - Incorrect URL entered
  - Network interruption
- File processing is particular error-prone:
  - Missing files
  - Lack of appropriate permissions

* ^^but ^^it ^^usually ^^is

Typical Java exceptions we've seen

Exception	Purpose
NullPointerException	When an application attempts to use an object reference that is set to null
ArrayIndexOutOfBoundsException	Indicate that an index is out of range
ClassCastException	Indicate that the code has attempted to cast an object to a subclass of which it is not an instance
ConcurrentModificationException	Indicate concurrent modification of an object when such modification is not permissible

Dealing with unexpected behaviour in C++

- Two principle approaches:

1. **Assertions** are for detecting programming errors
2. **Exceptions** for situations that prevent proper continuation of the program (errors that cannot be handled locally)

Assertions

- The macro assert from header <cassert> is inherited from C
- Evaluates an expression, immediately terminates the program if false
- Easy to turn off by defining NDEBUG before including <cassert>

```
#define NDEBUG // Turns off assertions
#include <cassert>
```

Assertion example

```
#include <cassert>
// Compute square root of non-negative number
double square_root(double x)
{
  check_somehow(x >= 0);
  … // Perform our calculation
  assert(result >= 0.0); // Should be positive
  return result;
}
```

C error codes

In C, programmers used to return error codes (like main still does)

```
int read_matrix_file(const char* fname)
{
    fstream f(fname);
    if (!f.is_open()) { return 1; }
    …
    return 0;
}
```

Problem # 1: we can ignore the error code

C error codes

More problems:

1. We can't return our computational results
2. We have to return a success/error code
3. We are forced to pass references as arguments
4. This can prevent us from building expressions with the results

Enter the exception

```
int read_matrix_file(cont char* fname)
{
    fstream f(fname);
    if (!f.is_open()) { throw “Can’t open file”; }
    …
    return 0;
}
```

C++ exceptions

- C++ lets us **throw** anything as an **exception**:

    1. Strings
    2. Numbers
    3. User types
    4. Exceptions from the standard library.

- **It is best, however, to define exception types or use exceptions from the standard library.**

Refined exception example

```
struct cannot_open_file { … };

int read_matrix_file(cont char* fname)
{
    fstream f(fname);
    if (!f.is_open()) {throw cannot_open_file(); }
    …
    return 0;
}
```

Reacting to an exception

- We must catch exceptions (just like Java)
- We use a try-catch block:

```
try
{
      … //code that throws exception
}
catch (e1_type1& e1)
{
      //handle the exception
}
```

EXCEPTIONS GUIDELINES

Some guidelines

1. Catch exceptions by **reference**
   - Captures exceptions that are derived from the reference type
2. When an exception is thrown, the **first catch-block** with a matching type is executed

```
try
{
      … //code throws e1_type1 or e2_type2 exception
}
catch (e1_type1& e1) { //handle the exception }
catch (e1_type2& e2) { //handle the exception }
```

Some guidelines

3. **Further catch-blocks of the same type or child-types are ignored**

```
try
{
      … //code throws e1_type1 exception
}
catch (e1_type1& e1) {} //catches e1_type1
catch (e1_type1& e2) {} //IGNORED - same type as previous
```

Some guidelines

3. Further catch-blocks of the same type or child-types are ignored

```
try
{
      … //code throws child_type exception.
}
catch (child_type& e1) {} //order matters. Child type first
catch (parent_type& e2) {} //sub type exception caught here
```

Some guidelines

4. A catch-block with an **ellipsis**, i.e., **three dots**, catches all exceptions
   - Obviously this should be the last one

```
try
{
      … //code throws e1_type1, e1_type2, and other exceptions
}
catch (e1_type1& e1) {}
catch (e1_type2& e2) {}
catch (...) { // This catches EVERYTHING }
```

Some guidelines

5. If nothing else, consider capturing the exception, providing an informative error message, and terminating the program:

```
try {
    int result = read_matrix_file(“No file”);
} catch (cannot_open_file& e) {
    cerr << “FILE NOT FOUND.  TERMINATING…\n”;
    exit(EXIT_FAILURE); // <cstdlib>
}
```

Some guidelines

6. Alternatively, we can continue after the error message or after implementing some sort of rescue, by **rethrowing** the exception

```
try {
    int result = read_matrix_file(“No file”);
} catch (cannot_open_file& e) {
    cerr << “FILE NOT FOUND.\n”;
    …
    throw; // Rethrows cannot_open_file exception
}
```

Noexcept qualification for functions

- C++03 allowed us to specify which types of exceptions can be thrown from a function (like Java)
- Was very quickly deprecated (don't do this)

- So what should we do?
- **C++11 added a new qualification for specifying that no exceptions must be thrown out of a function**

```
double square_root(double x) noexcept { … }
```

Noexcept qualification for functions

- Benefits:
  1. Calling code never needs to check for thrown exceptions from square_root
  2. If an exception is somehow thrown despite the qualification, the program ends (which is what should happen).

- Destructors are implicitly declared noexcept
  1. **NEVER throw an exception from a destructor**
  2. If you do, it will be treated as a run-time error and execution will end!

Standard exceptions

- <exception> header
- **std::exception** is a base class designed to be derived
- All exceptions thrown by members of the standard library are derived from this class
- Contains a **virtual** member function called **what** that returns a null-terminated char sequence (char *)
- Override this to deliver a meaningful exception message:

```
struct myexception: public exception {
  const char* what() const noexcept override
  {    return "My exception happened";  }
}
```

exception.cpp, exception2.cpp

Derived from std::exception

Exception	Description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_function_call	thrown on a bad call
bad_typeid	thrown by typeid
logic_error	thrown when a logic error occurs
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr

These exceptions are actually useful

<**stdexcept**> defines two exception types that can be inherited by custom exceptions to report errors:

1. logic_error:
    1. invalid_argument
    2. length_error //exception3.cpp
    3. out_of_range.
2. runtime_error:
    1. range_error
    2. overflow_error.

std::invalid_argument example

```
class Name
{
private:
  std::string first;
public:
  Name(std::string first) : first(first)
  {
    if (first.length() == 0)
    {
      throw std::invalid_argument("No first name!");
    }…
```

//exception4.cpp

What if we don't catch an exception?

- If an exception is not caught by any catch statement because there is no catch statement with a matching type, the special function **terminate** will be called.
- std::terminate is in <exception>
- Calls the termination handler
- The termination handler calls abort
- **CRASH AND BURN**

//exception5.cpp

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

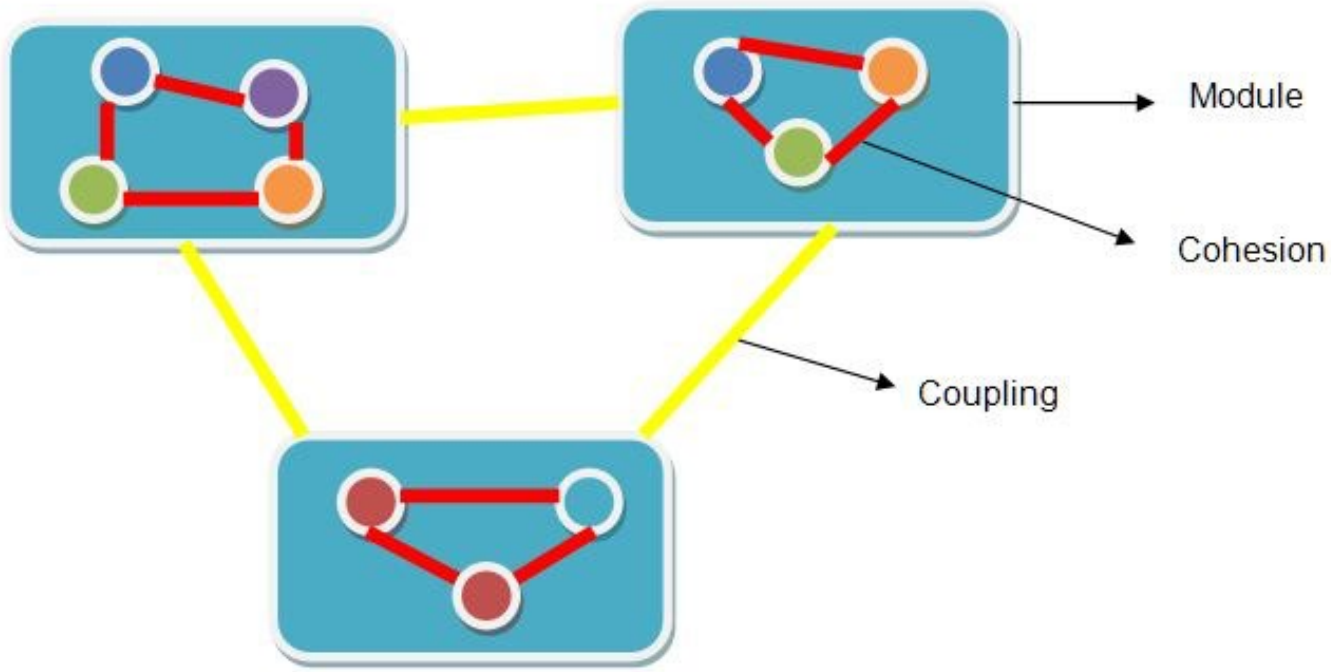
- Decompose large-scale problem statements into modules and components
  - **Independent** developability
  - **Independent** deployability
- Plugin architecture
  - Modules that contain high-level policies are **independent** of modules that contain low-level details
- **To achieve this goal we have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

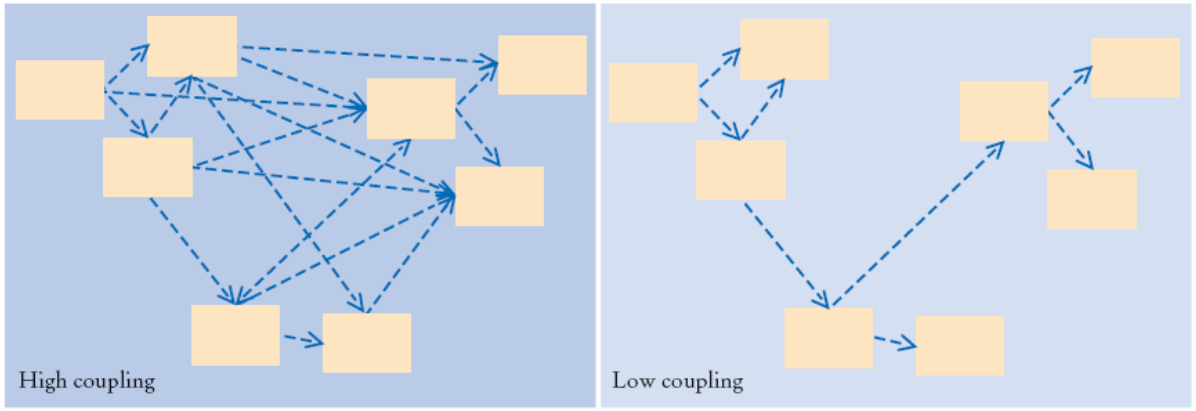
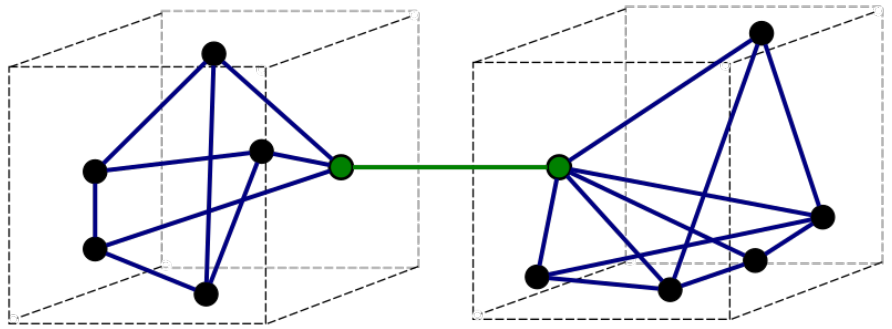


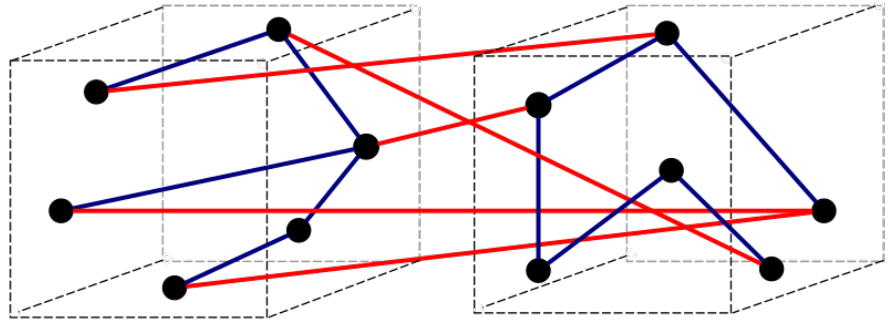
Figure 2 High and Low Coupling Between Classes

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



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

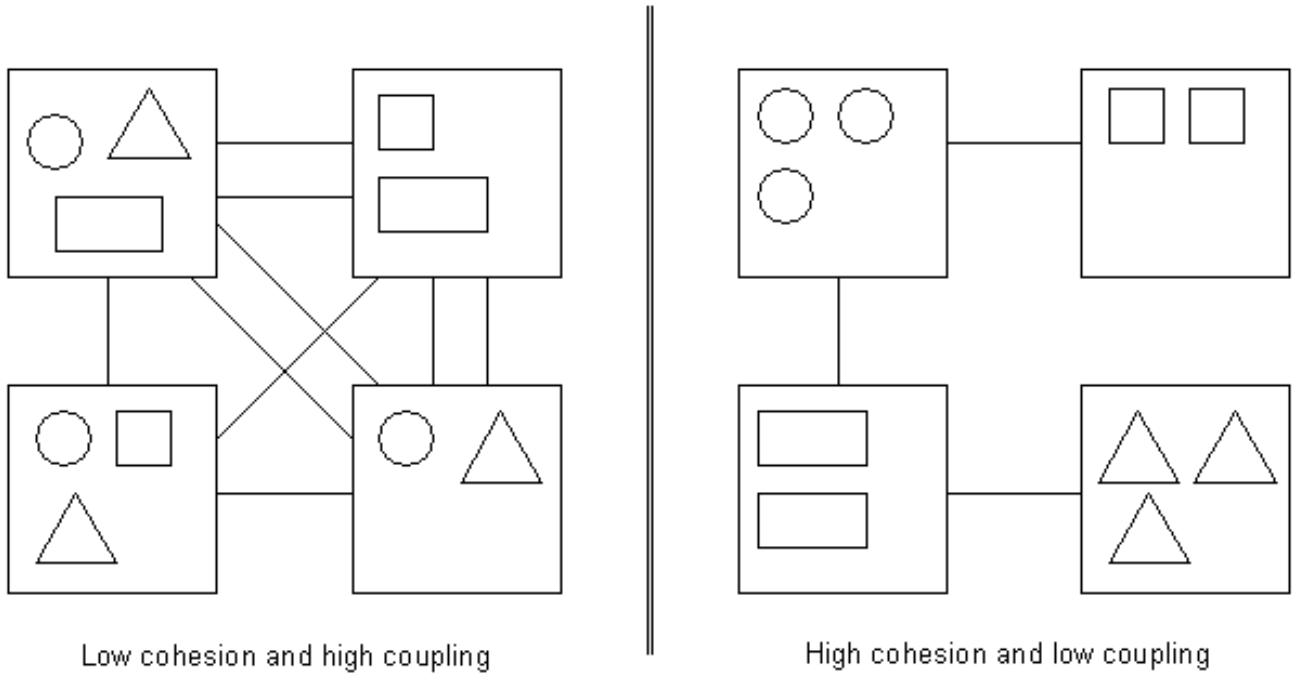
Cohesion

- **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 examples

Class level:

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

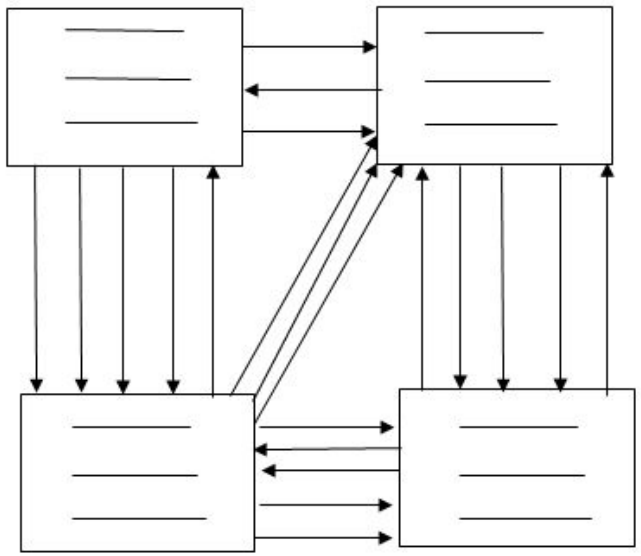
Method level:

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

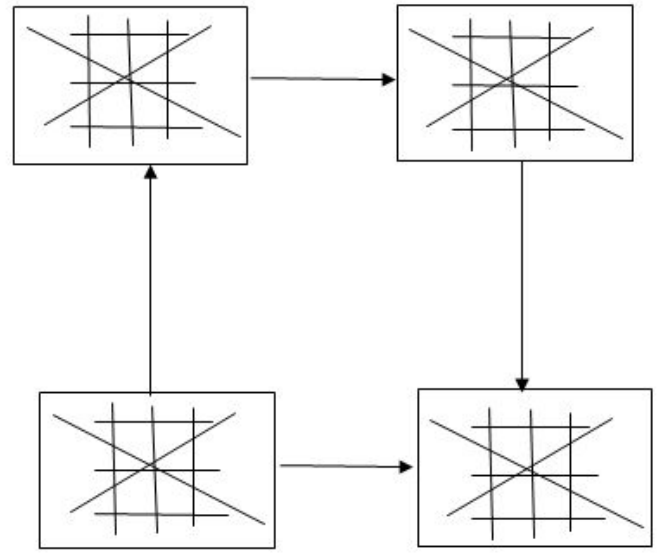
- We **avoid** loosely cohesive classes and methods
  - Methods perform multiple tasks # BAD
  - Classes have no clear identity # BAD

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?

COUPLING & COHESION EXAMPLES

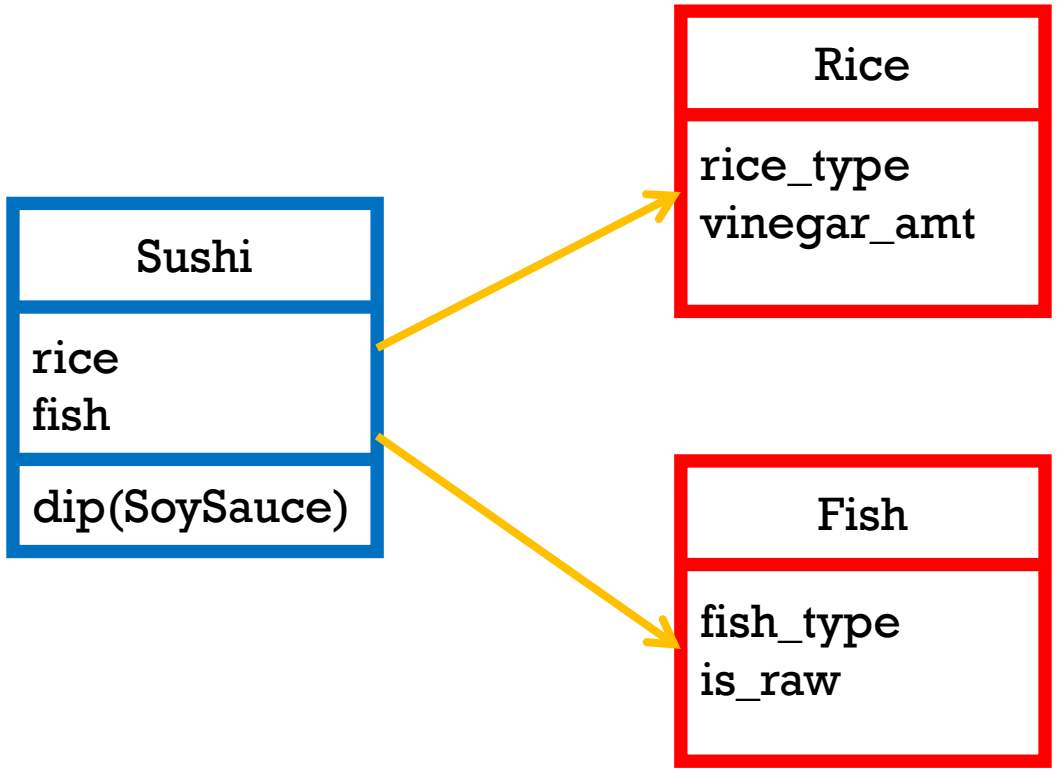
Dependencies

- When one entity depends on another entity.
- If entity A **uses** entity B, then A is said to be **dependent** on B.

- In this context an entity could be a Library, **Class**, or Sub-system of related code

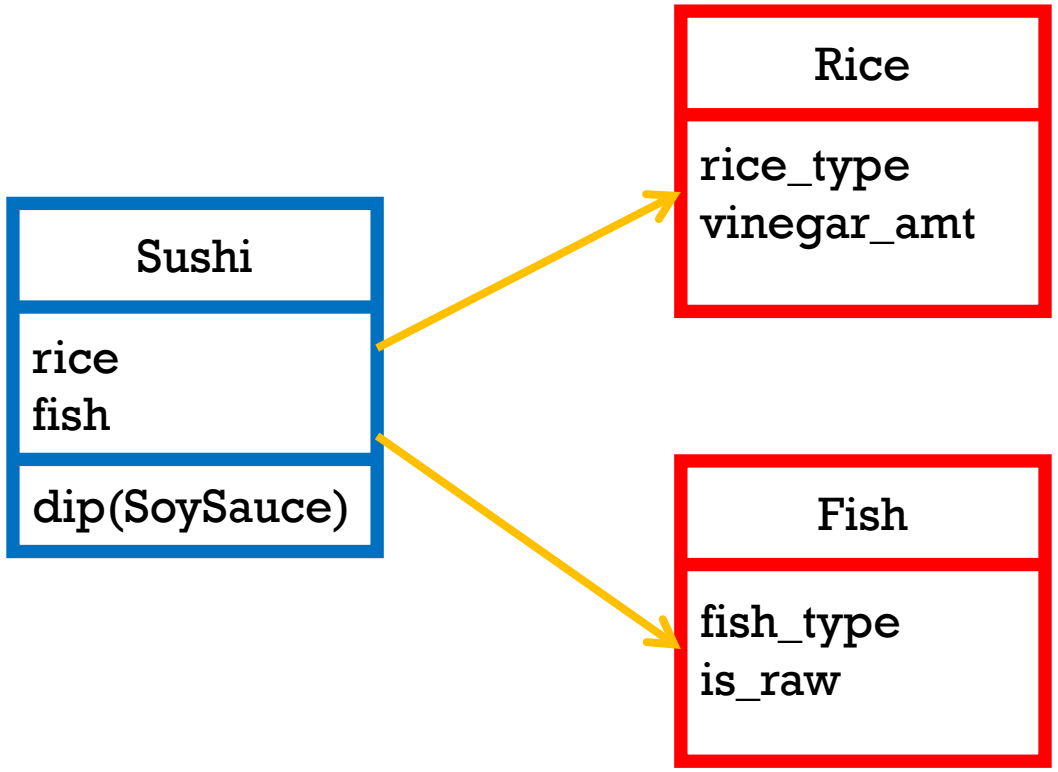
For example, say we were simulating a sushi restaurant. **Class Sushi would be dependent on Class Rice and Class Fish**

Dependencies



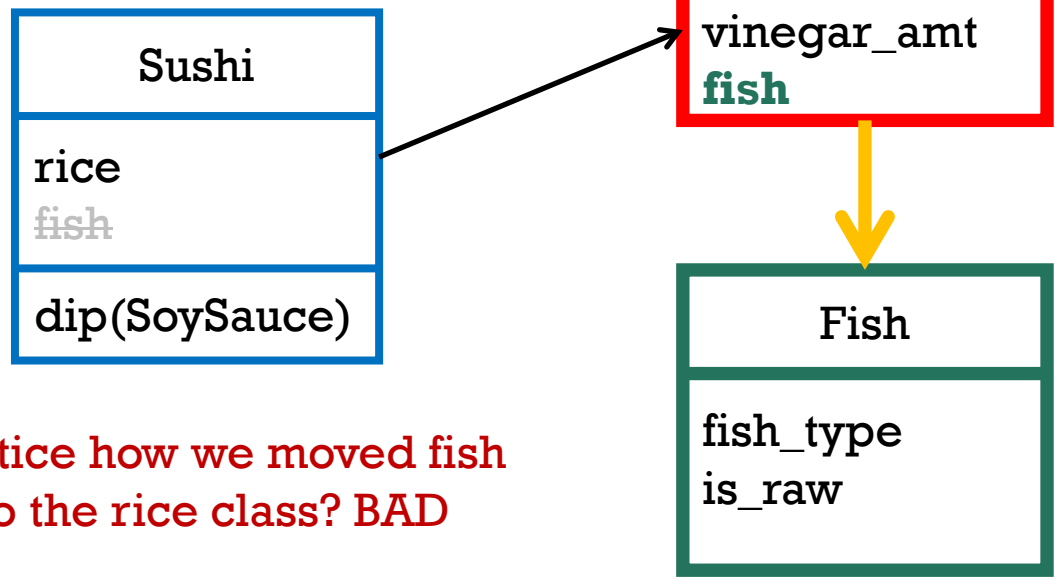
- If entity A is strongly dependent on entity B then any change in entity B would possibly require us to change entity A as well.
- This may not sound like much.
- Imagine if A and B were sub-systems or packages in a larger application. Now this is an issue.

Dependencies



- If entity A is strongly dependent with entity B then entity A will not be able to function without entity B.
- **Tight coupling** is a form of **strong dependency**.
  - The **dependency is so tight**, the system becomes **more rigid and less flexible**
- We try to avoid tight coupling where possible

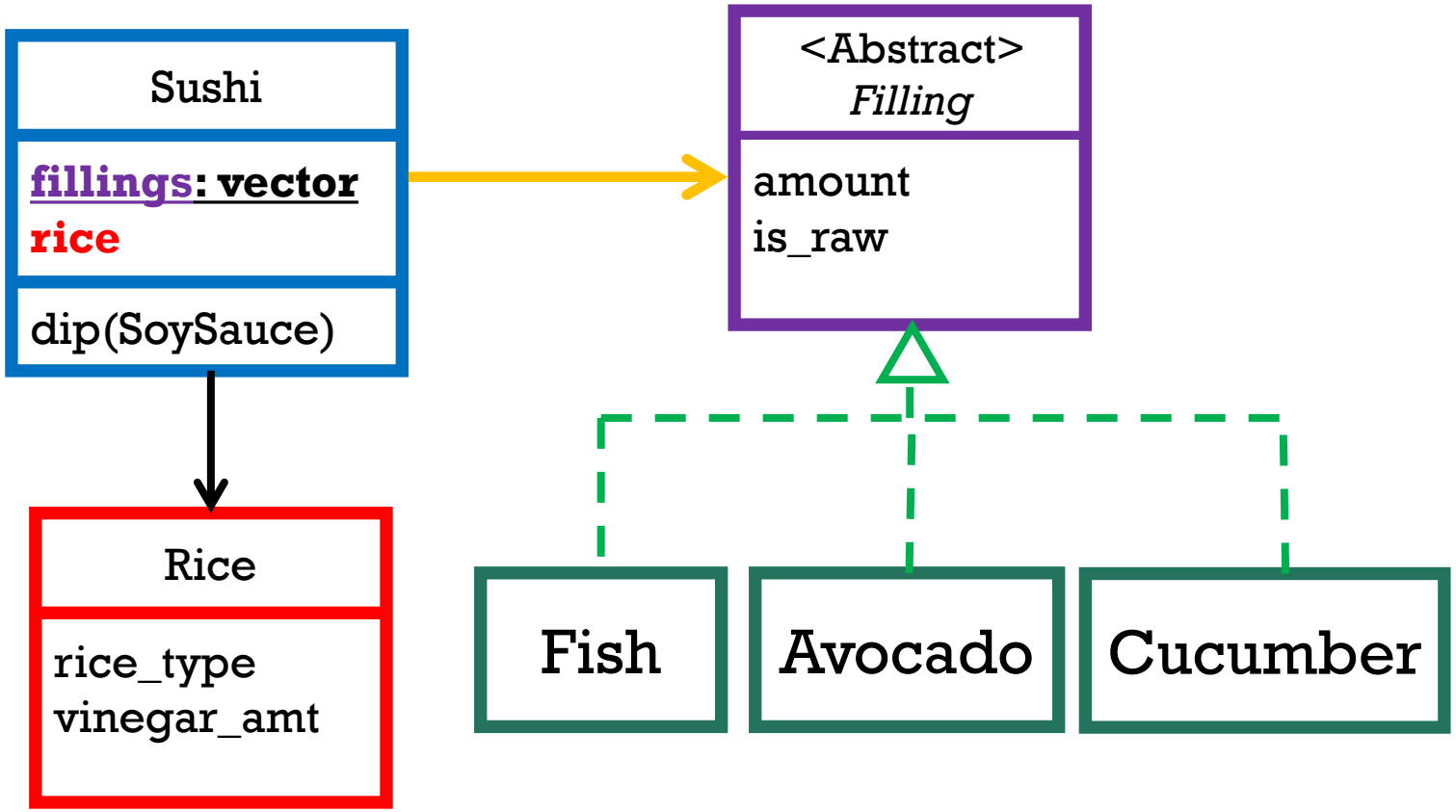
Tight coupling



Notice how we moved fish into the rice class? BAD

- **Say fish and rice are tightly coupled.** This is bad because **rice** will always come with **fish**.
- We wouldn't be able to have other kinds of sushi without fish.
  - *(What about the Tamago!! :O)*
- We don't want tight coupling like these. We want to **decouple our code** if possible so it can be flexible and polymorphic.

Decoupling sushi – Inheritance!



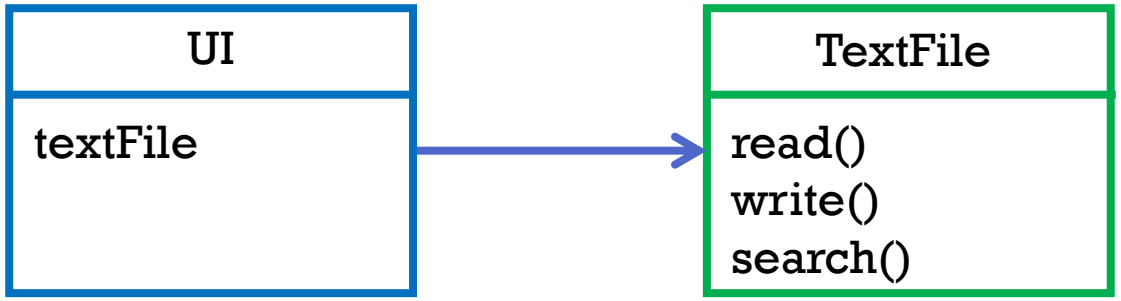
Inheritance and Interfaces/Abstract classes are the most common and **effective way of decoupling code**.

NOTE: We still have a **dependency**!

Entities will always be dependent on each other, that's how OOP works

It's better to have **weak dependencies** like these

A more realistic example



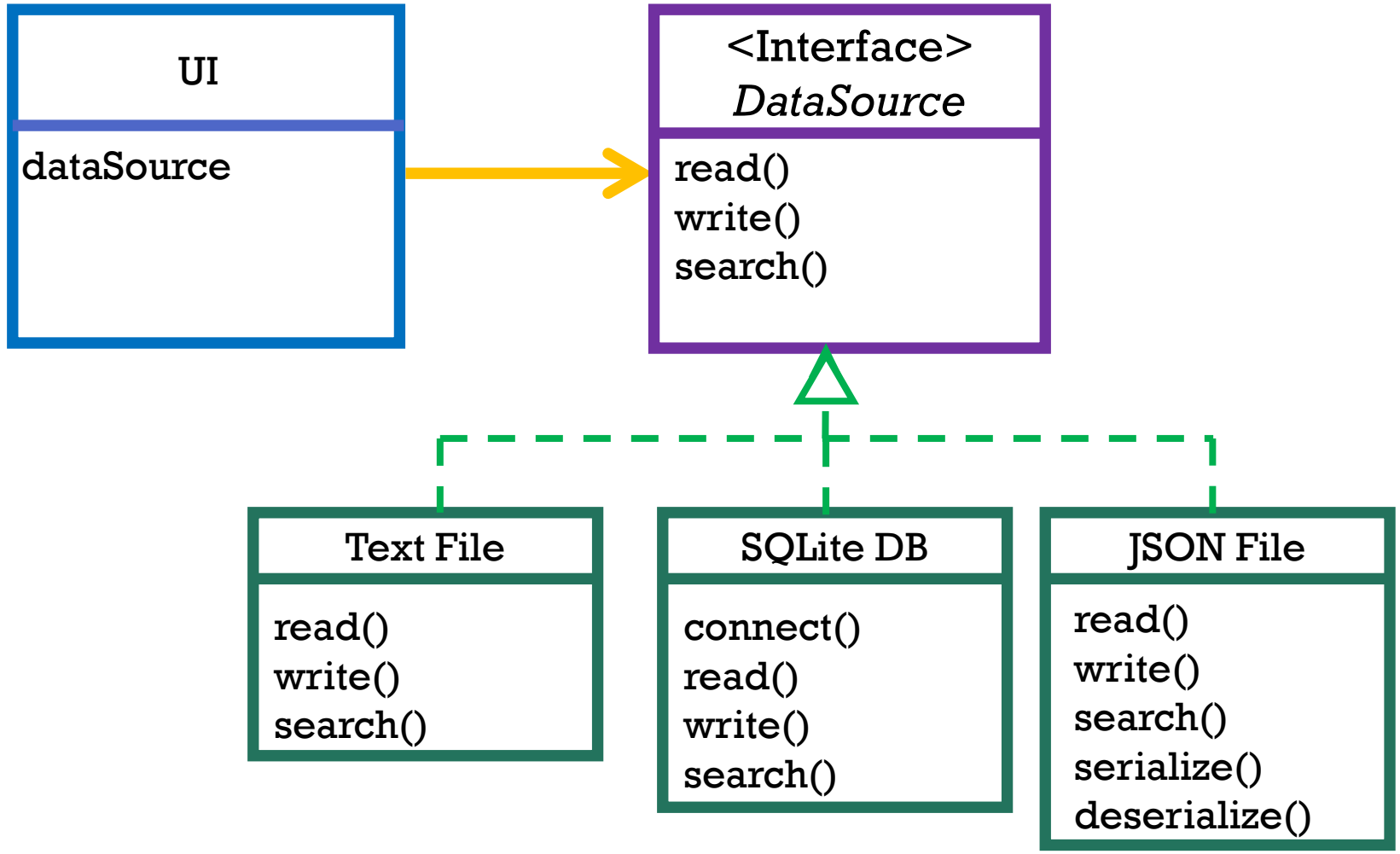
Say we were developing an app with a UI that was populated with some data from a text file.

After a few releases and years of development, for some reason we decide to switch out to a more secure source of data, perhaps an encrypted **JSON** file or even perhaps a **SQLite** Database. We would have to edit all the modules/classes that dealt with our app's UI!

A more realistic example

We can decouple our system to instead depend on a **data source** which is an **abstraction** that **hides** different **data sources**.

Our **UI** would just have to be **dependent on a common interface** provided by the **Data Source** and not be concerned with how that data source is implemented



Agenda

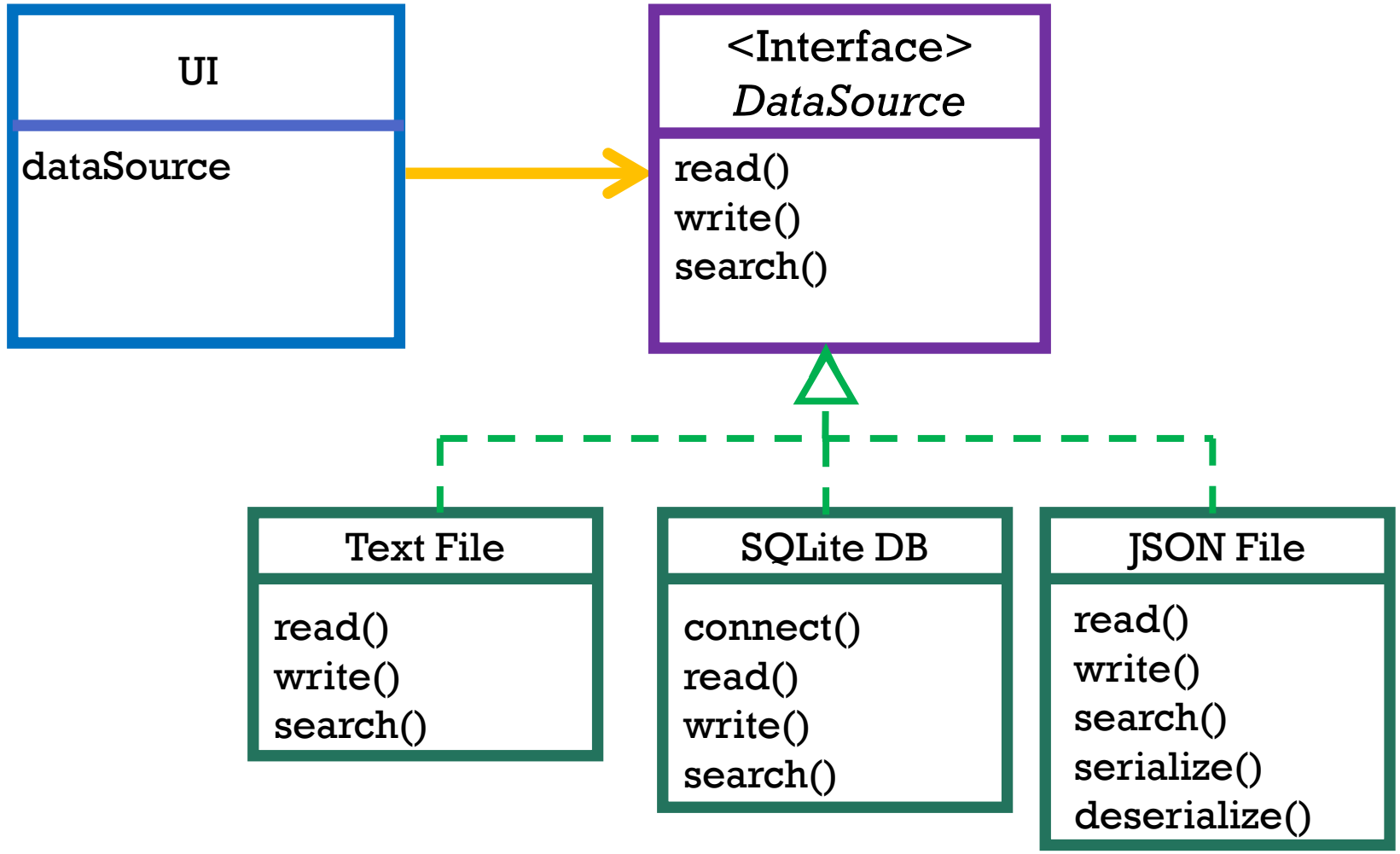
1. SOLID design principles
2. Law of Demeter

COMP 3522

Recap: Decoupling dependencies

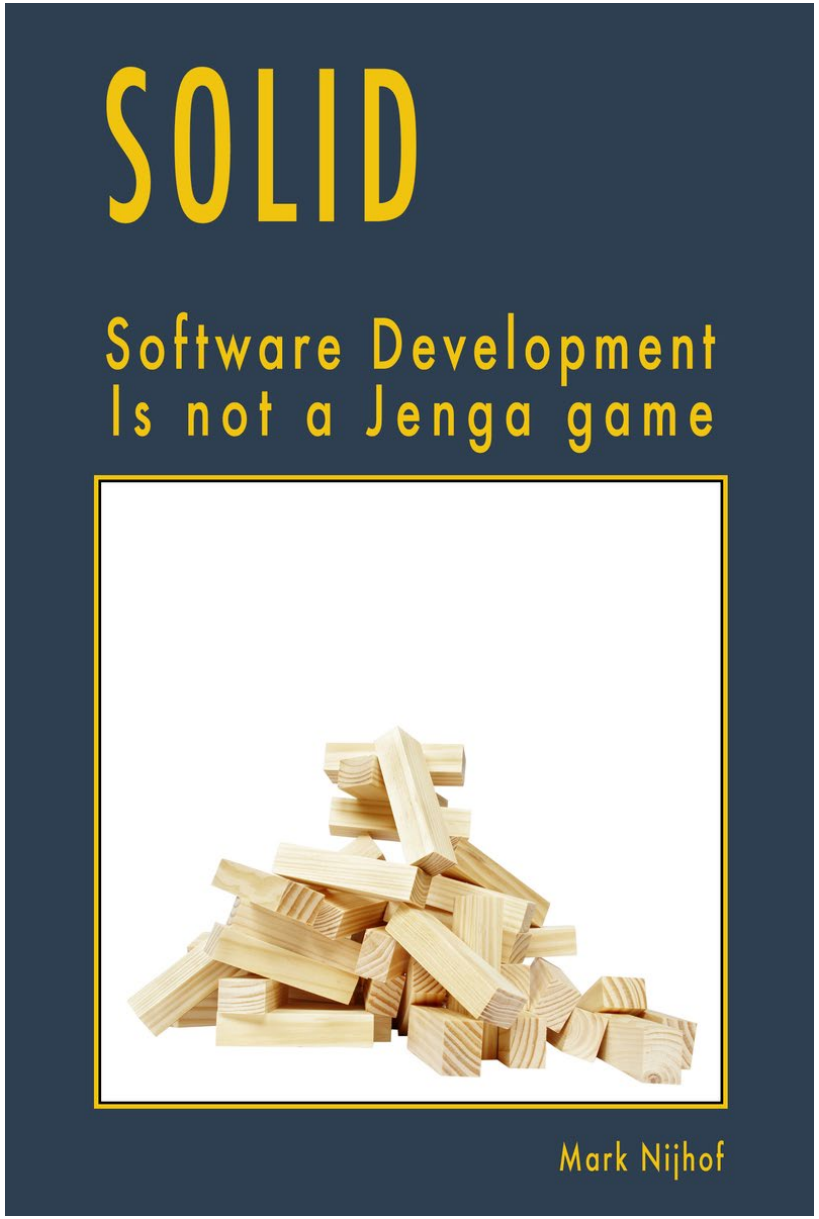
We actually used at least 3 **SOLID Design Principles** here!

Today we're going to learn what these principles are and why they are so important.



SOLID Design Principles

And the Law of Demeter.



SOLID Design Principles

- A set of **guidelines** to help us write **maintainable**, **decoupled**, **flexible** Object Oriented Code
- Created by Robert C. Martin, also known as Uncle Bob amongst developers.
- Highly respected and has a bunch of books on writing clean code.

Single Resposibility Principle
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

Open / Closed Principle
A software module (it can be a class or method) should be open for extension but closed for modification.

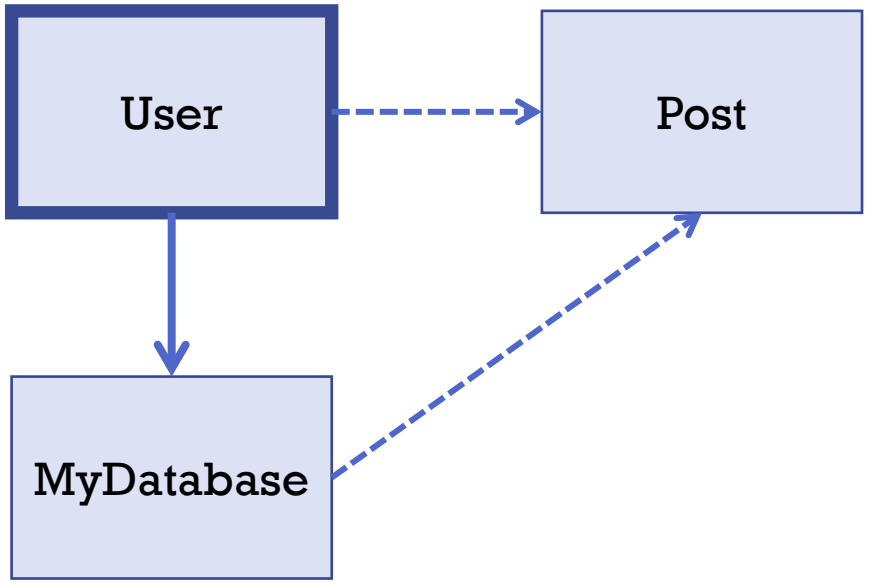
Liskov Substitution Principle
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Interface Segregation Principle
Clients should not be forced to depend upon the interfaces that they do not use.

Dependency Inversion Principle
Program to an interface, not to an implementation.

SINGLE RESPONSIBILITY PRINCIPLE

Say we had a system where a user could write and create forum posts

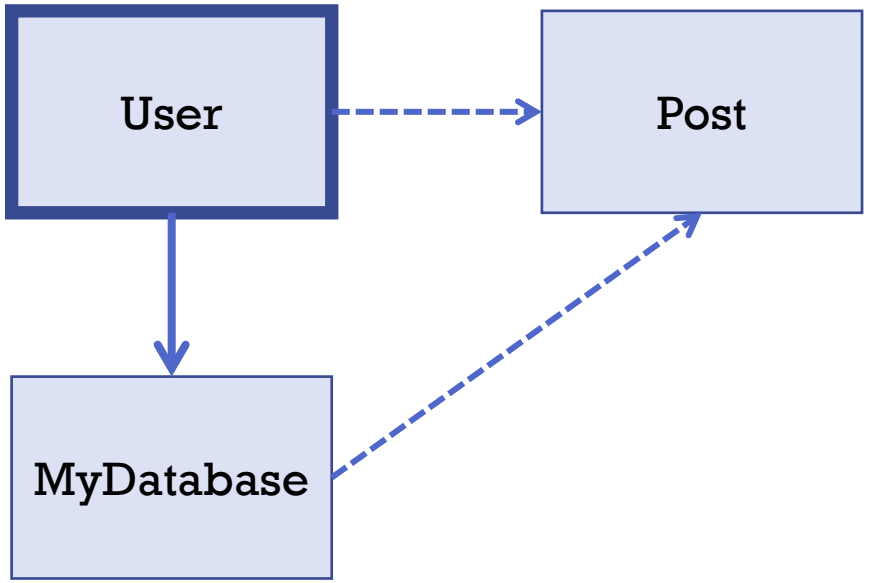



```
class User {
…
User(string username, string password, string name, string email)
{
        this->username = username;
        this->password = password;
        this->name = name;
        this->email = email;
        this->database = MyDatabase();
    }

    void post_to_forum(string message, string subject)
    {
        Post post = Post(subject, message, username);
        database.add_simple_post(post);
    }
};
```

Why is this code bad?

- The User class is very coupled and dependent. It carries out many **different roles and responsibilities**.
- Changing any part of this system will probably cause a change in the User class as well.

Say we had a system where a user could write and create forum posts



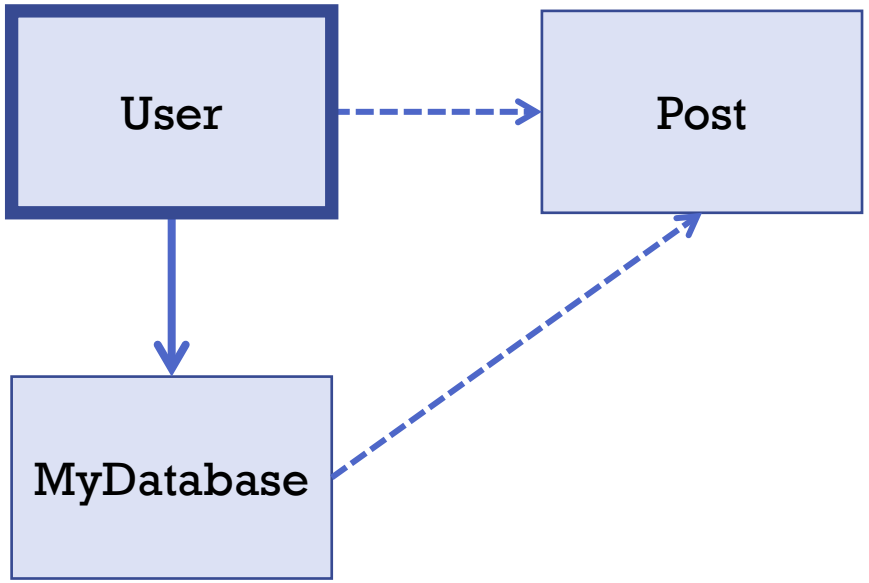

```
class User {
…
User(string username, string password, string name, string email)
{
        this->username = username;
        this->password = password;
        this->name = name;
        this->email = email;
        this->database = MyDatabase();
    }

    void post_to_forum(string message, string subject)
    {
        Post post = Post(subject, message, username);
        database.add_simple_post(post);
    }
};
```

Why is this code bad?

User class has code to:

- **be a user**

Say we had a system where a user could write and create forum posts



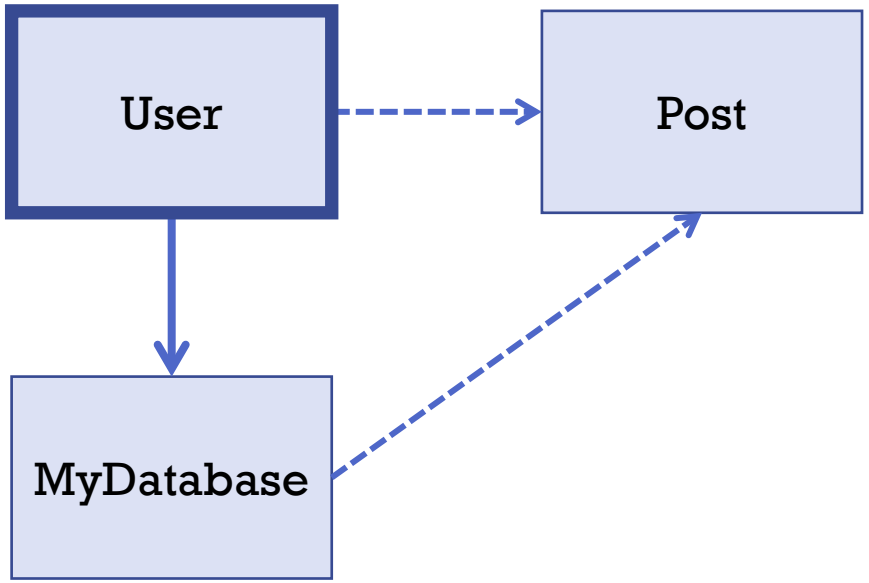

```
class User {
…
User(string username, string password, string name, string email)
{
        this->username = username;
        this->password = password;
        this->name = name;
        this->email = email;
        this->database = MyDatabase();
    }

    void post_to_forum(string message, string subject)
    {
        Post post = Post(subject, message, username);
        database.add_simple_post(post);
    }
};
```

Why is this code bad?

User class has code to:

- be a user
- **create a database**

Say we had a system where a user could write and create forum posts



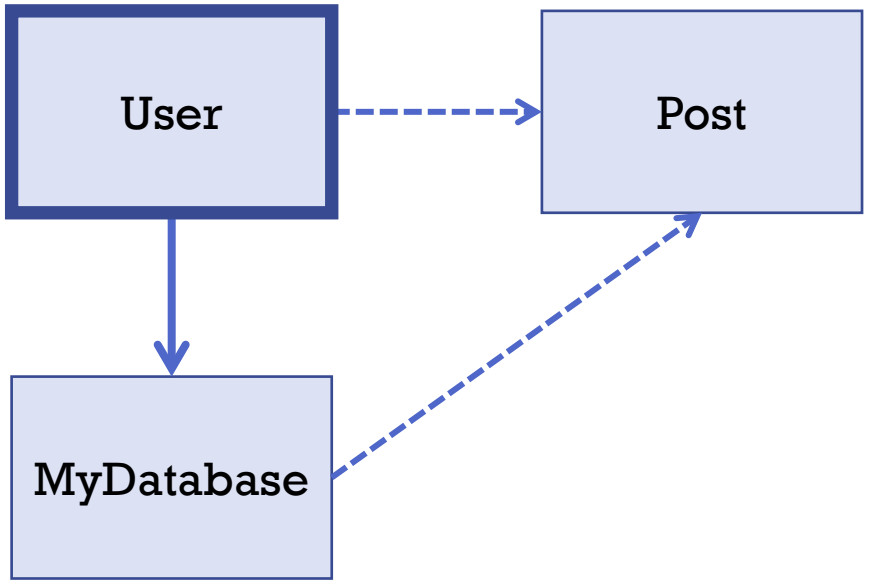

```
class User {
…
User(string username, string password, string name, string email)
{
        this->username = username;
        this->password = password;
        this->name = name;
        this->email = email;
        this->database = MyDatabase();
    }

    void post_to_forum(string message, string subject)
    {
        Post post = Post(subject, message, username);
        database.add_simple_post(post);
    }
};
```

Why is this code bad?

User class has code to:

- be a user
- create a database
- **post to a forum**

Say we had a system where a user could write and create forum posts



The **User class** should focus on only **being a user**

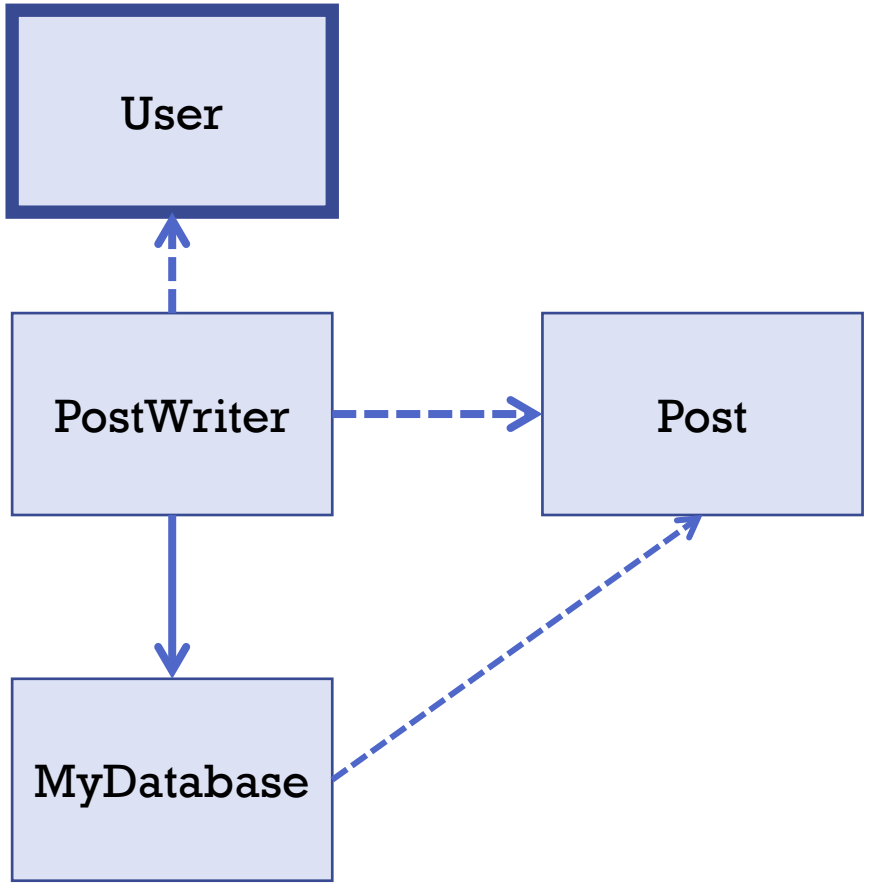
```
class User {
…
User(string username, string password, string name, string email)
{
        this->username = username;
        this->password = password;
        this->name = name;
        this->email = email;
        this->database = MyDatabase();
    }

    void post_to_forum(string message,string subject)
    {
        Post post = Post(subject, message, username);
        database.add_simple_post(post);
    }
};
```

Single Responsibility Principle

"The Single Responsibility Principle requires that each class is responsible for only one thing."

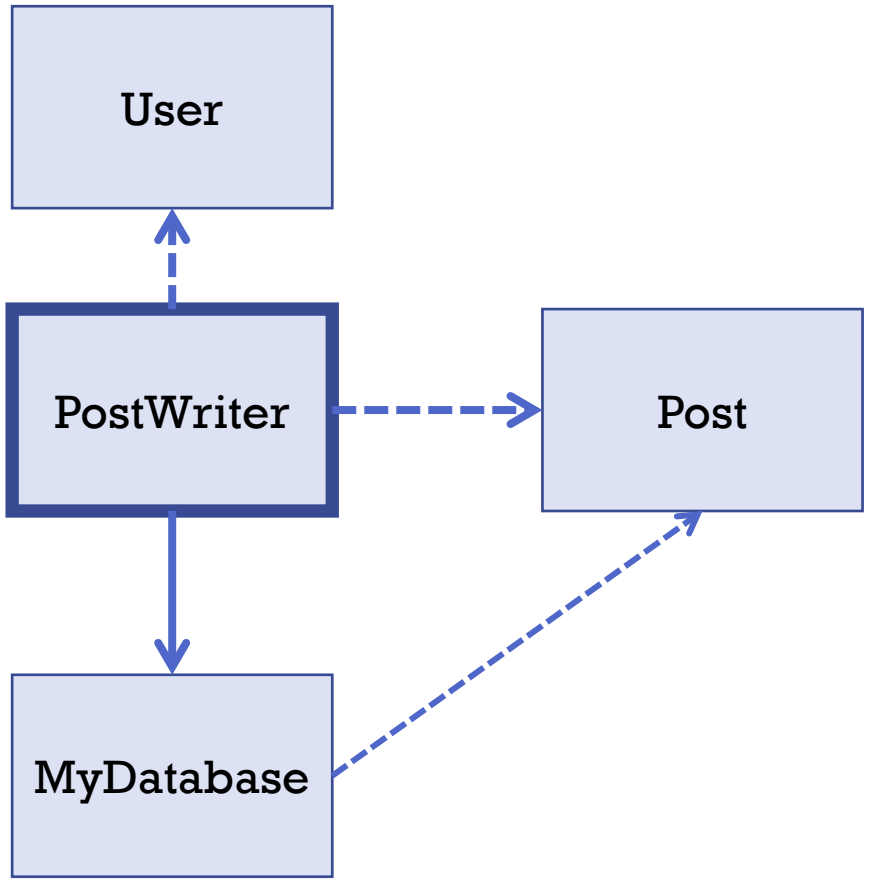
- Ask yourself, are there multiple reasons to change my class?
- There should only be one reason to change your class. Each class should have one and only one responsibility.

We Split the User Class! Pt.1




```
class User {
…
User(string username, string password, string name, string email)
{
        this->username = username;
        this->password = password;
        this->name = name;
        this->email = email;
}
```

We Split the User Class! Pt.2




```
class PostWriter {

    MyDatabase* database;

    PostWriter(MyDatabase* database)
    {
        this->database = database;
    }

    void post_to_forum(User& user, string message, string subject)
    {
        Post post = Post(subject, message, user.username);
        database->add_simple_post(post);
    }
};
```

OPEN-CLOSED PRINCIPLE

Now, Say we had a UI widget class….

Say we had a graphical user interface, where each UI element was a UIWidget.

```
class UIWidget {
    string type;
    void draw() {
        if (type == “Button”)
            //complicated button draw logic here
        else
            //complicated draw logic for a
            //generic UI box (Panel)
```

Now over time, what if our UIWidget class looked like this?

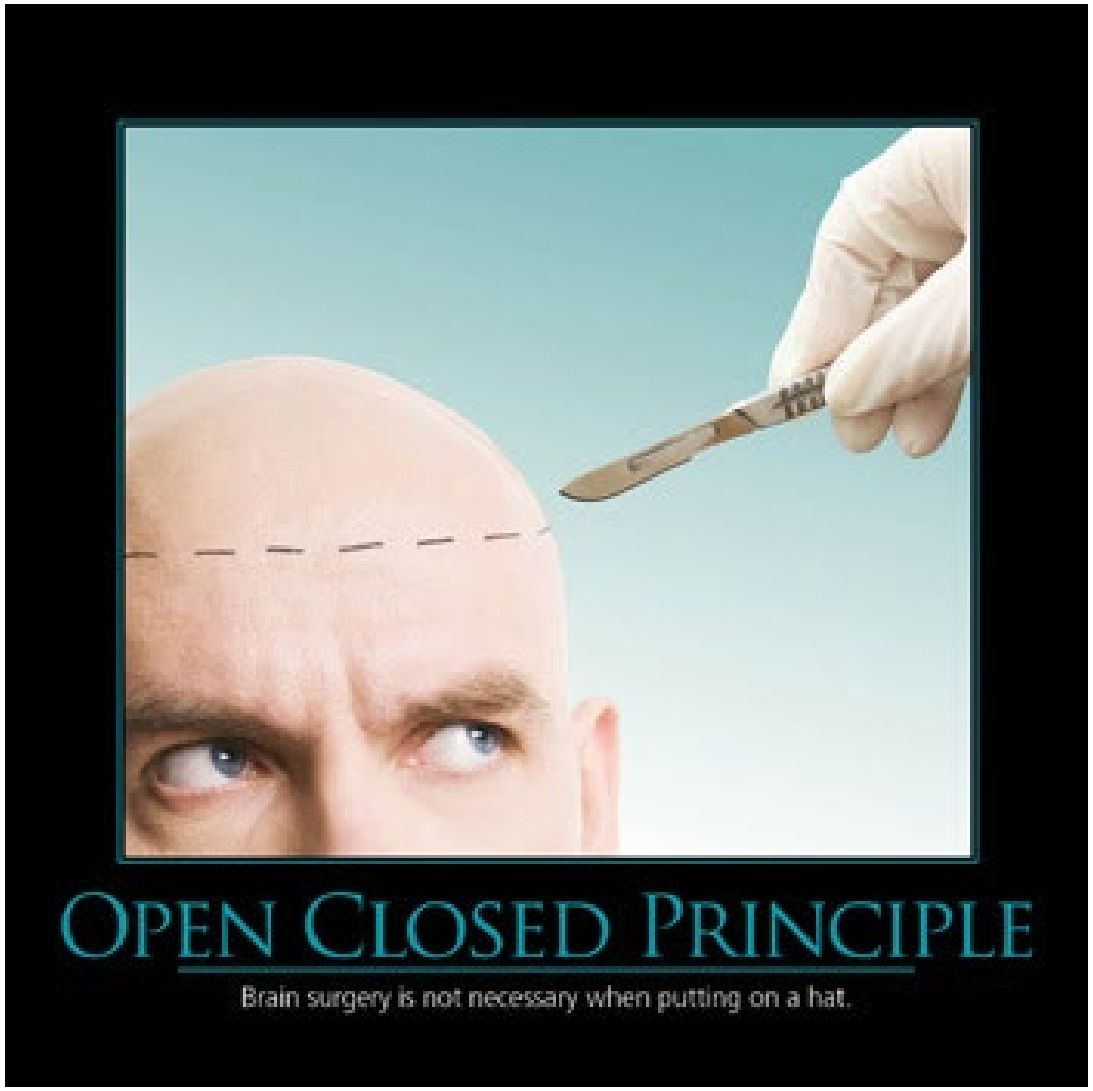
```
class UIWidget {
    string type;
    void draw() {
       if (type == "Button“)
            //complicated button draw logic here
       else if(type == “Text Box”)
            //complicated draw logic for a text box
       else if(type == “ScrollBar”)
            //complicated draw logic for a scroll bar
       else if(type == “Label”)
            //complicated draw logic for a label
       else
            //complicated draw logic for a panel
```

- This is terrible design! If we wanted to change the logic / algorithm behind drawing a button, we would need to edit this massive **draw member function.**
- An error or bug here would affect all the widgets on screen and crash our whole system!

Open Closed Principle

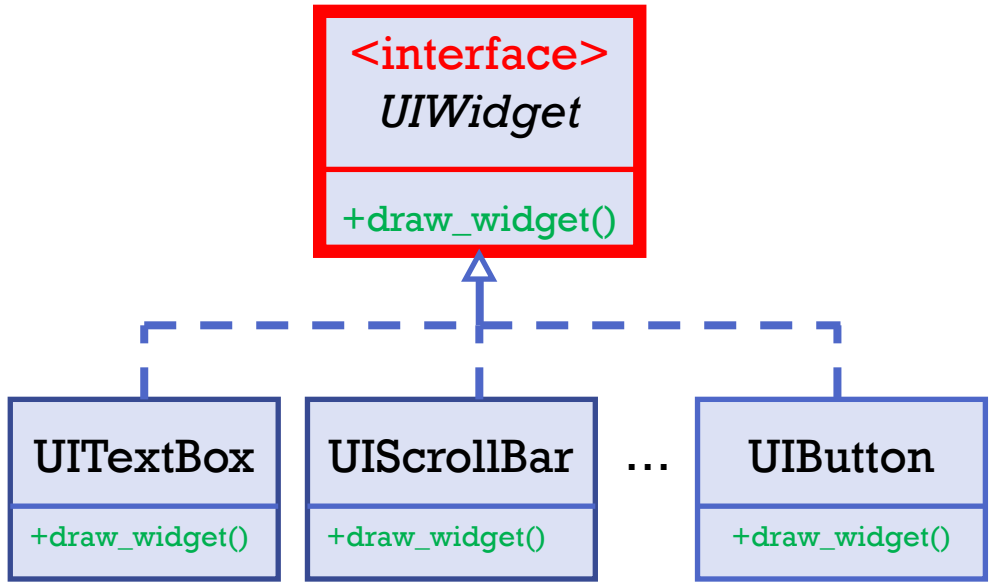
The Open/Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

- Once a class is made, its behaviour shouldn't be modified to support special cases. It should be **CLOSED for modification**.
- Instead, we extend (or inherit) from the class and override behaviours. That is, it's **OPEN for extension**.



We use Inheritance (and overriding)!

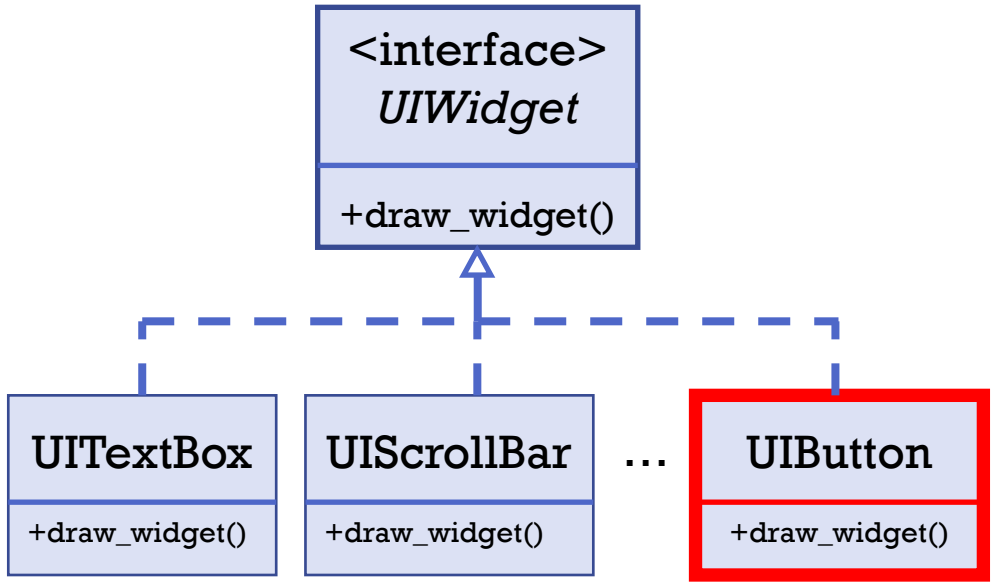
```
class UIWidget {
     virtual void draw() = 0;
}
class UIButton : public UIWidget {
    void draw() override {
        //complicated draw code for buttons goes here }
class UITextBox : public UIWidget {
    void draw() override {
        //complicated draw code for text box goes here }
class UIScrollBar : public UIWidget {
    void draw() override {
        //complicated draw code for scroll bar goes here }
```



We used **abstraction** to do this.
The **UIWidget class** is now an **interface**, ensuring that each UI component that inherits from it has the **same interface**.

We use Inheritance (and overriding)!

```
class UIWidget {
     virtual void draw() = 0;
}
class UIButton : public UIWidget {
    void draw() override {
        //complicated draw code for buttons goes here }
class UITextBox : public UIWidget {
    void draw() override {
        //complicated draw code for text box goes here }
class UIScrollBar : public UIWidget {
    void draw() override {
        //complicated draw code for scroll bar goes here }
```

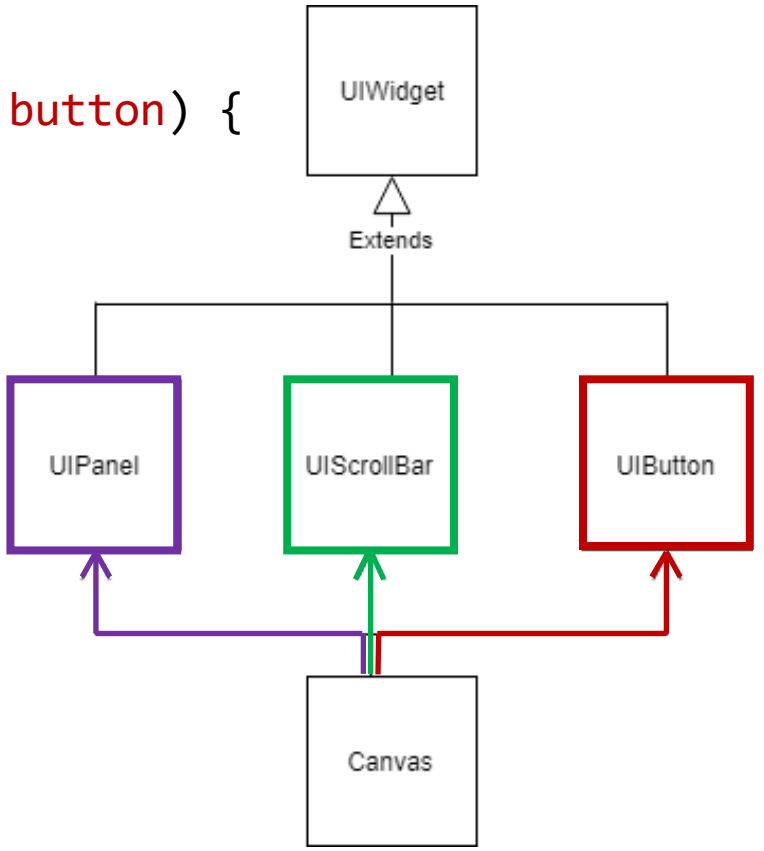


Now, any change to **UIButton** would not affect any other widget.
Each widget class is modular and decoupled. It even follows the Single Responsibility Principle!

LISKOV SUBSTITUTION PRINCIPLE

Now how do we draw all these widgets?

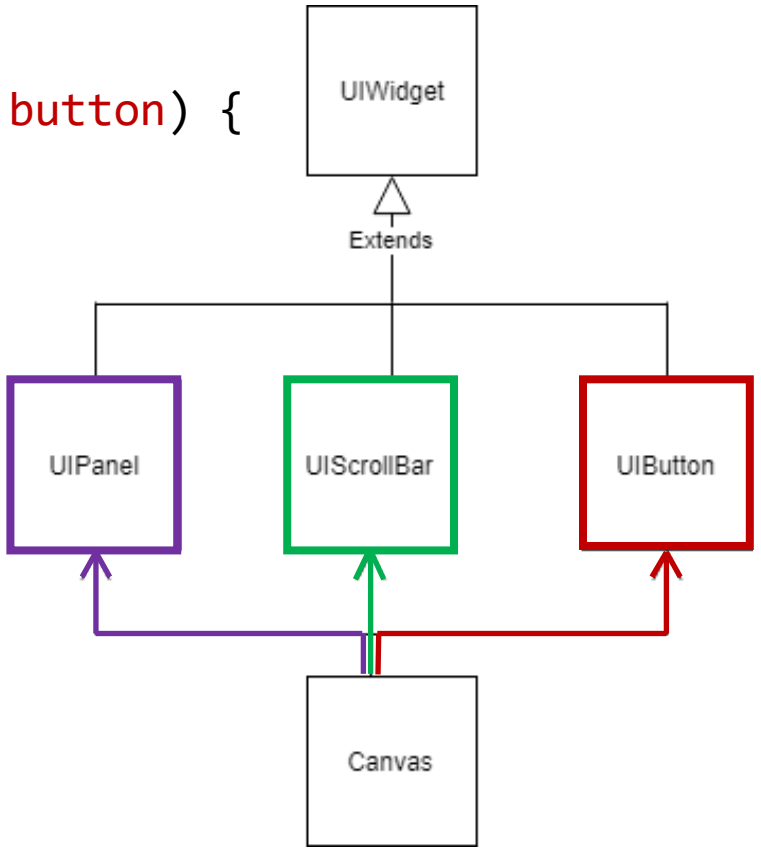
```
class Canvas:
    UIPanel* panel; UIScrollbar* scrollbar; UIButton* button;

    Canvas(UIPanel* panel, UIScrollbar* scrollbar, …, Button* button) {
        this->panel = panel
        this->scrollbar = scrollbar
        this->button = button
          …

    void draw_screen():
        panel->draw()
        scrollbar->draw()
        button->draw()
        …
```



- Say we have a Canvas object that needs to draw the screen. Would this be a good way of drawing all the widgets?

Now how do we draw all these widgets?

```
class Canvas:
    UIPanel* panel; UIScrollbar* scrollbar; UIButton* button;

    Canvas(UIPanel* panel, UIScrollbar* scrollbar, …, Button* button) {
        this->panel = panel
        this->scrollbar = scrollbar
        this->button = button
          …

    void draw_screen():
        panel->draw()
        scrollbar->draw()
        button->draw()
        …
```



- This would be terrible! Why inherit if you have to keep an **explicit reference to each subtype**?

Liskov Substitution Principle

- The **Liskov substitution principle** states that if ***S*** is a subtype of ***T***, then objects of type ***T*** may be replaced (or substituted) with objects of type ***S***.

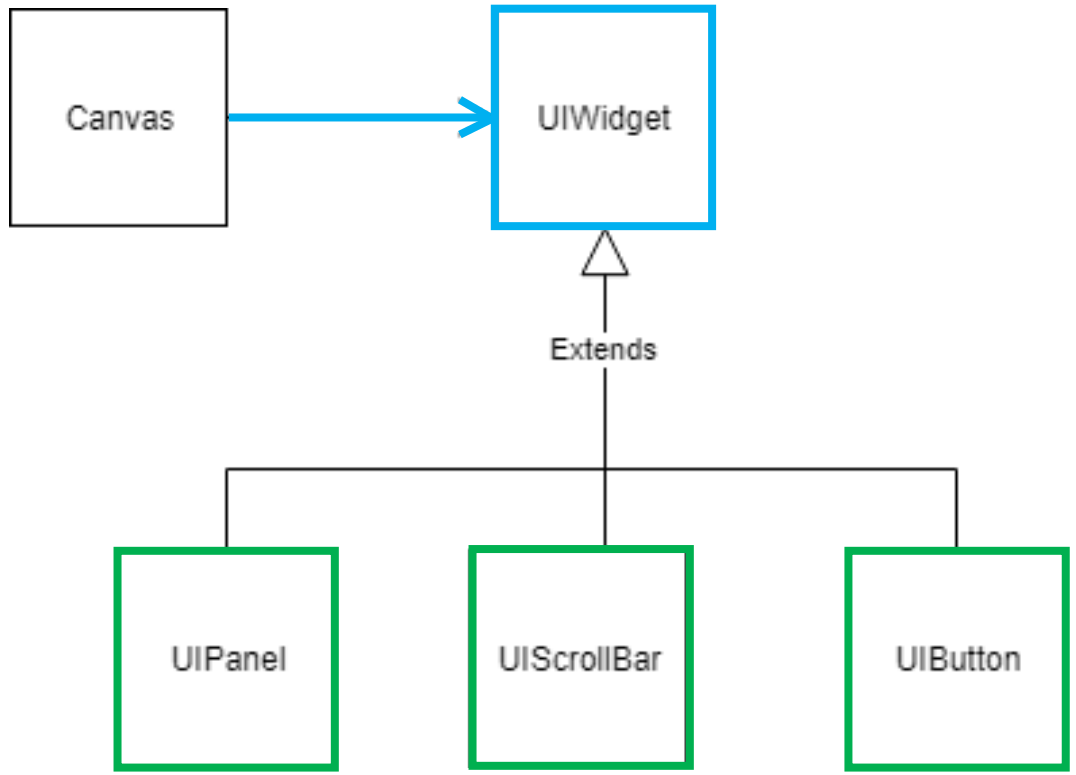
OR

- Objects in a program should be replaceable in code with references of their **base types** without altering the correctness of that program.

- Simply put, you should be able to refer to all the different **child UI components** as its base class, **UIWidget**

Liskov Substitution Principle

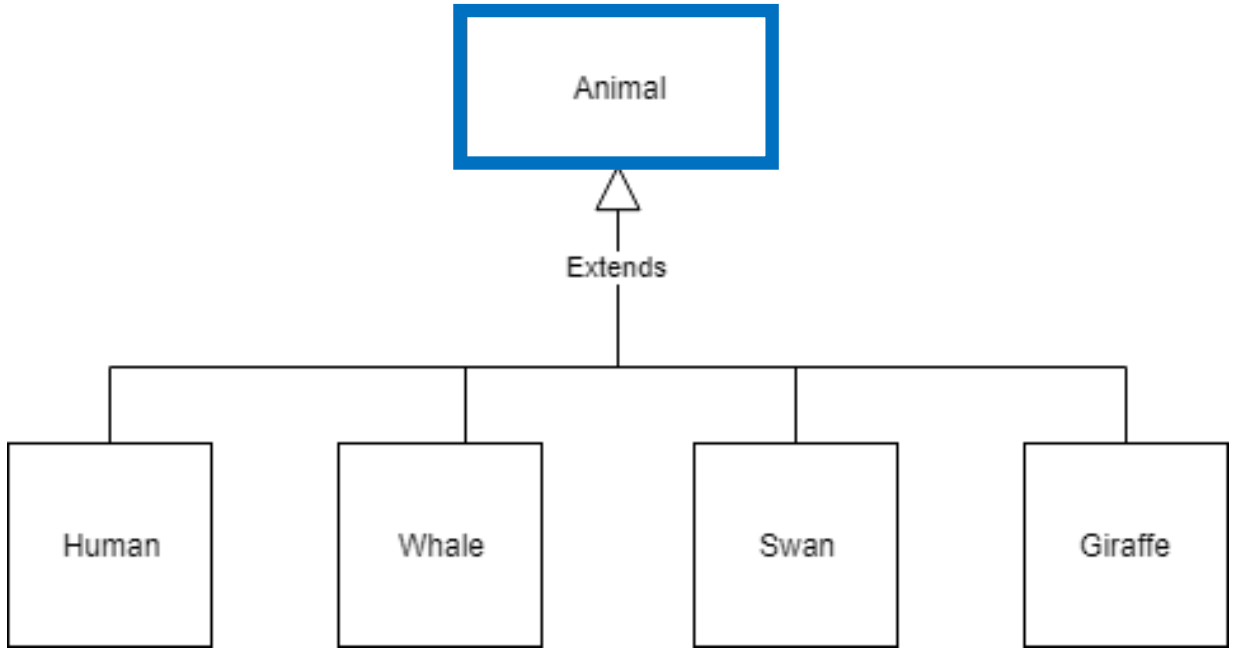
```
class Canvas:
    vector<UIWidget*> widget_list;

    Canvas(vector <UIWidget*> widget_list):
        this->ui_widgets = widget_list

    void draw_screen():
        for (UIWidget *widget : ui_widgets)
            widget->draw()
```



- Now we just have a **vector** filled with pointers to many derived **UIWidgets** (**UIPanel**, **UIScrollBar**, **UIButton** etc).
- Since they all have the same interface (**draw function**) they can be treated as if they were their base class (**UIWidget**).
- We don't care about the derived type. Let polymorphism determine the derived type at runtime

INTERFACE SEGREGATION PRINCIPLE

Now Say We Were Modeling A Zoo…




```
class Animal {
    virtual void breathe() {
        //common breathing code here
    }

    virtual void eat() {
        // common eating code here
    }
    virtual void walk() = 0;

    virtual void swim() = 0;

    virtual void fly() = 0;
}
```

So we followed all our design principles and wrote the **Animal** abstract base class

But this is terrible! **Now our Human class needs to override the fly method. Our Fish can walk!**

Interface Segregation Principle

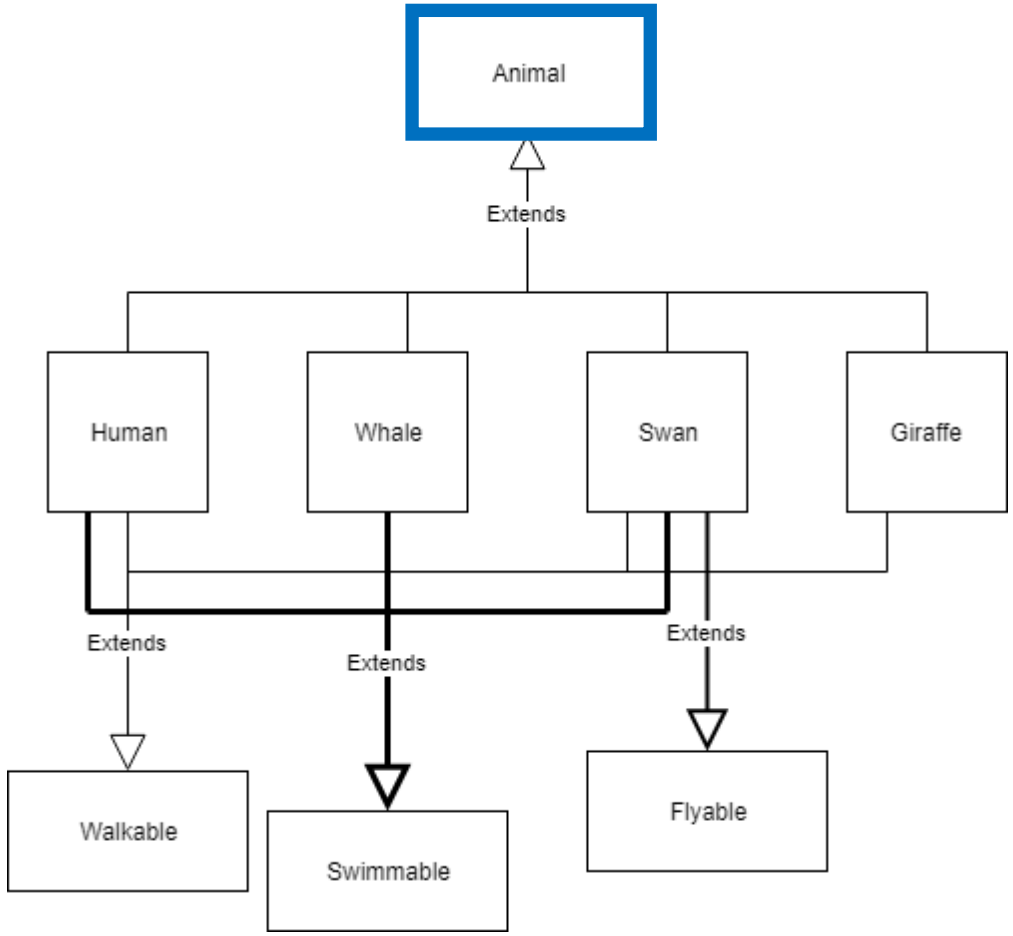
- Many client-specific interfaces are better than one general-purpose interface.

OR

- The interface segregation principle states that **no client should be forced to depend on methods it does not use.**

- Put more simply: Do not have interfaces with too many unnecessary methods. Have multiple simpler interfaces with fewer methods instead.
- This is the Single Responsibility Principle for Interfaces/Abstractions!

Back To Our Zoo!



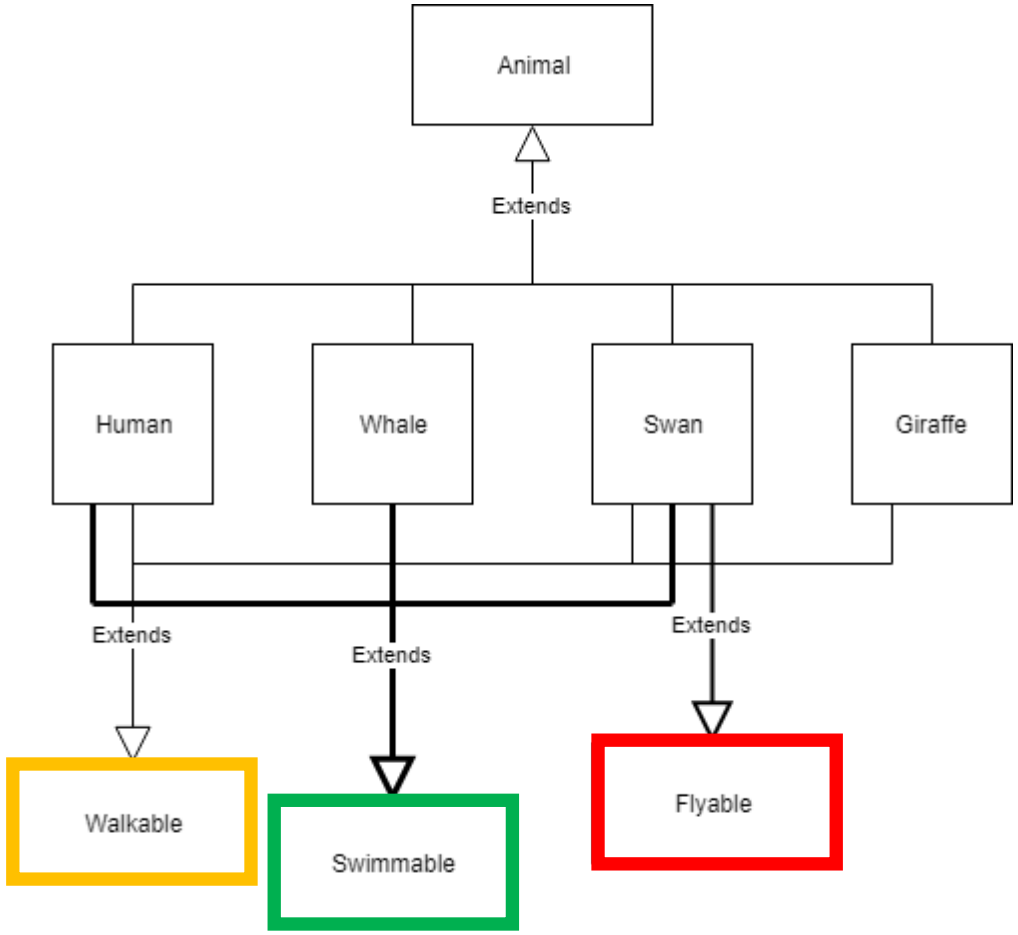
Instead of polluting the **Animal** base class with additional functionality, we **create separate interfaces that handle different responsibilities.**

This may seem difficult to maintain, but in fact it isn't.

Extract code that all Animals share

```
class Animal {
    virtual void breathe() {
        //common breathing code here
    }

    virtual void eat() {
        // common eating code here
    }
}
```

Back To Our Zoo!



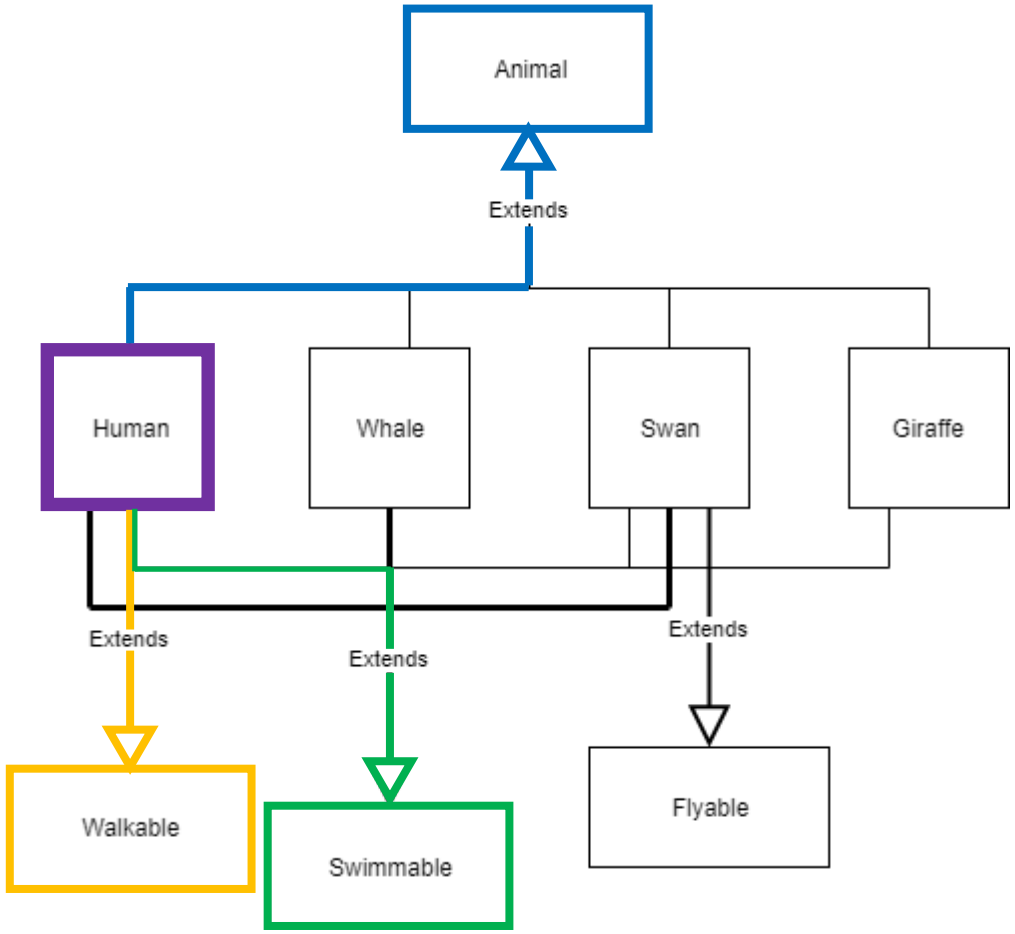
Instead of polluting the **Animal** base class with additional functionality, we **create separate interfaces that handle different responsibilities.**

This may seem difficult to maintain, but in fact it isn't.

Extract optional functionality to separate **interfaces**

```
class Walkable {
    virtual void walk() = 0;
};

class Swimmable {
    virtual void swim() = 0;
};

class Flyable {
    virtual void fly() = 0;
};
```

Back To Our Zoo!



Instead of polluting the **Animal** base class with additional functionality, we **create separate interfaces that handle different responsibilities.**

This may seem difficult to maintain, but in fact it isn't.

Human class inherits only the functionality they need

```
class Human : public Animal, public Walkable, public
Swimmable {
    void walk() override {
        //overridden walking code
    }

    void swim() override {
        //overridden swimming code
    }
};
```

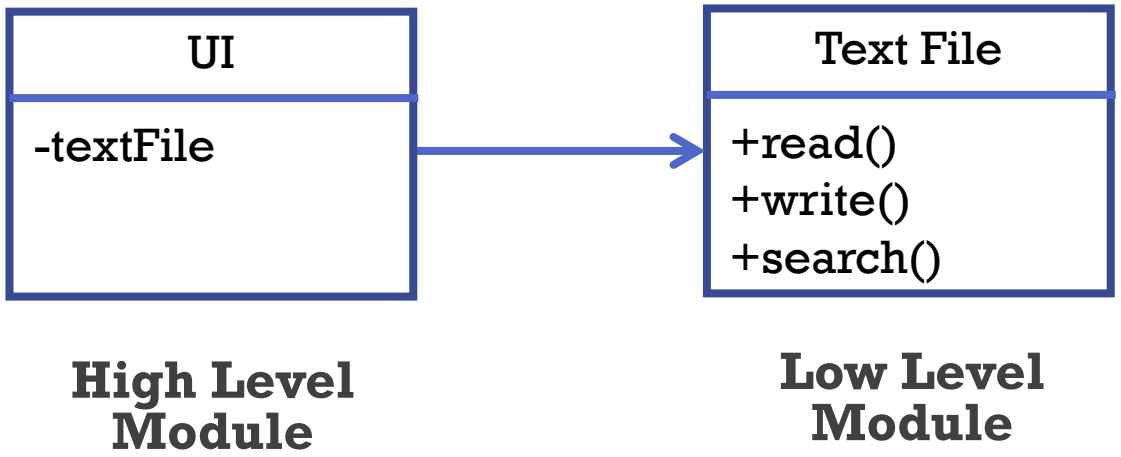
DEPENDENCY INVERSION PRINCIPLE

Dependency Inversion Principle

- **Dependency Inversion Principle** can be thought as a combined effect of the previous principles.
- Specifically, the **Open Closed Principle** and **Liskov Substitution Principle**

- Before we dive into this, we need to look at a few concepts.

High Level & Low Level Modules

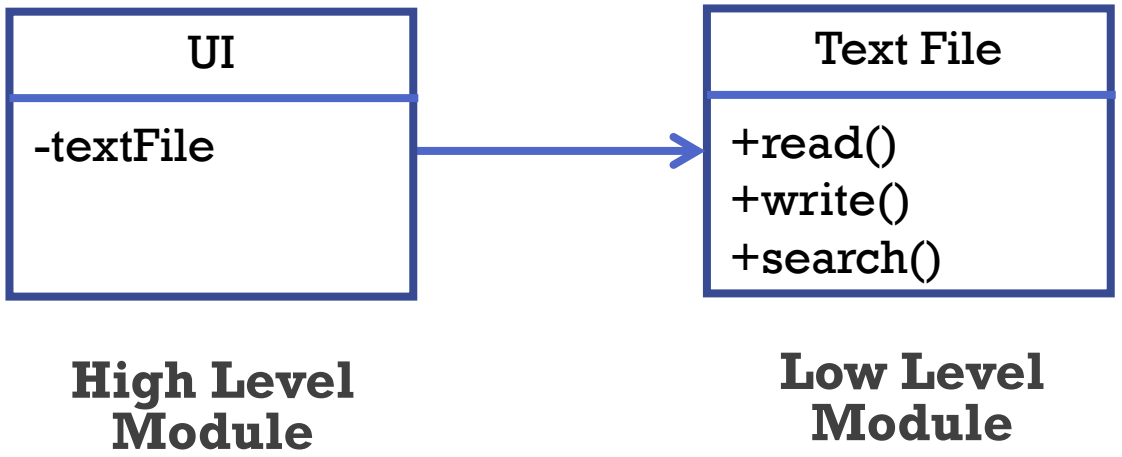


High Level Modules are the ones that contain complex logic made up of small 'chunks' of simpler code. For example, our UI class

A **Low Level Module** on the other hand, is a class that encapsulates some simple atomic behaviour. Such as writing and reading from a file, displaying an image, etc.

We generally compose high level modules with low level modules. That is, **high level modules are Coupled and Dependent on low level modules**

High Level & Low Level Modules

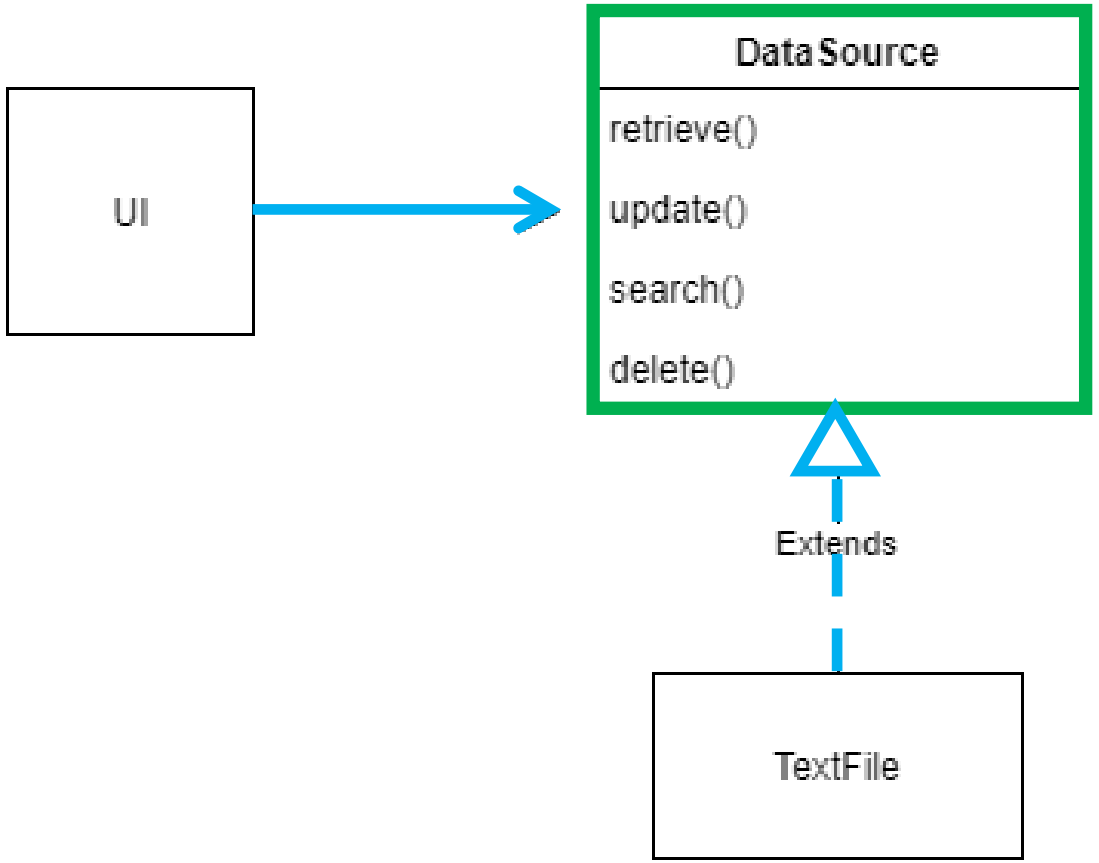


<u>This is not very good design.</u>

We don't want this dependency. We want to be able to change or replace the lower level modules without tampering with the complex code in the higher level modules

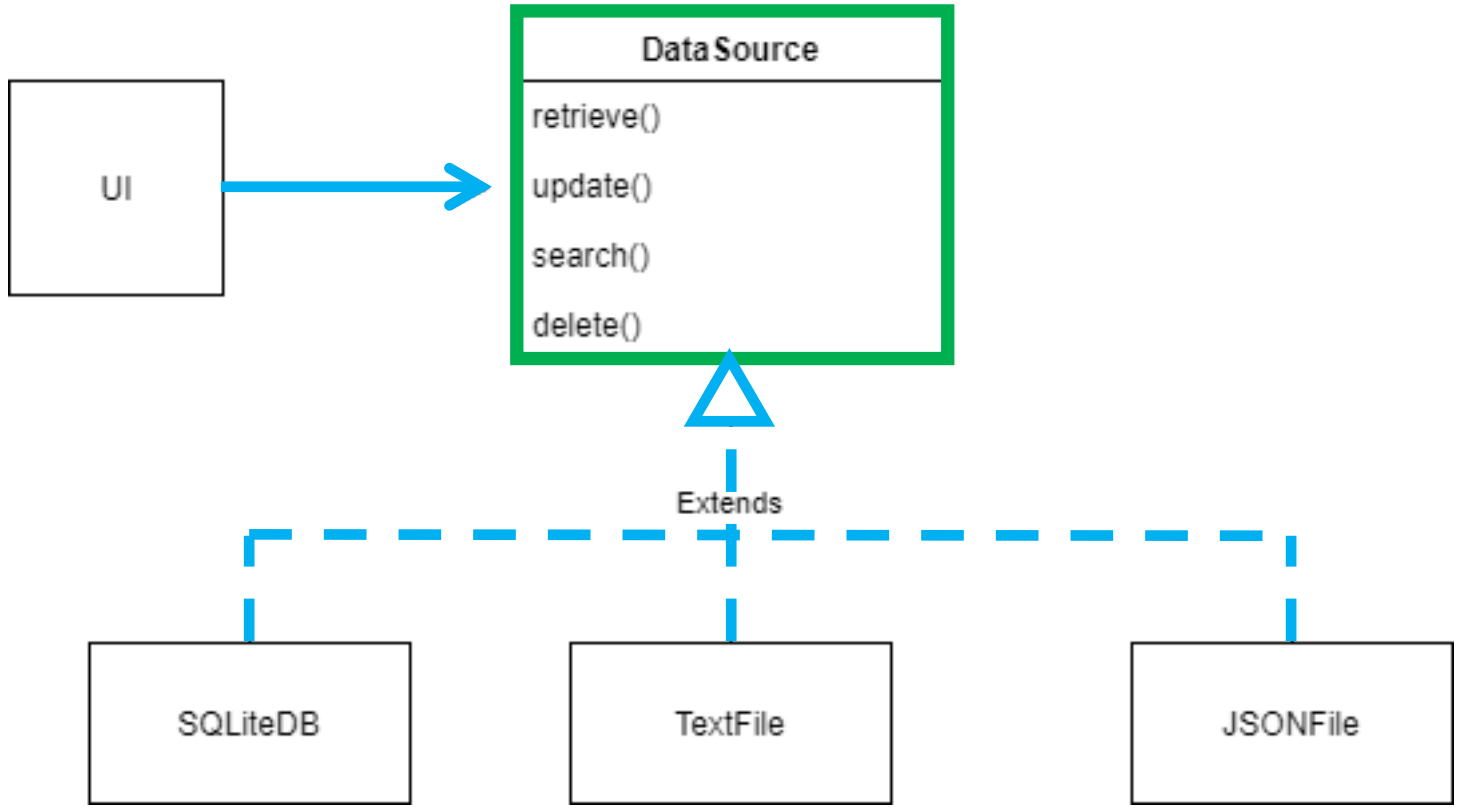
We inversed the dependencies!

- We introduced an **abstract layer**, that does not depend on anything.
- We made both our **high level (UI)** and **low level (TextFile)** modules **depend on** the **abstract** instead.
- Now we can switch out either side, the **UI** and the **text file** could be replaced with something completely different



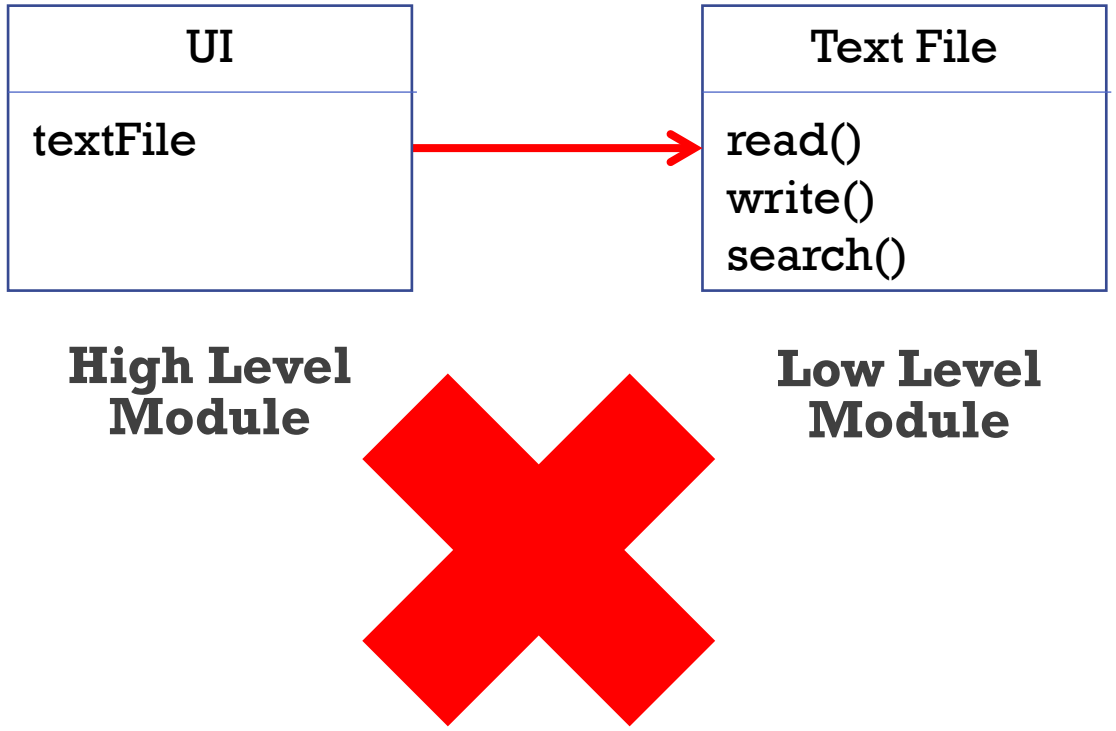
We inversed the dependencies!

- Everyone here has seen a diagram like this many times now.
- We can even see the building blocks that are **Liskov Substitution** and **Open Closed Principles**.

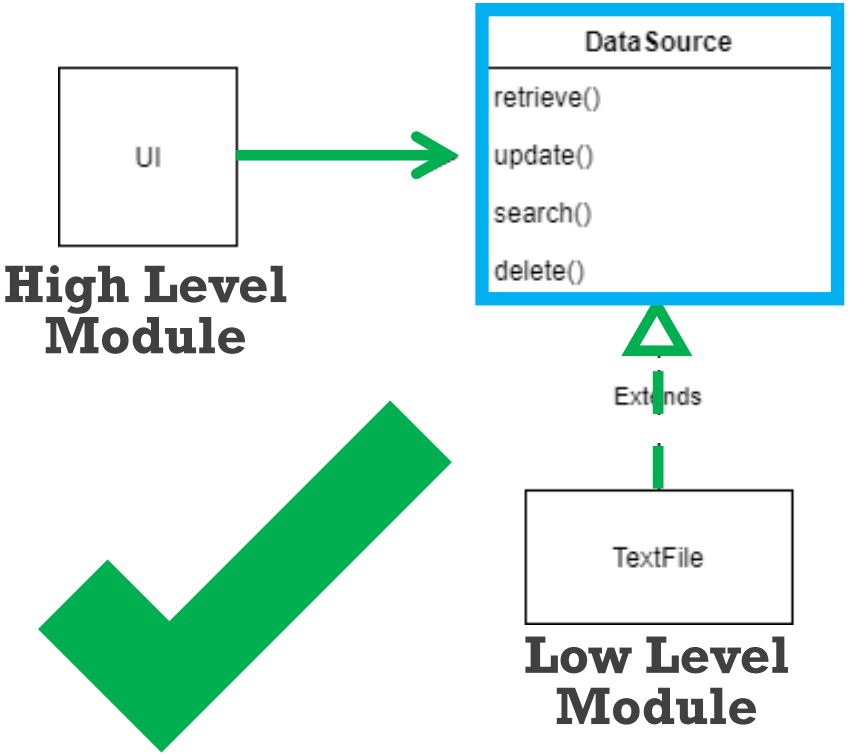


Dependency Inversion Principle

1. **High-level modules should not depend on low-level modules.**

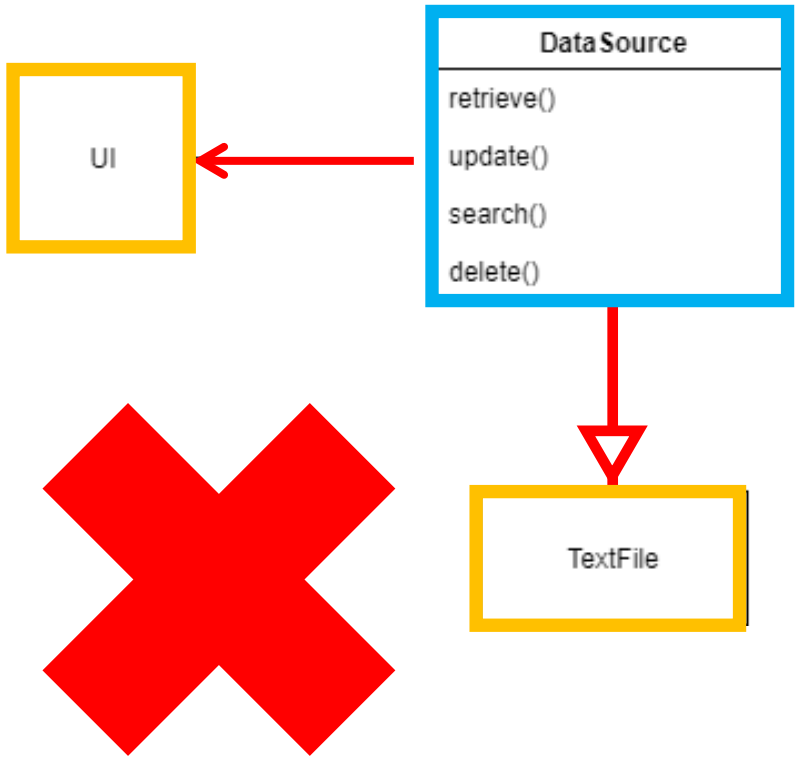


High-level modules and **low-level modules should depend on abstractions.**

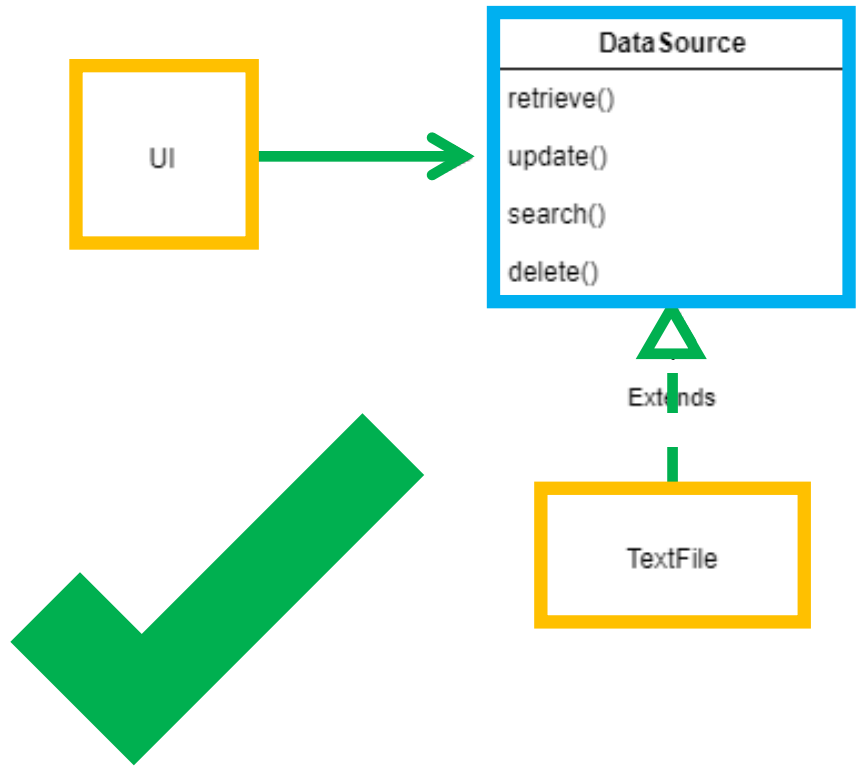


Dependency Inversion Principle

2. **Abstractions should not depend on details**

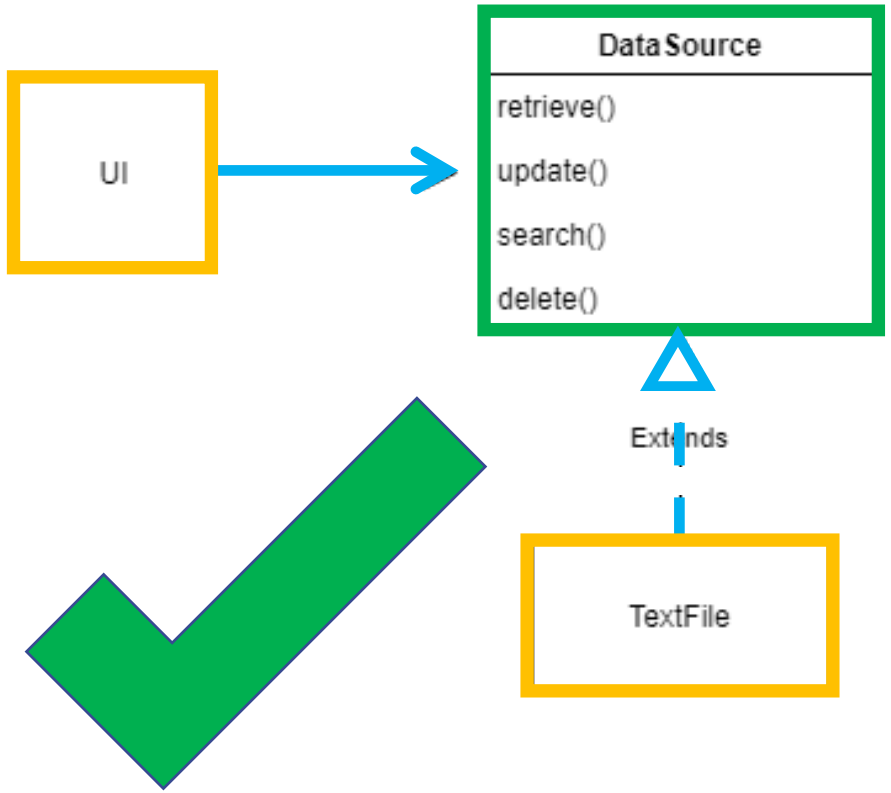


Details should depend on abstractions



Dependency Inversion Principle

In the example we just saw, the **DataSource** is our **abstract class or interface**, and the **UI** and **TextFile** classes are **concrete classes**.



Interfaces and Abstract classes

- declare an interface
- can NOT be instantiated

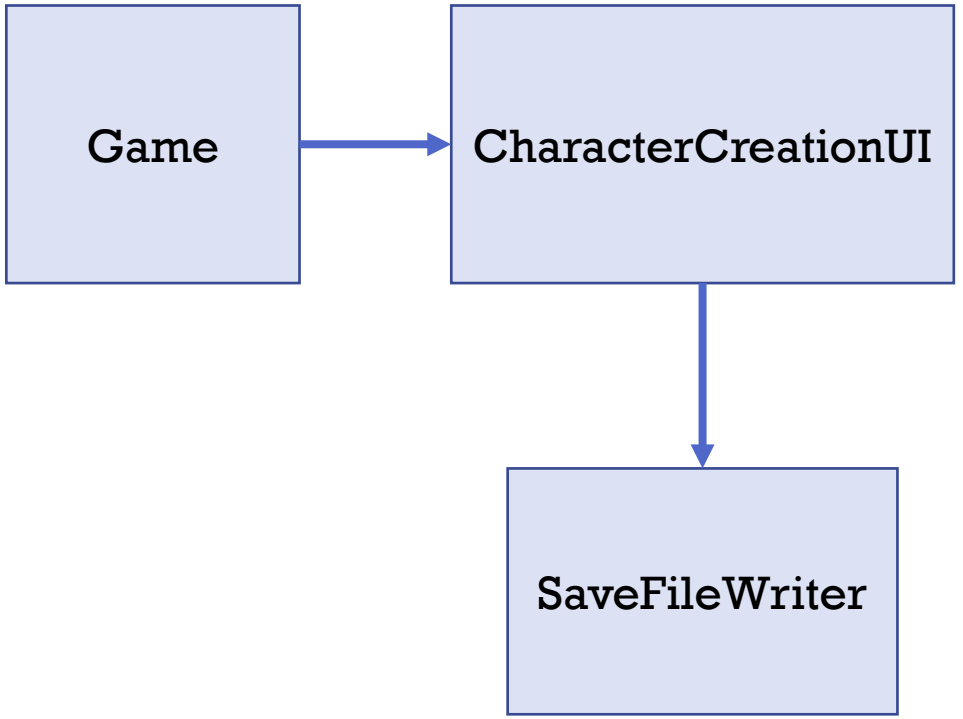
Concrete classes

- contain the code that implements the details of the interface (TextFile)
- are fully implemented classes (UI, TextFile)
- CAN be instantiated

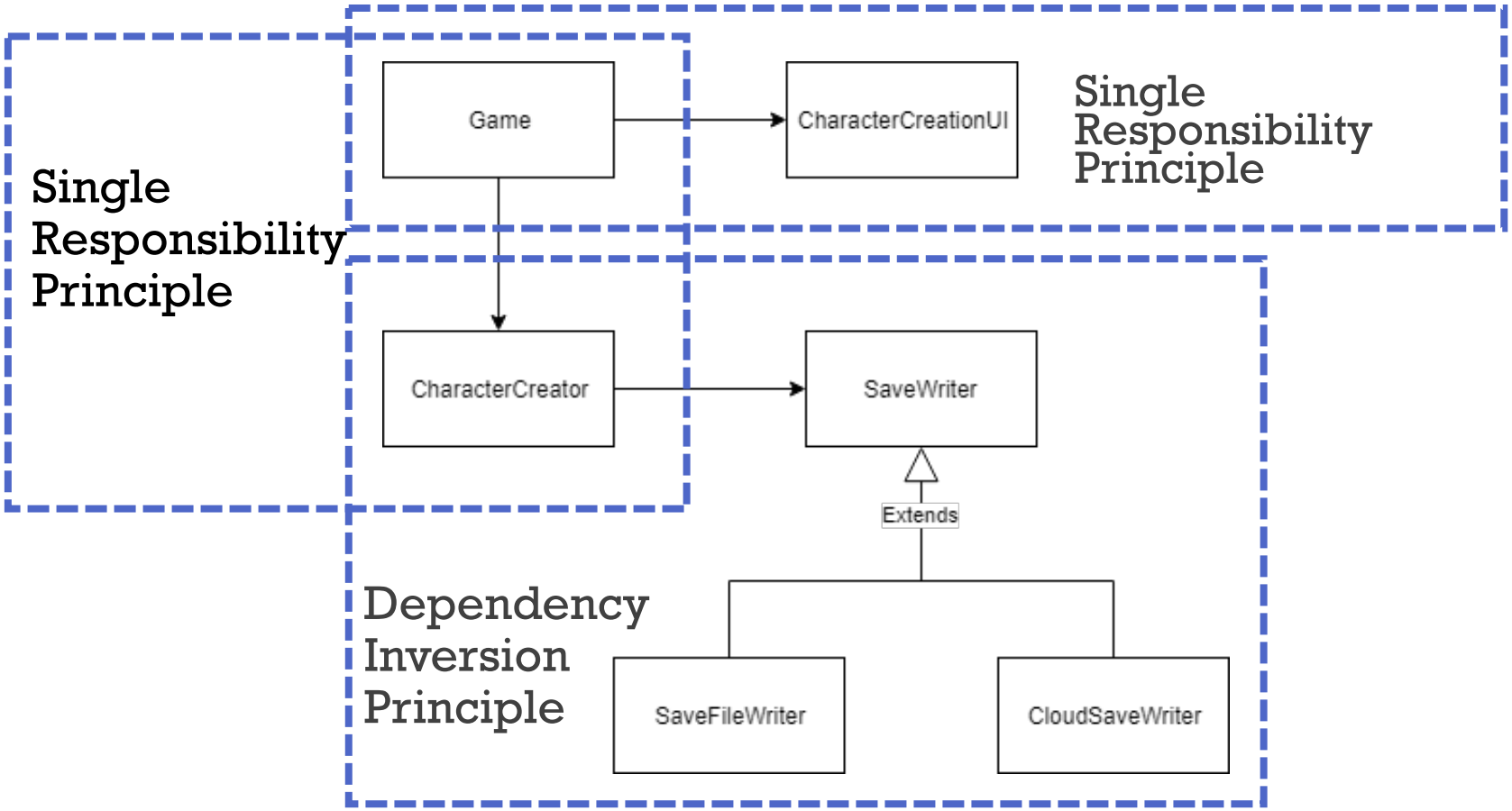
Say we had a character creation UI in a game.

We have a **game** with a **Character Creation UI**, where a player creates their character. When they click the create button, it gets saved to a **save file locally or online**.

Apply everything you've learnt so far to "fix" this terrible design.



My Solution


Single
Responsibility
Principle
Game
CharacterCreationUI
Single
Responsibility
Principle
CharacterCreator
SaveWriter
Extends
Dependency
Inversion
Principle
SaveFileWriter
CloudSaveWriter

LAW OF DEMETER

Law of Demeter

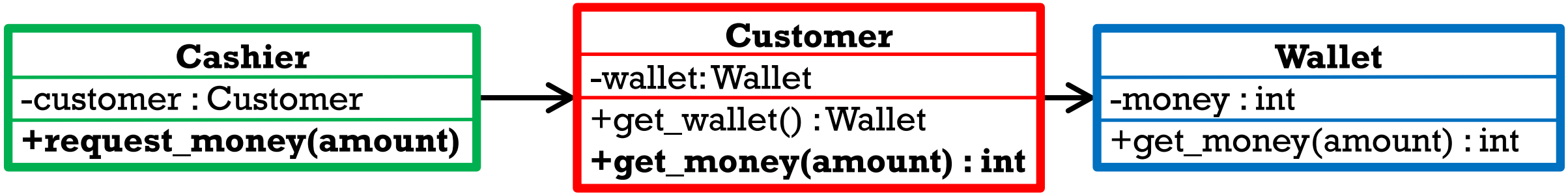
Real life - Wallet example

Law of Demeter - Problem

- **Cashier**, **Customer**, and **Wallet** are classes

- **Goal: Cashier** wants to get **int money** from the **Wallet**
- **Cashier** *should not* reach in **Customer**, then reach into their **Wallet** to get the money.
  - In **Cashier** **take_money()**: `money = customer.get_wallet().get_money(amount) #BAD`
- Classes **should not** have knowledge of public members several classes away

Law of Demeter - Solution



- **Goal:** **Cashier** wants to get **int money** from the **Wallet**
- **Cashier** can access **Customer**, but should not access **Wallet** through **Customer**
- Refactor code so **Cashier** requests money from **Customer**. **Customer** accesses own **Wallet** to give money
  - In **Cashier** **request_money()**: `money = customer.get_money(amount) #OK!`
  - In **Customer** **get_money()**: `return wallet.get_money(amount)`
- Classes **should** have knowledge of public members of **neighboring classes**

Law of Demeter

"Each unit should have only limited knowledge about other units: only units "closely" related to the current unit. Each unit should only talk to its friends; don't talk to strangers."

- Don't have trainwrecks like these:

```
obj.getX().getY().getZ().doSomething();
```

Or

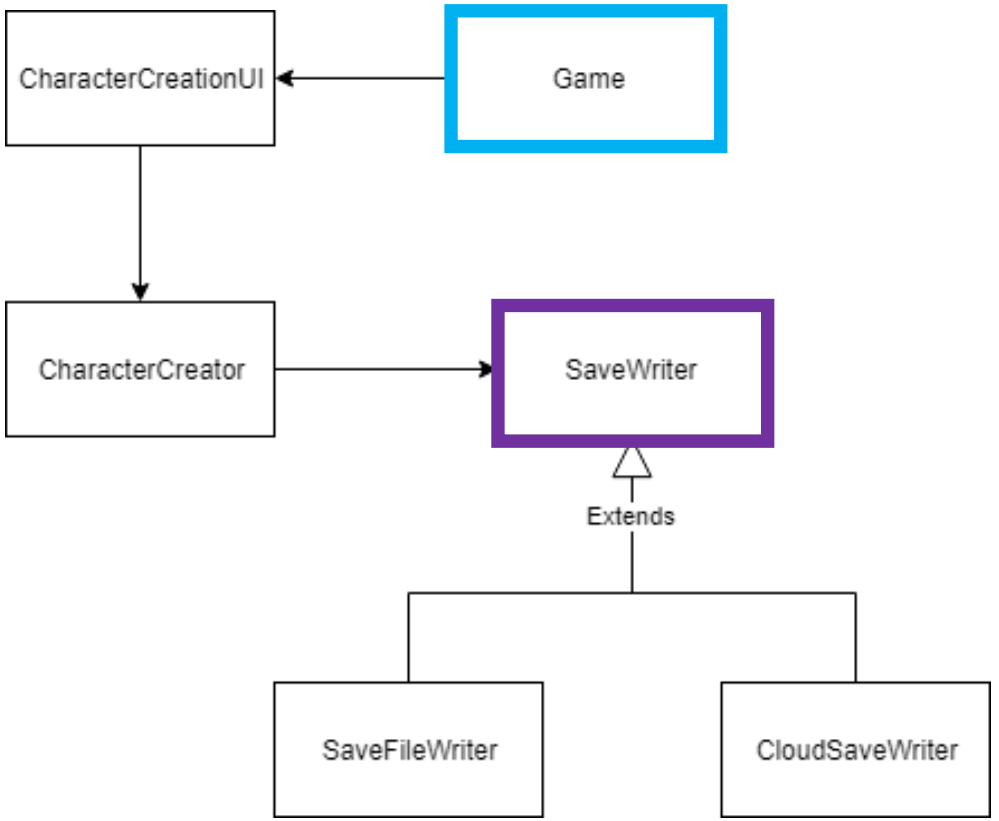
```
MyCharacterUI.GetCharacterData().CreateCharacter().Save()
```

- Poor Demeter is often forgotten; some people even call it "Suggestion of Demeter" since this is difficult to avoid

Law of Demeter – Example Code

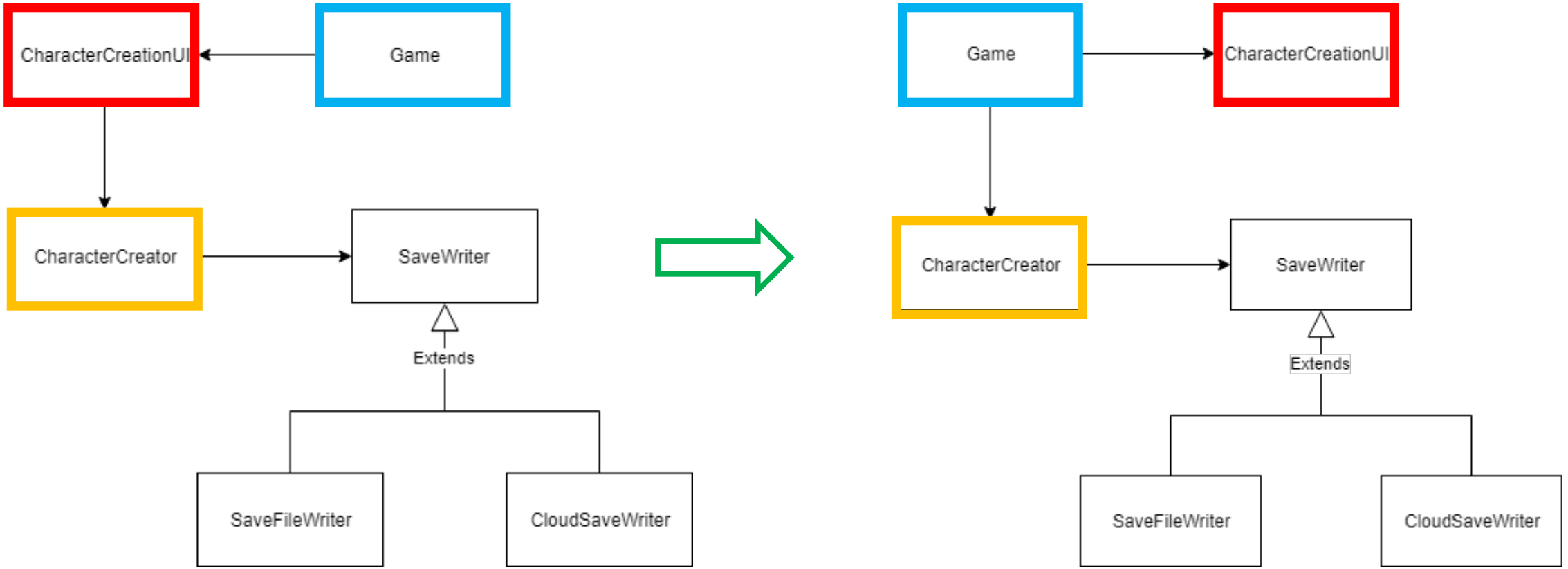
```
class Game {
    Data data;
    CharacterUI character_ui;
    void create_character() {
        data =
character_ui.get_character_creator().save(character_ui.get_character_creator().create_character());
```

- This code tries to create a character and save it
- This is obviously bad and hard to read code
- I also had a hard time getting the lines to wrap in powerpoint. Bad for many reasons
- Any change in the **save method**, which is all the way in **SaveWriter**, could have a ripple effect of changes all the way to our Game class



Law of Demeter – Example Code

- Part of the solution is to restructure the system by swapping Game and CharacterCreationUI
- Therefore Game does not need to *reach through* CharacterCreationUI to get CharacterCreator



Law of Demeter – Example Code

Restructure code so each class is responsible for their own data. This prevents classes reaching deeply into multiple classes

```
class Game {
    void create_character() {
      data = character_ui.get_data();
      character_creator.initiate_character_creation(data);
    }
};

class CharacterCreationUI {
    void get_data() {
      return character_data;
    }
};

class CharacterCreator {
    void initiate_character_creation(Data data) {
      my_char = create_character(data);
      save_writer.save(my_char);
    }
};
```