CSci 3081W: Program Design and Development

Inheritance, Polymorphism, SOLID (and other) design principles

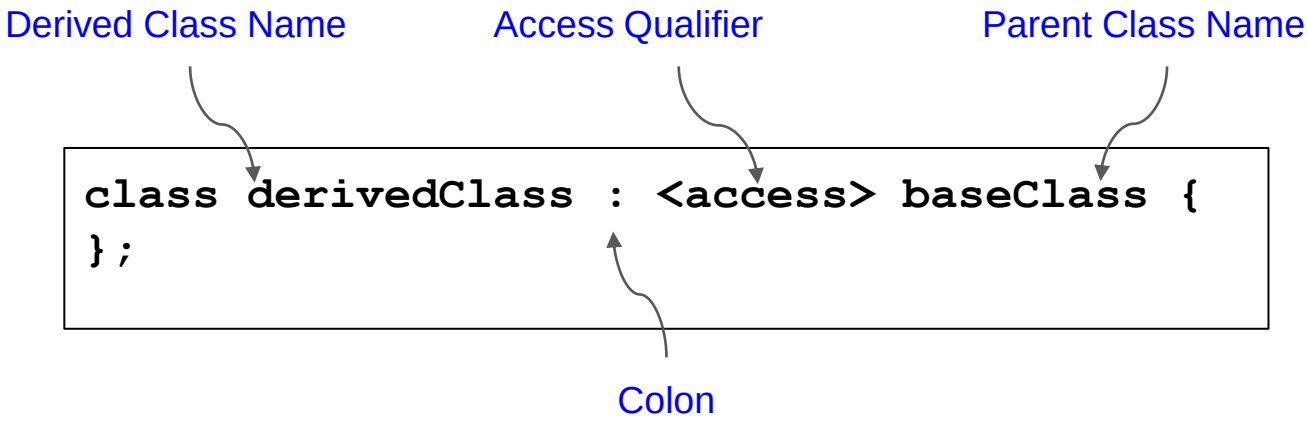
Inheritance in c++ is what you would think it is. Inheriting traits from a parent class.

```
class derived : public base {
};
```

In Human Biological Systems, **multiple inheritance** is implemented. Inheriting traits from multiple parents.

```
class derived : public baseA, public baseB {
};
```

In C++ **inheritance** allows us to inherit functionality from other classes.



<access> can be **public**, **protected** or **private**

In C++ **inheritance** allows us to inherit functionality from other classes.

Definition:

```
class derivedClass : <access> baseClass, <access> baseClass2, etc... {
};
```

Example:

```
#include <iostream>
using namespace std;

// Base class Shape
class Shape {
public:
                void setWidth(int w) {
                                width = w;
                }
                void setHeight(int h) {
                                height = h;
                }

protected:
                int width;
                int height;
};


// Derived class
class Rectangle: public Shape {
public:
                int getArea() {
                                return (width *
height);
                }
};
```

```
int main() {
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm

In C++ **inheritance** allows us to inherit functionality from other classes.

Definition:

```
class derivedClass : <access> baseClass, <access> baseClass2, etc... {
};
```

Example:

```
#include <iostream>
using namespace std;

// Base class Shape
class Shape {
public:
                void setWidth(int w) {
                                width = w;
                }
                void setHeight(int h) {
                                height = h;
                }

protected:
                int width;
                int height;
};


// Derived class
class Rectangle: public Shape {
public:
                int getArea() {
                                return (width *
height);
                }
};
```

```
int main() {
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

Almost always public so we can reuse code.

https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm

In C++ **inheritance** allows us to inherit functionality from other classes.

Definition:

```
class derivedClass : <access> baseClass, <access> baseClass2, etc... {
};
```

Example:

```
#include <iostream>
using namespace std;

// Base class Shape
class Shape {
public:
        void setWidth(int w) {
                width = w;
        }
        void setHeight(int h) {
                height = h;
        }

protected:
        int width;
        int height;
};


// Derived class
class Rectangle: public Shape {
public:
        int getArea() {
                return (width *
height);
        }
};
```

```
int main() {
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

What do you think protected means?

https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm

The **protected** access qualifier allows derived classes to use member variables and methods.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

In C++ **inheritance** allows us to inherit functionality from other classes.

Example:

```
// Base class Shape
class Shape {
public:
            void setWidth(int w) {
                        width = w;
            }
            void setHeight(int h) {
                        height = h;
            }

protected:
            int width;
            int height;
};


// Derived class
class Rectangle: public Shape {
public:
            int getArea() {
                        return (width *
height);
            }
};
```

```
int main() {
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea()
<< endl;

   return 0;
}
```

Protected allows us to use width and height in the derived class.

https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm

Example: Let's create more derived classes from the Shape class.

```
// Base class Shape
class Shape {
public:
            void setWidth(int w) {
                        width = w;
            }
            void setHeight(int h) {
                        height = h;
            }

protected:
            int width;
            int height
};


// Derived class
class Rectangle: public Shape {
public:
            int getArea() {
                        return (width *
height);
            }
};
```

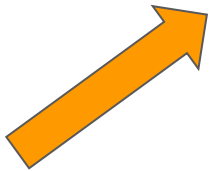
Example: Let's create more derived classes from the Shape class.

```
// Base class Shape
class Shape {
public:
            void setWidth(int w) {
                        width = w;
            }
            void setHeight(int h) {
                        height = h;
            }

protected:
            int width;
            int height;
};


// Derived class
class Rectangle: public Shape {
public:
            int getArea() {
                        return (width *
height);
            }
};
```

```
// Derived class
class Triangle: public Shape {
Public:
            int getArea() {
                        return 0.5 * (width
* height);
            }
};
```

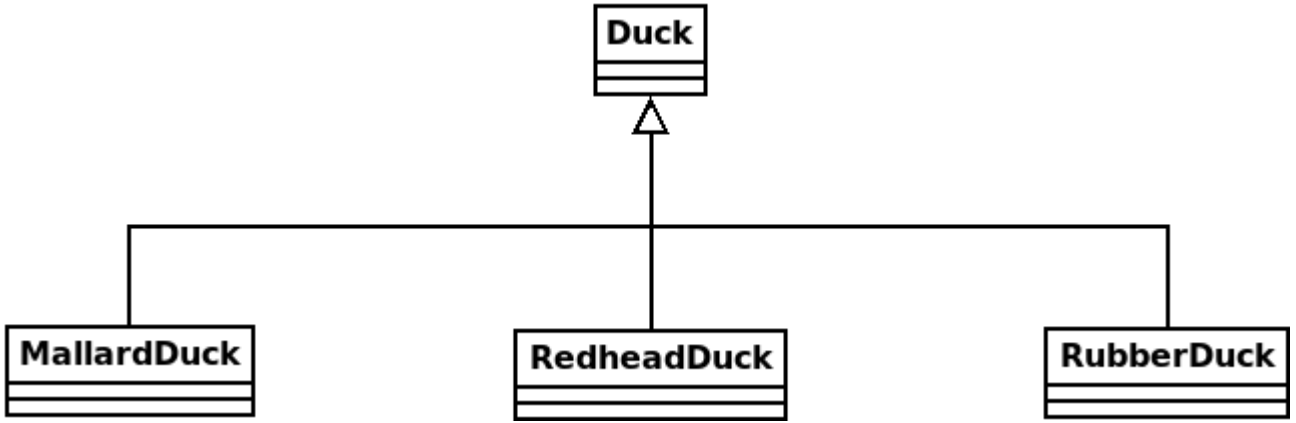
Example: Let's create more derived classes from the Shape class.

```
// Base class Shape
class Shape {
public:
            void setWidth(int w) {
                        width = w;
            }
            void setHeight(int h) {
                        height = h;
            }

protected:
            int width;
            int height;
};


// Derived class
class Rectangle: public Shape {
public:
            int getArea() {
                        return (width *
height);
            }
};
```

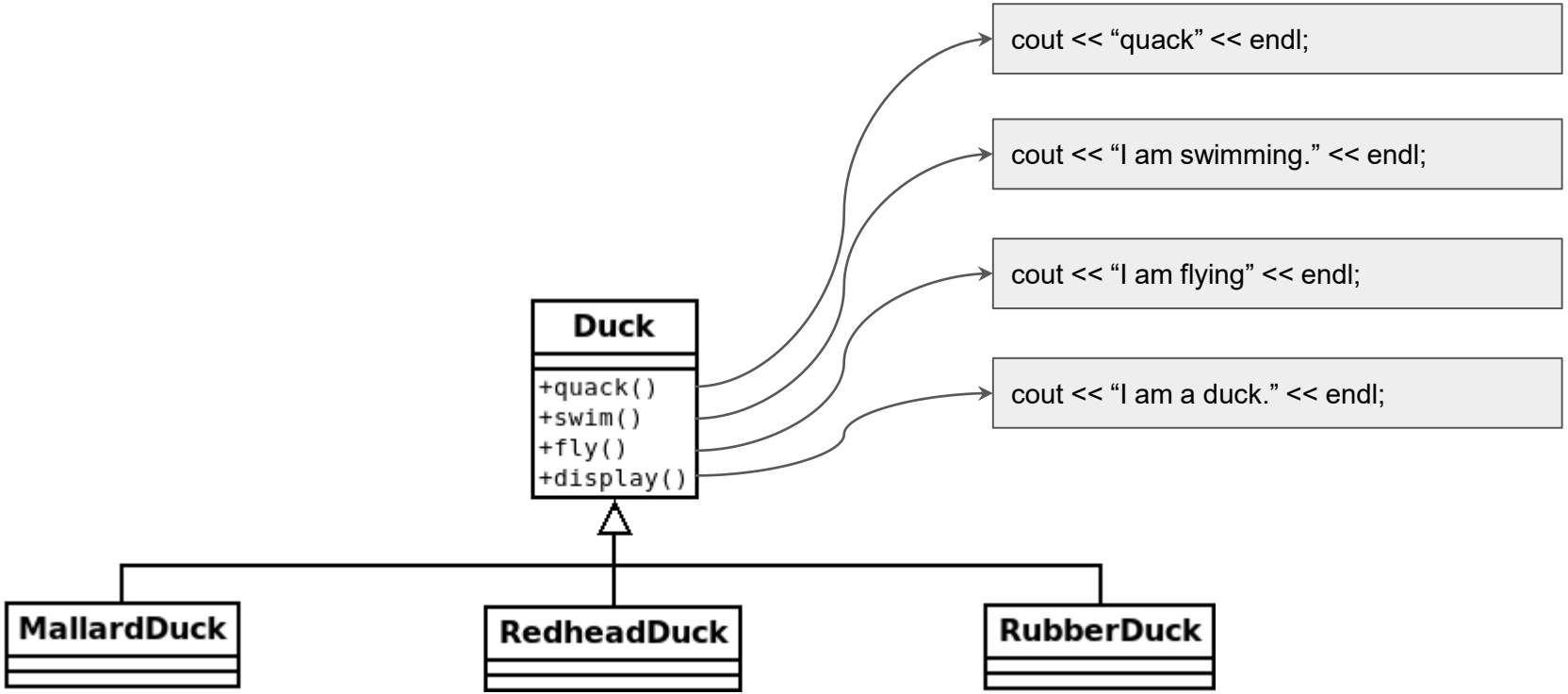


```
// Derived class
class Triangle: public Shape {
Public:
            int getArea() {
                        return 0.5 * (width
* height);
            }
};


// Derived class
class Square: public Rectangle {
public:
            void setWidth(int w) {
                        width = w;
                        height = width;
            }
            void setHeight(int h) {
                        height = h;
                        width = height;
            }
};
```

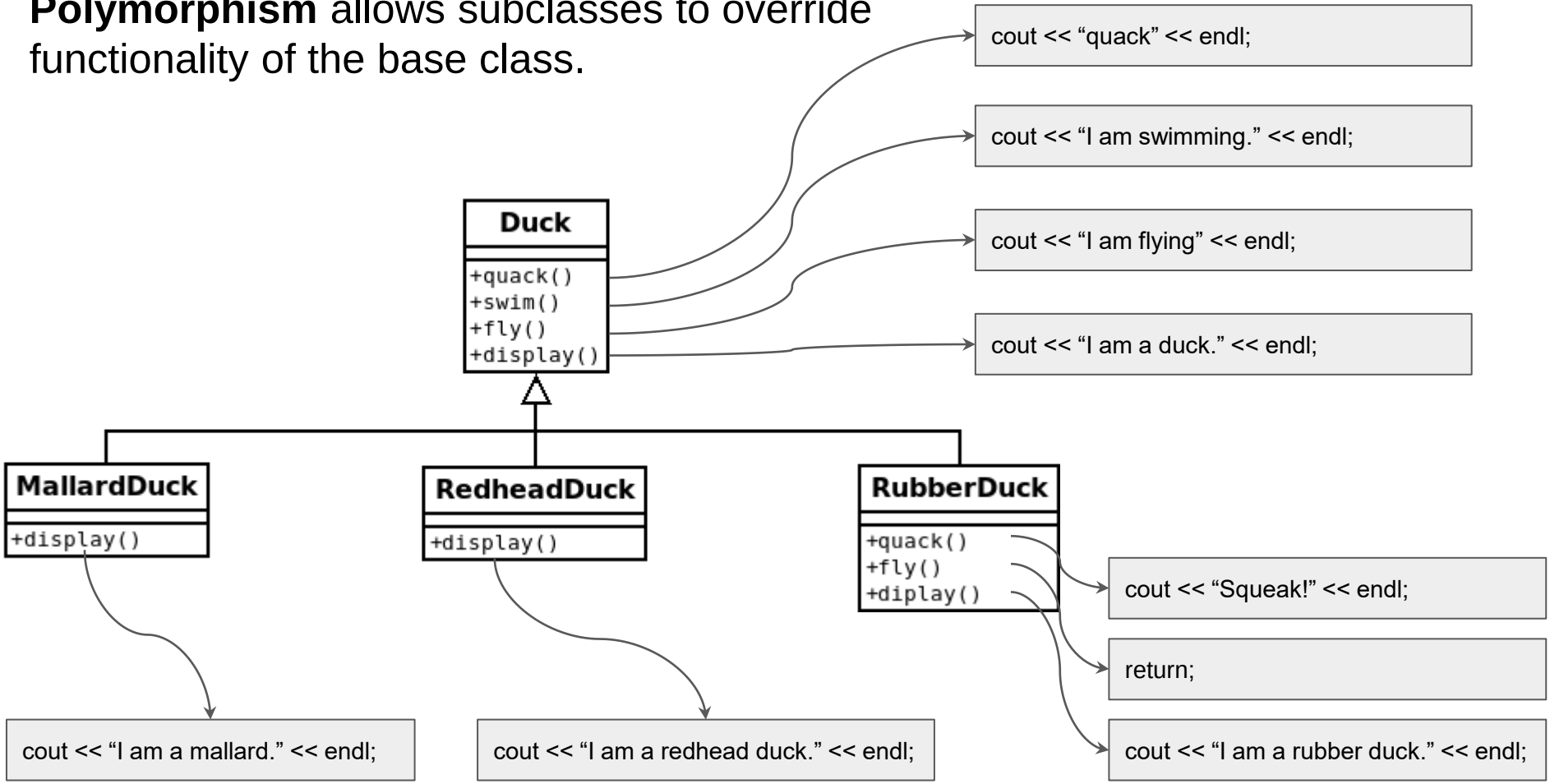
What do all ducks share in common?



What is wrong with this picture?

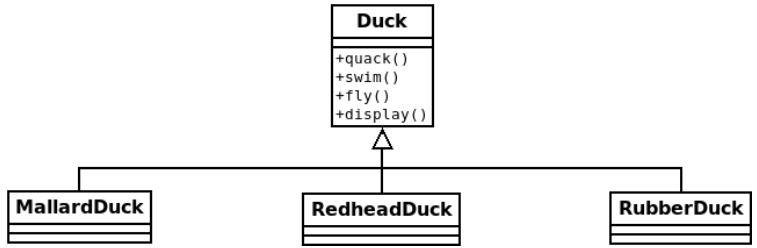


Polymorphism allows subclasses to override functionality of the base class.



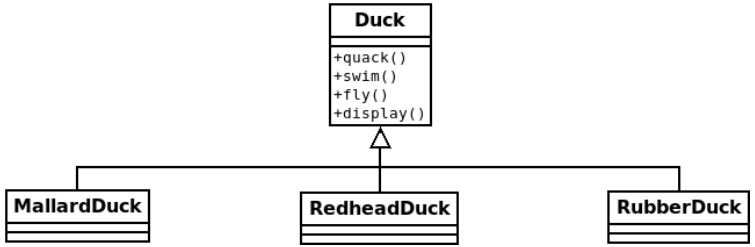
We have already studied basic inheritance (so let's start there).

```
class Duck {
public:
              void quack() { cout << "Quack" << endl; }
              void swim() { cout << "Swim" << endl; }
              void fly() { cout << "Fly" << endl; }
              void display() { cout << "I am a duck." << endl; }
};


class MallardDuck : public Duck {
public:
              void display() { cout << "I am a mallard duck." << endl; }
};


class RedheadDuck : public Duck {
public:
              void display() { cout << "I am a redhead duck." << endl; }
};


class RubberDuck : public Duck {
public:
              void quack() { cout << "Squeak!" << endl; }
              void fly() { return; }
              void display() { cout << "I am a rubber duck." << endl; }
};
```



We have already studied basic inheritance (so let's start there).

```
class Duck {
public:
            void quack() { cout << "Quack" << endl; }
            void swim() { cout << "Swim" << endl; }
            void fly() { cout << "Fly" << endl; }
            void display() { cout << "I am a duck." << endl; }
};


class MallardDuck : public Duck {
public:
            void display() { cout << "I am a mallard duck." << endl; }
};


class RedheadDuck : public Duck {
public:
            void display() { cout << "I am a redhead duck." << endl; }
};


class RubberDuck : public Duck {
public:
            void quack() { cout << "Squeak!" << endl; }
            void fly() { return; }
            void display() { cout << "I am a rubber duck." << endl; }
};
```



What is the output here?

```
int main()
{
            Duck duck;
            duck.quack();

            Duck someDuck = RubberDuck();
            someDuck.quack();

            RubberDuck rubberDuck =
RubberDuck();
            rubberDuck.quack();

            return 0;
}
```

We have already studied basic inheritance (so let's start there).

```
class Duck {
public:
            void quack() { cout << "Quack" << endl; }
            void swim() { cout << "Swim" << endl; }
            void fly() { cout << "Fly" << endl; }
            void display() { cout << "I am a duck." << endl; }
};


class MallardDuck : public Duck {
public:
            void display() { cout << "I am a mallard duck." << endl; }
};


class RedheadDuck : public Duck {
public:
            void display() { cout << "I am a redhead duck." << endl; }
};


class RubberDuck : public Duck {
public:
            void quack() { cout << "Squeak!" << endl; }
            void fly() { return; }
            void display() { cout << "I am a rubber duck." << endl; }
};
```

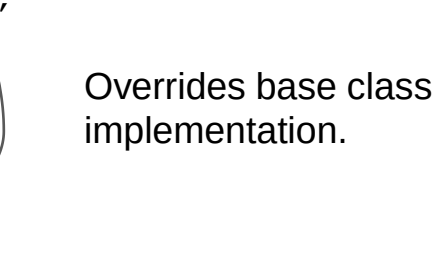
What is the output here?

```
int main()
{
            Duck duck;
            duck.quack();

            Duck someDuck = RubberDuck();
            someDuck.quack();

            RubberDuck rubberDuck =
RubberDuck();
            rubberDuck.quack();

            return 0;
}
```

Polymorphism allows us to override methods using **virtual functions**.

(Poly)morphism = many forms

```
class BaseClass {
public:
        virtual return_type function(params);
};


class DerivedClass : public BaseClass {
public:
        return_type function(params);
};
```



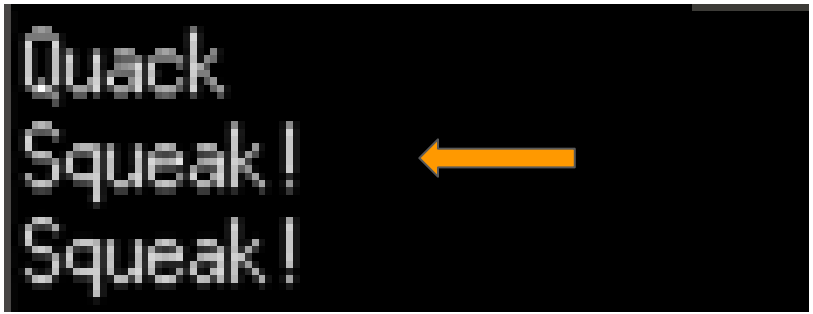
Polymorphism allows us to override methods using **virtual functions**.

```
class Duck {
public:
		virtual void quack() { cout << "Quack" << endl; }
};


class RubberDuck : public Duck {
public:
		void quack() { cout << "Squeak!" << endl; }
};



int main()
{
		Duck duck;
		duck.quack();

		Duck someDuck = RubberDuck();
		someDuck.qu

		RubberDuck rubberDuck =
RubberDuck();
		rubberDuck.quack();

		return 0;
}
```

Polymorphism allows us to override methods using **virtual functions**.

```
class Duck {
public:
        virtual void quack() { cout << "Quack" << endl; }
};


class RubberDuck : public Duck {
public:
        void quack() { cout << "Squeak!" << endl; }
};



int main()
{
        Duck duck;
        duck.quack();

        Duck* someDuck = new
RubberDuck();
        someDuck>quack();
        delete someDuck;

        RubberDuck rubberDuck =
RubberDuck();
        rubberDuck.quack();

        return 0;
}
```

Why does this work?



Pointers use the actual object rather than making a copy.

```
// base class copy
int main()
{
                Duck duck;
                duck.quack();

                RubberDuck rubberDuck =
RubberDuck();
                rubberDuck.quack();

                Duck someDuck = rubberDuck; //
copy
                someDuck.quack();

                return 0;
}
```

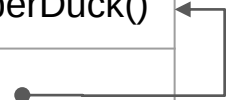
Memory

Variable	Value
duck	Duck()
rubberDuck	RubberDuck()
someDuck	Duck()

```
// virtual functions and pointers
int main()
{
                Duck duck;
                duck.quack();

                RubberDuck rubberDuck =
RubberDuck();
                rubberDuck.quack();

        Duck* someDuck = new RubberDuck();
                someDuck>quack();
                delete ducsomeDuckk2;

                return 0;
}
```

Memory

Variable	Value
duck	Duck()
rubberDuck	RubberDuck()
someDuck	

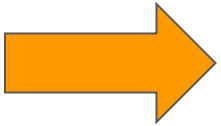


So what is the big deal?

```
int main()
{
    vector<Duck*> ducks;
    ducks.push_back(new MallardDuck());
    ducks.push_back(new RedheadDuck());
    ducks.push_back(new RubberDuck());
    ducks.push_back(new MallardDuck());
    ducks.push_back(new MallardDuck());
    ducks.push_back(new Duck());

    for (int i = 0; i < ducks.size(); i++) {
        ducks[i]->display();
        cout << "\t";
        ducks[i]->quack();
    }

    return 0;
}
```

Take away: We can call **overridden** methods from the base class!

```
int main()
{
    vector<Duck*> ducks;
    ducks.push_back(new MallardDuck());
    ducks.push_back(new RedheadDuck());
    ducks.push_back(new RubberDuck());
    ducks.push_back(new MallardDuck());
    ducks.push_back(new MallardDuck());
    ducks.push_back(new Duck());

    for (int i = 0; i < ducks.size(); i++) {
        ducks[i]->display();
        cout << "\t";
        ducks[i]->quack();
    }

    return 0;
}
```



```
I am a mallard duck.
        Quack
I am a redhead duck.
        Quack
I am a rubber duck.
        Squeak!
I am a mallard duck.
        Quack
I am a mallard duck.
        Quack
I am a duck.
        Quack
```

Pure virtual functions enforce a **contract**, but don't allow "instantiation" or an object (or "creation" of an object)

```
class BaseClass {
public:
        virtual return_type function(params) = 0;
};
```

No implementation so we cannot create an object of type BaseClass.

```
class DerivedClass : public BaseClass {
public:
        return_type function(params);
};
```

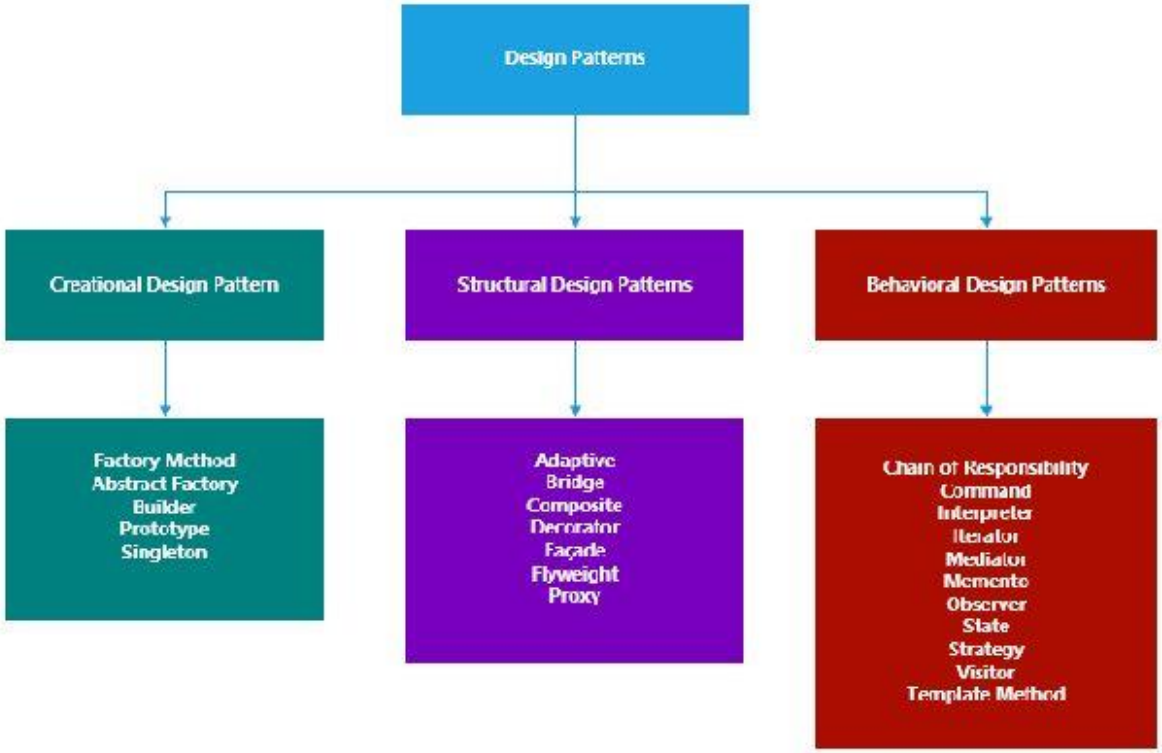
Let’s investigate this with ducks.

```
class Duck {
public:
            virtual void display() = 0;
};


class RubberDuck : public Duck {
public:
            void display() { cout << "I am a rubber duck." << endl; }
};


int main() {
            Duck d;

            RubberDuck rubberD

            return 0;
}
```

Abstract class

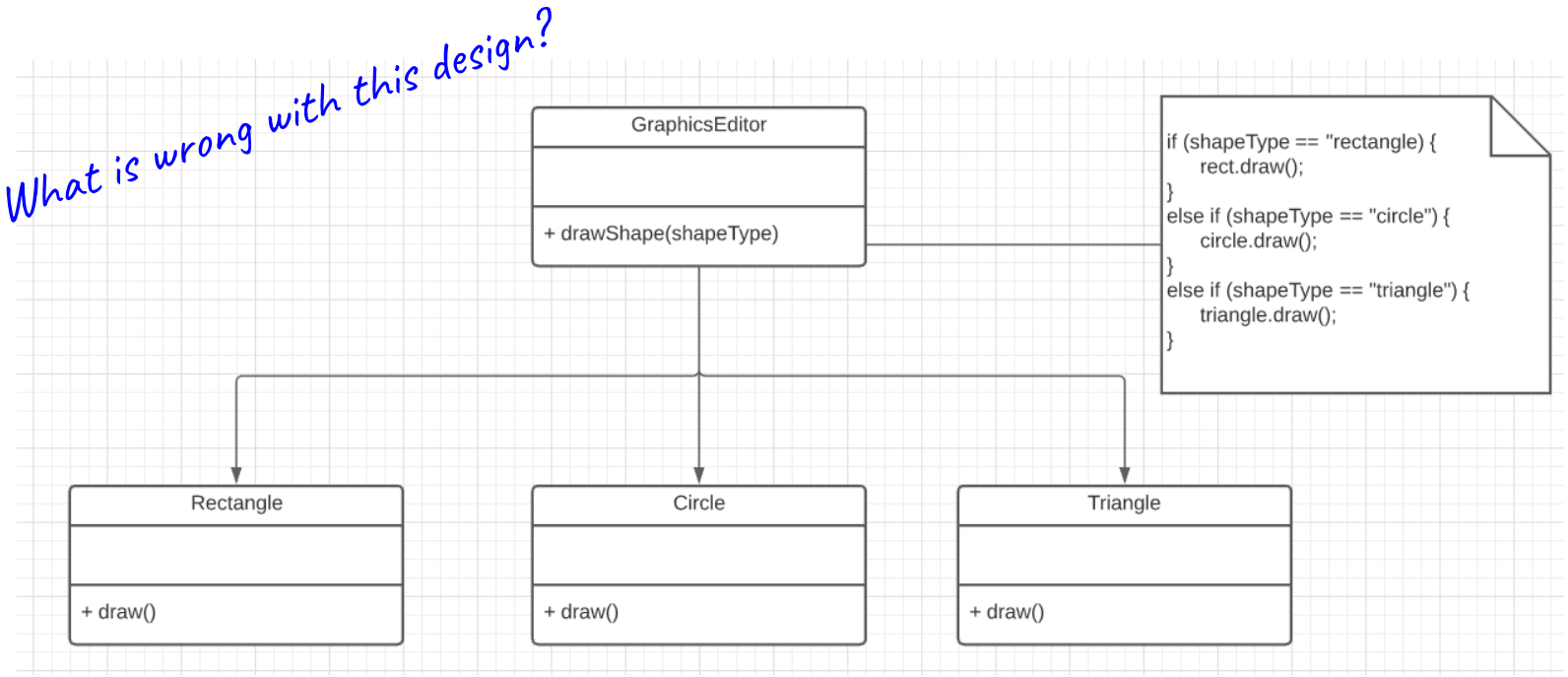
Pure virtual function

Design Patterns are object oriented designs that increase **flexibility**.

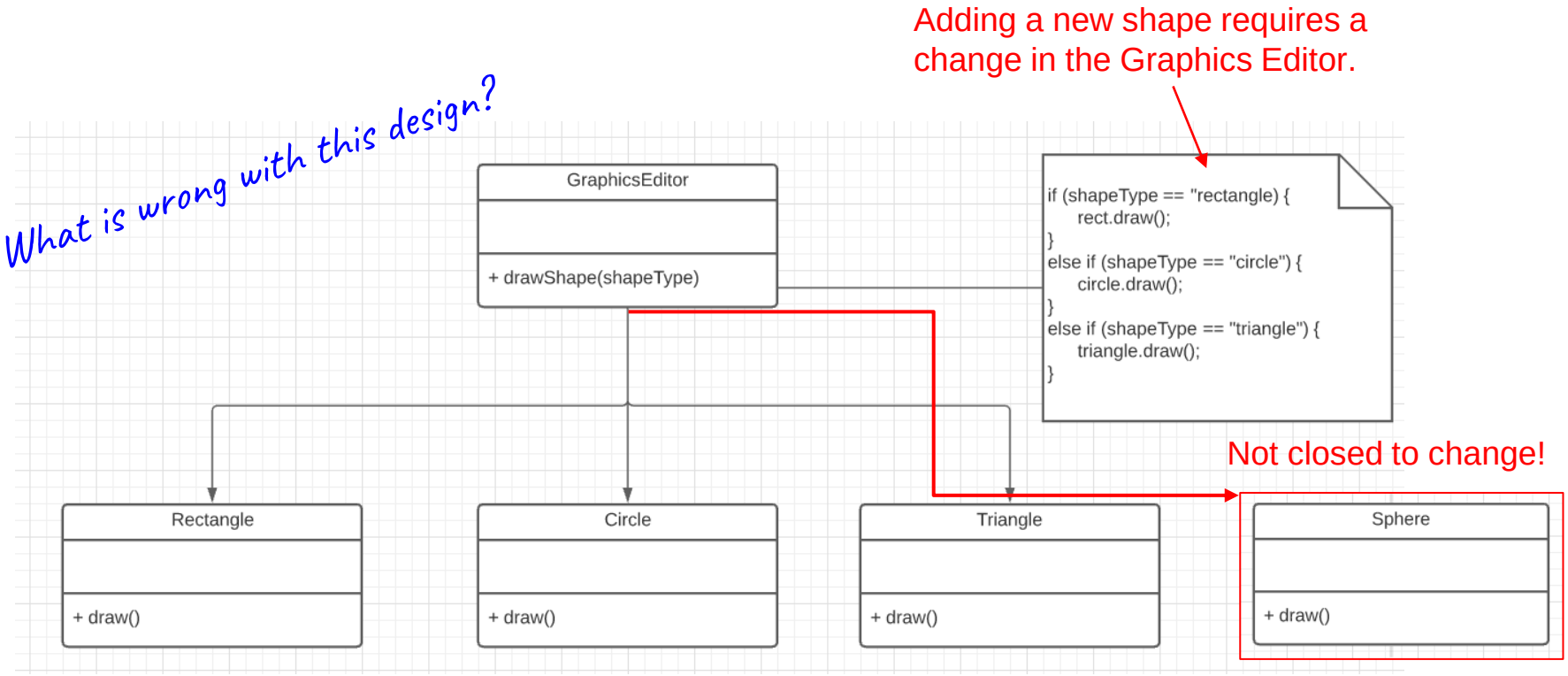


https://www.dofactory.com/net/design-patterns

The Open/Closed Design Principle: Software should open to extension, but closed to modification.

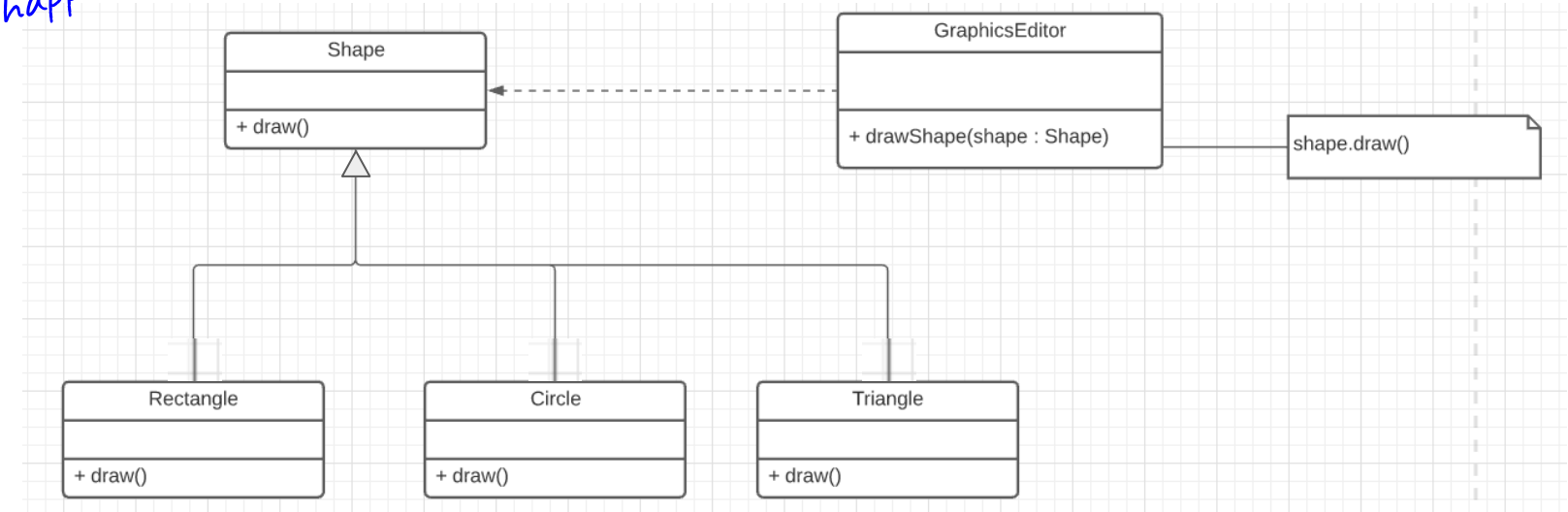


The Open/Closed Design Principle: Software should open to extension, but closed to modification.



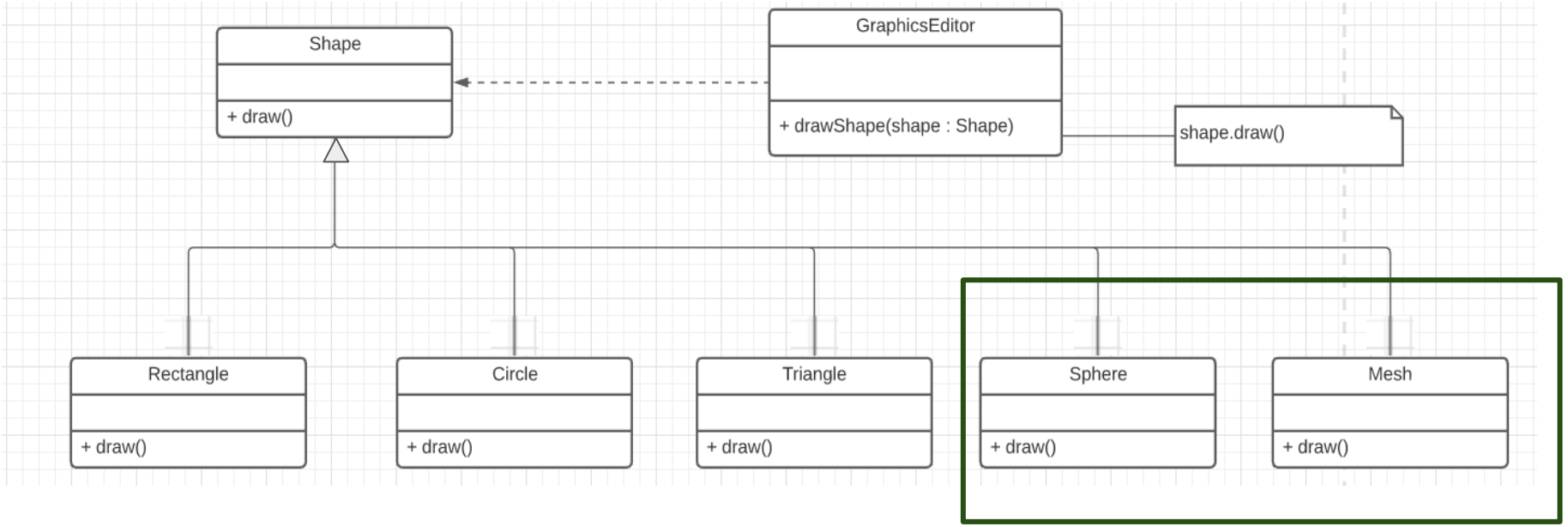
The Open/Closed Design Principle: Software should open to extension, but closed to modification.

How about this design?
What happens when you add a new shape?



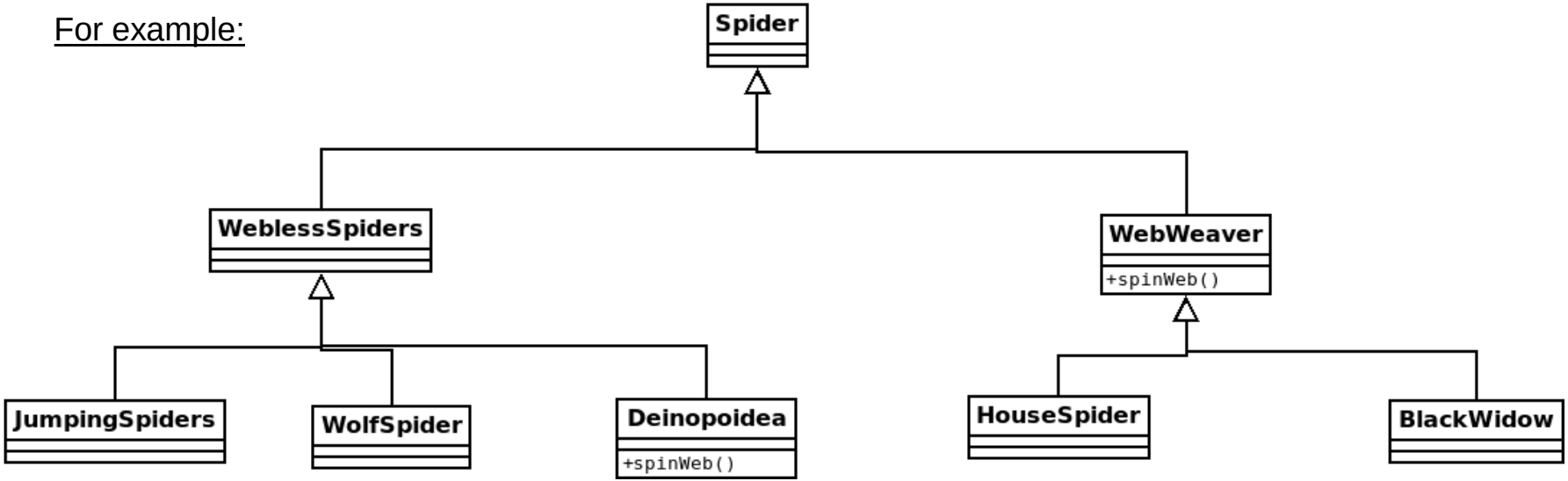
The Open/Closed Design Principle: Software should open to extension, but closed to modification.

Closed to change and open to extension.
Uses polymorphism!



Inheritance is not always the best tool to use.

For example:

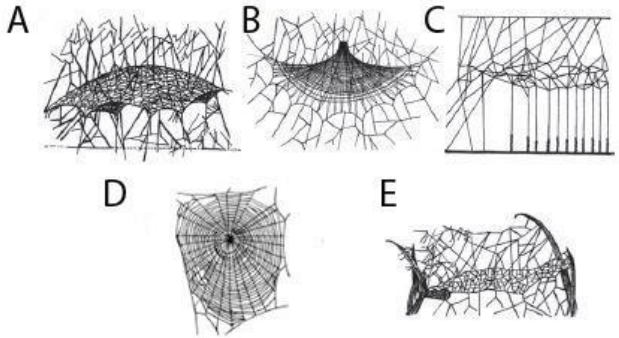


Inheritance sometimes makes designs more complex and inflexible.

We have an even bigger problem that inheritance cannot solve!

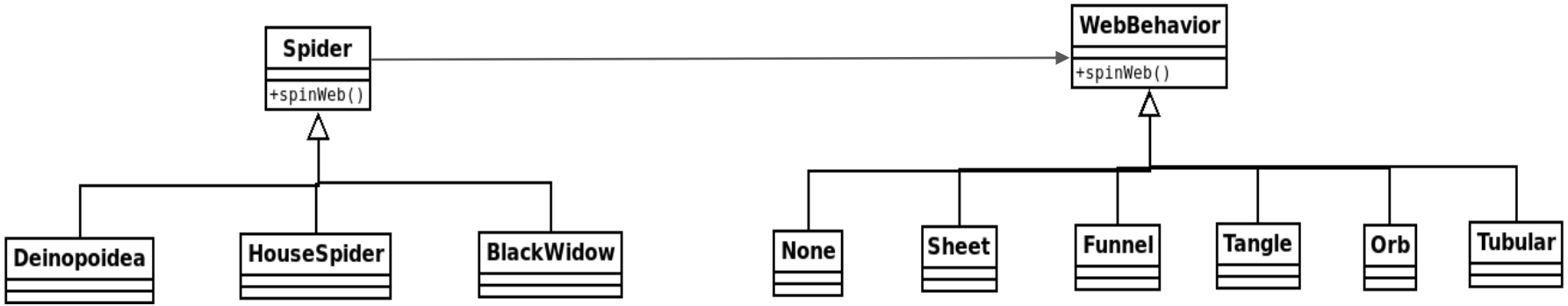
Various web types:

A - Sheet
B - Funnel
C - Tangle (Cob)
D - Orb
E - Tubular

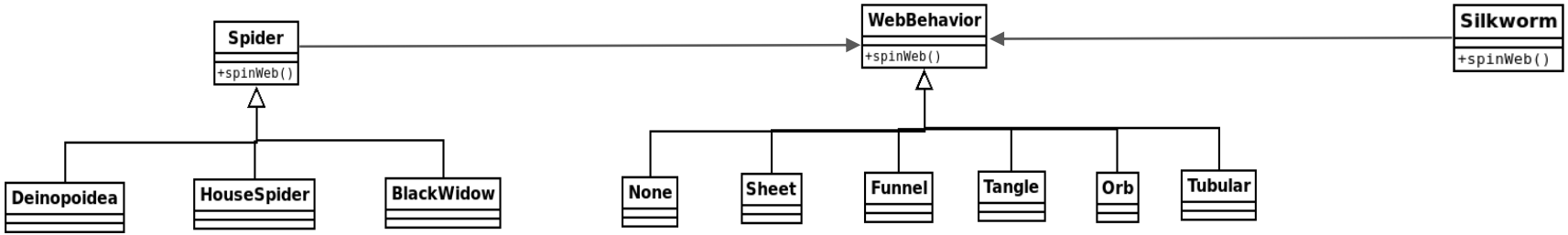


Imagine the inheritance hierarchy.

The solution: use **inheritance** with **polymorphism** and **composition**.



This has the added benefit of allowing other organisms to reuse the same functionality.



Take away: for complex systems, composition is the way to go!

This pattern is called the **Strategy Pattern** and examples are found throughout the biological world.

https://www.dofactory.com/net/strategy-design-pattern

Inheritance - The Tangled Web

- Inheritance alone can be problematic.

Polymorphism

- **Polymorphism** allows us to override functionality.
- Enables high cohesion and low coupling!

Design Patterns

- It is good design to favor **composition** over inheritance.
- The strategy pattern allows us to reuse functionality anywhere within a class hierarchy.

Design Principles

- Low coupling, high cohesion
- Readability and code style
- SOLID
  - Single responsibility principle
  - Open to extension, closed to modification
  - Liskov substitution principle
  - Interface segregation principle
  - Dependency inversion

Single Use Responsibility

A class should only have a single responsibility.

```
Journal journal("My Journal");
journal.add("First Entry");
journal.add("Second Entry");
journal.add("Third Entry");


// Use a separate class/entity for saving.
// Saving journals is not a base responsibility of a journal.
PersistenceManager().save(journal, "journal.txt");
```

Open to extension, closed to modification

Entities should be open for extension but closed for modification.

```
Product apple{"Apple", Color::Green, Size::Small};
Product tree{"Tree", Color::Green, Size::Large};
Product house{"House", Color::Blue, Size::Large};

ProductList all{apple, tree, house};

BetterFilter bf;
ColorSpecification green(Color::Green);

auto green_things = bf.filter(all, green);
for (auto& product : green_things)
  std::cout << product.name << " is green" << std::endl;

SizeSpecification big(Size::Large);
// green_and_big is a product specification
AndSpecification<Product> green_and_big{big, green};

auto green_big_things = bf.filter(all, green_and_big);
for (auto& product : green_big_things)
  std::cout << product.name << " is green and big" <<
std::endl;
```

Liskov Substitution Principle

Objects should be replaceable with instances of their subtypes without altering program correctness.

```
Rectangle r{5, 5};
process(r);

// Square (subtype of Rectangle)
// violates the Liskov Substitution
// Principle

Square s{5};
process(s);
```

Interface Segregation Principle

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

```
Printer printer;
Scanner scanner;
Machine machine(printer,scanner);
std::vector<Document>
documents{Document(std::string("
Hello")),

Document(std::string("Hello"))};
machine.print(documents);
machine.scan(documents);
```

Dependency Inversion

Dependencies should be abstract rather than concrete.
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Naïve example:

```
class Player
{
public:
    Player() {}

    void interactWith(Door *door)
    {
        if (door)
        {
            door->toggleOpen();
        }
    }
};
```

Dependency Inversion

Dependencies should be abstract rather than concrete.
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Naïve example:

```
class Door
{
public:
    Door() {}
    void toggleOpen()
    {
        // Open or close the door
        m_open = !m_open;
        if (m_open)
        {std::cout << "Door is open" <<
std::endl;}
        else
        {std::cout << "Door is closed" <<
std::endl;}
    }
private:
    bool m_open = false;
};
```

Dependency Inversion

Dependencies should be abstract rather than concrete.
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Issues:

Functionality works

Player can interact with doors in game

What if player wants to interact with other objects in the game?

We would need to write a separate method for every new object.

Dependency Inversion

Dependencies should be abstract rather than concrete.
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Solution:

Introduce an interface that the Door class can implement

Dependency Inversion

Dependencies should be abstract rather than concrete.
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Solution:

```
class InteractiveObject
{
public:
    virtual void interact() = 0;
    virtual ~InteractiveObject() =
default;
};
```

Dependency Inversion

Dependencies should be abstract rather than concrete.
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Solution:

```
class Door : public InteractiveObject
{
public:
    Door() {}
    void interact() override
    {
        // Open or close the door
        m_open = !m_open;
        if (m_open)
        {std::cout << "Door is open" <<
std::endl;}
        else
        {std::cout << "Door is closed" <<
std::endl;}
    }
private:
    bool m_open = false;
};
```

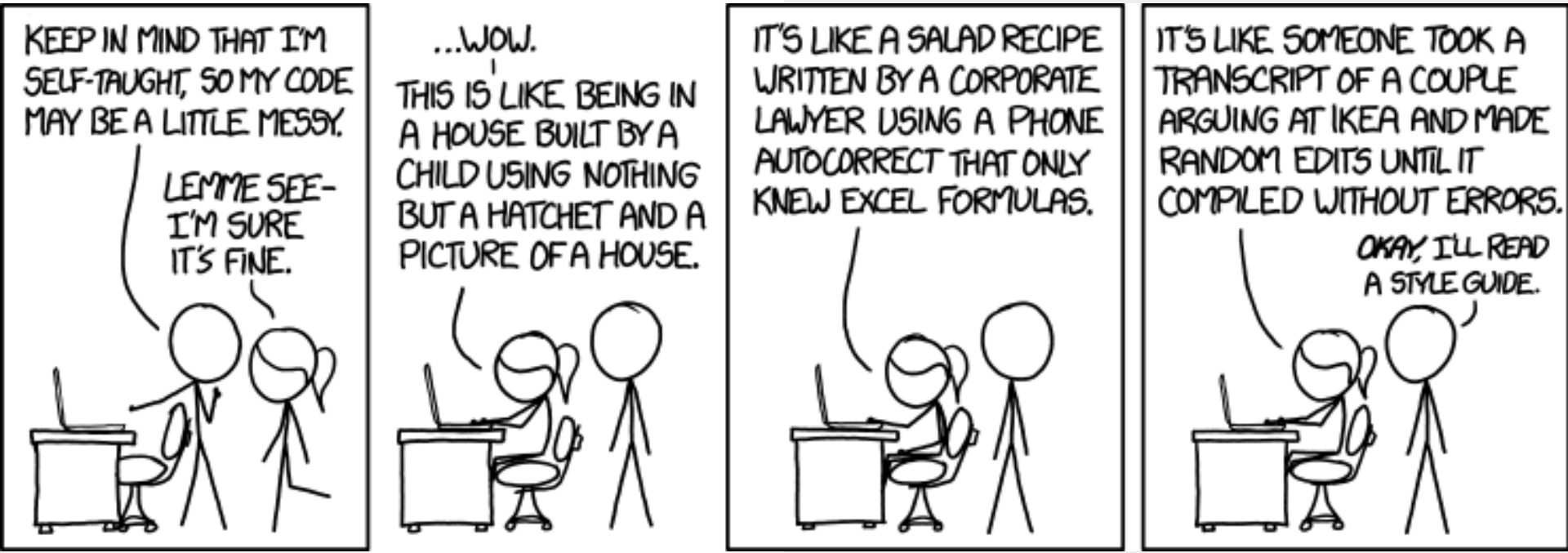
Dependency Inversion

Dependencies should be abstract rather than concrete.
High-level modules should not depend on low-level modules. Both should depend on abstractions.

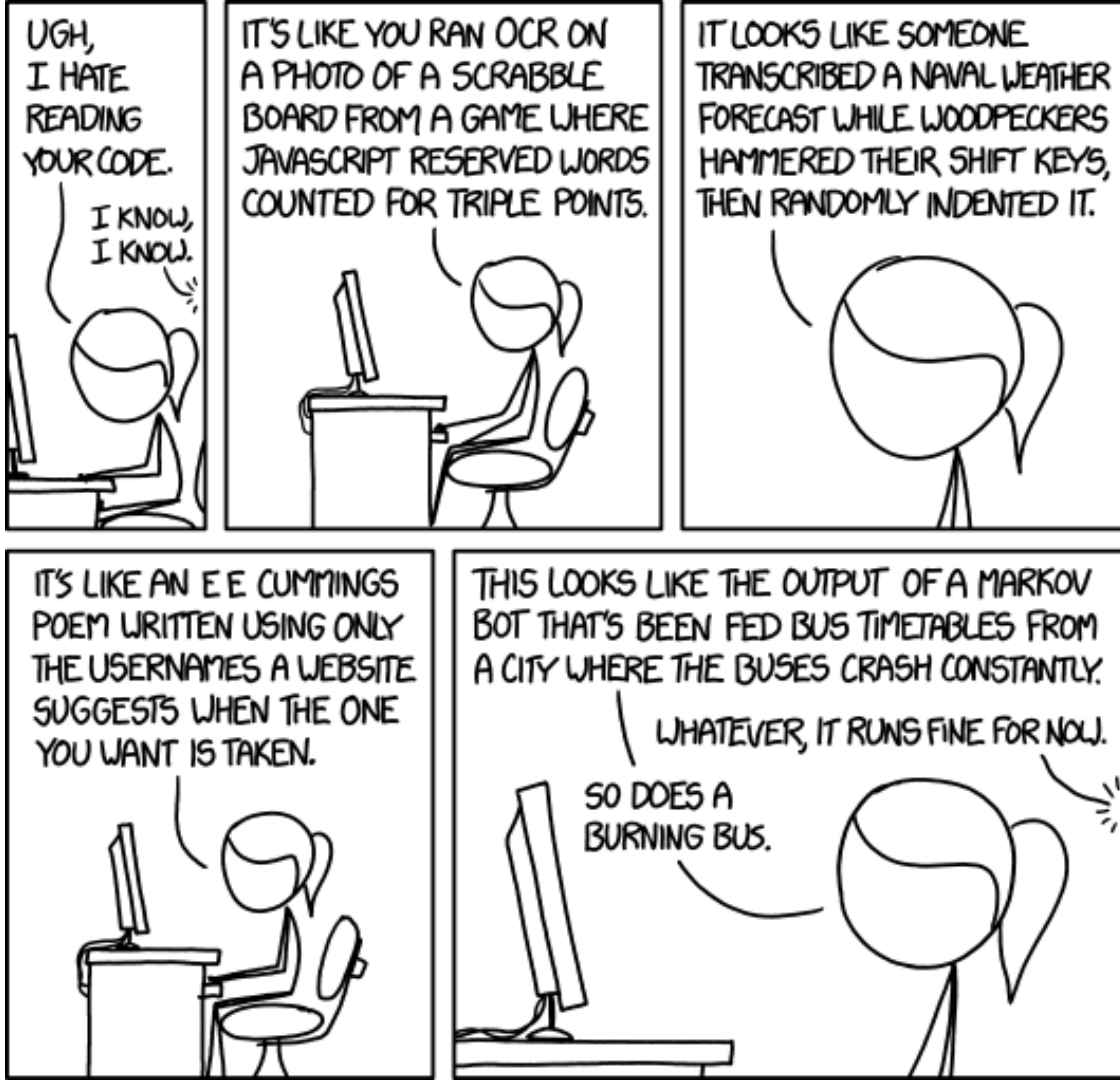
Solution:

```
class Player
{
public:
    Player() {}

    void interactWith(InteractiveObject *obj)
    {
        if (obj)
        {
            obj->interact();
        }
    }
};
```

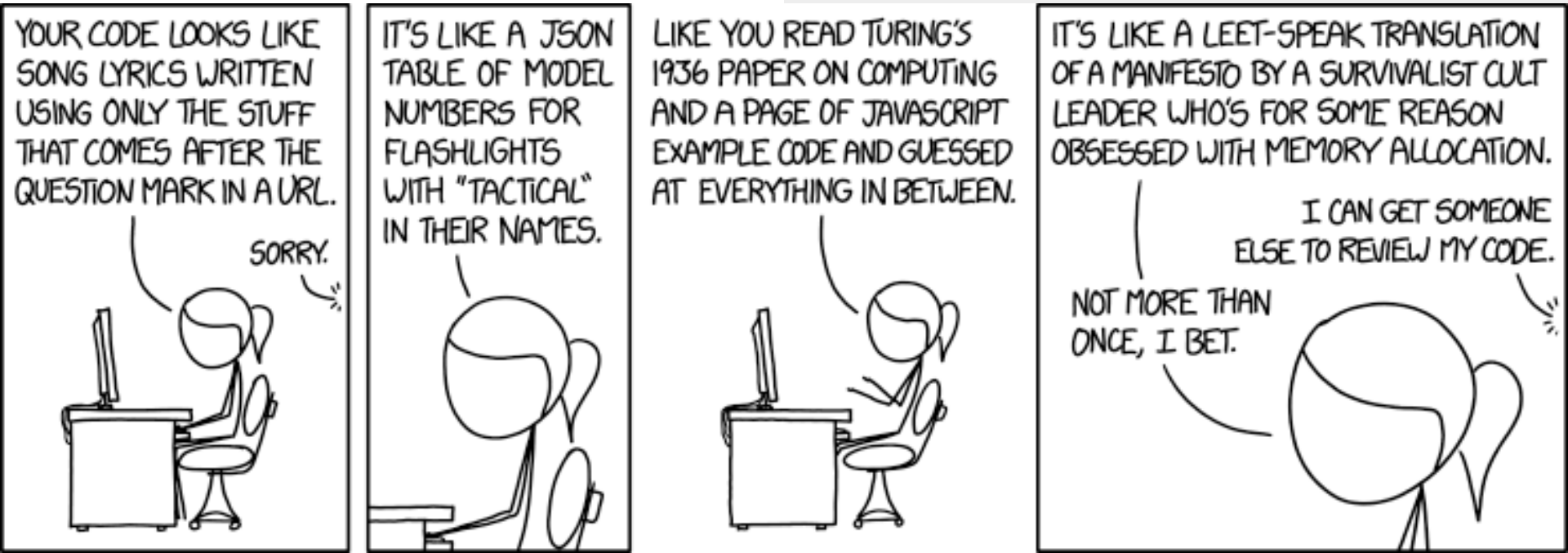
Code Style


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE- I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

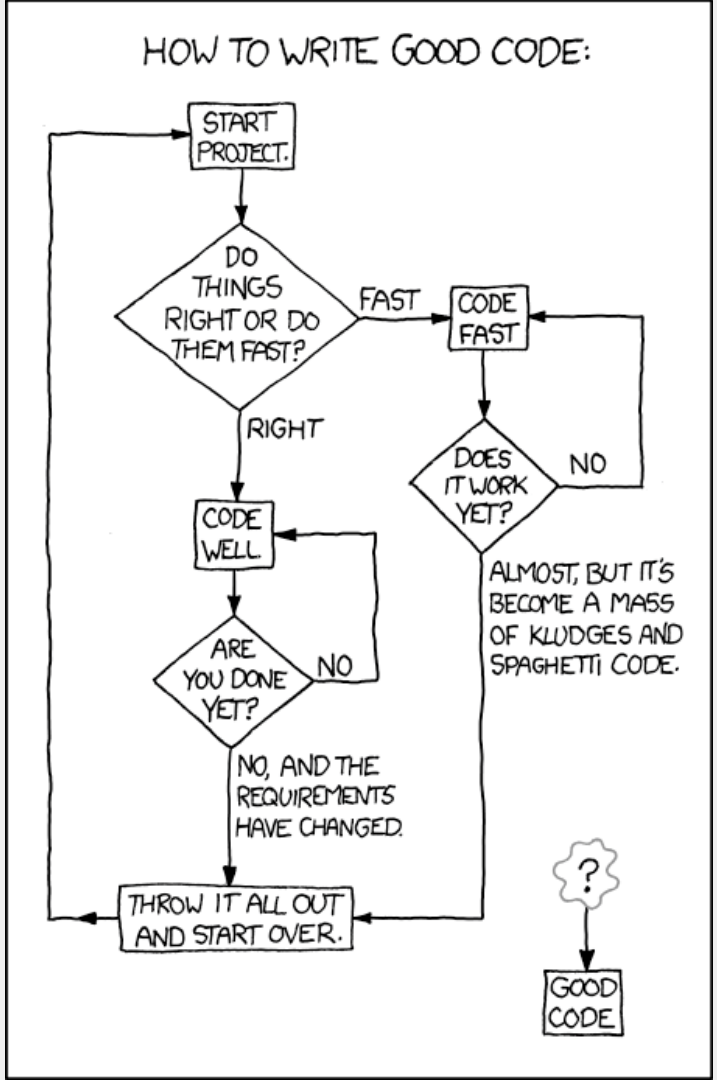
Code Style


UGH, I HATE READING YOUR CODE.
I KNOW, I KNOW.
IT'S LIKE YOU RAN OCR ON A PHOTO OF A SCRABBLE BOARD FROM A GAME WHERE JAVASCRIPT RESERVED WORDS COUNTED FOR TRIPLE POINTS.
IT LOOKS LIKE SOMEONE TRANSCRIBED A NAVAL WEATHER FORECAST WHILE WOODPECKERS HAMMERED THEIR SHIFT KEYS, THEN RANDOMLY INDENTED IT.
IT'S LIKE AN E E CUMMINGS POEM WRITTEN USING ONLY THE USERNAMES A WEBSITE SUGGESTS WHEN THE ONE YOU WANT IS TAKEN.
THIS LOOKS LIKE THE OUTPUT OF A MARKOV BOT THAT'S BEEN FED BUS TIMETABLES FROM A CITY WHERE THE BUSES CRASH CONSTANTLY.
WHATEVER, IT RUNS FINE FOR NOW.
SO DOES A BURNING BUS.

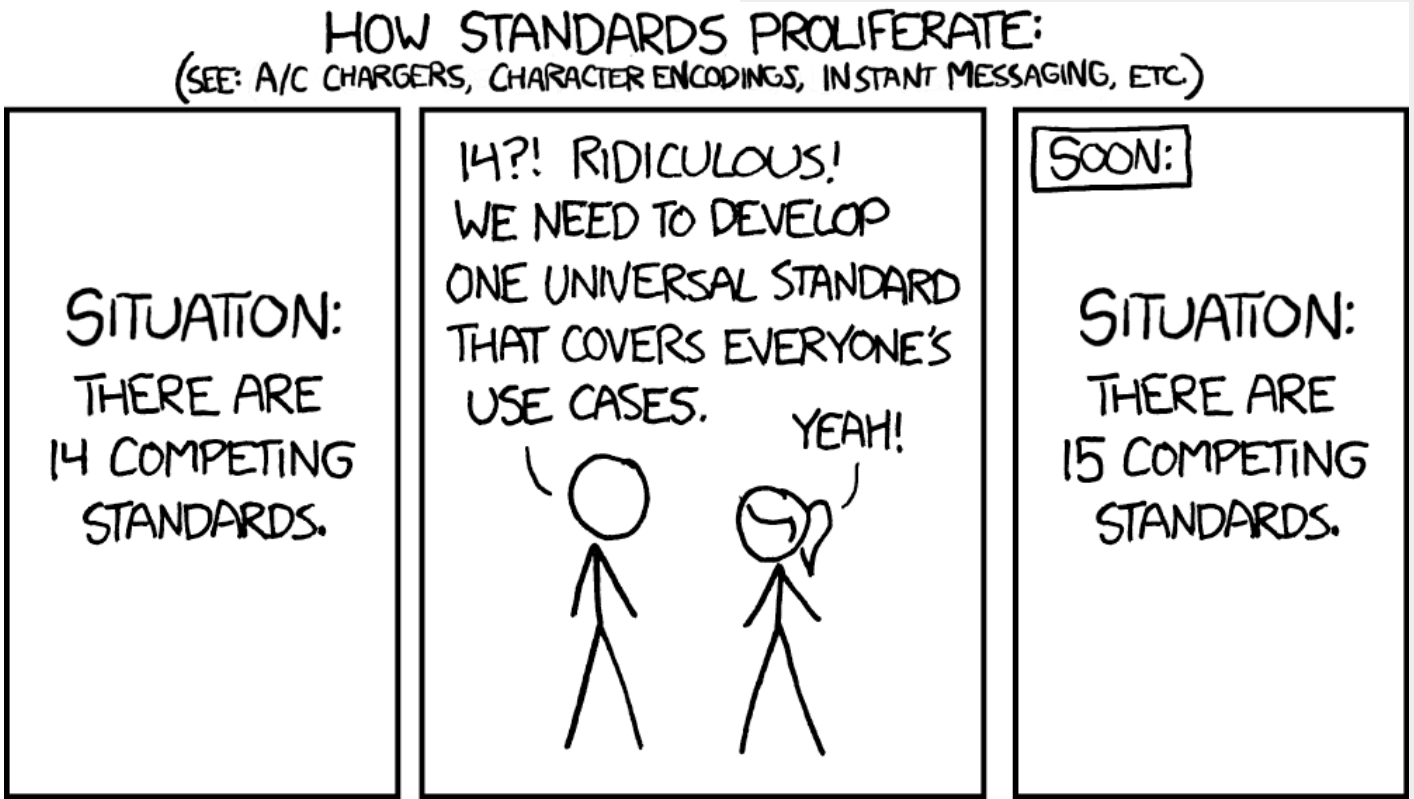
Code Style


YOUR CODE LOOKS LIKE SONG LYRICS WRITTEN USING ONLY THE STUFF THAT COMES AFTER THE QUESTION MARK IN A URL.
SORRY.
IT'S LIKE A JSON TABLE OF MODEL NUMBERS FOR FLASHLIGHTS WITH "TACTICAL" IN THEIR NAMES.
LIKE YOU READ TURING'S 1936 PAPER ON COMPUTING AND A PAGE OF JAVASCRIPT EXAMPLE CODE AND GUESSED AT EVERYTHING IN BETWEEN.
IT'S LIKE A LEET-SPEAK TRANSLATION OF A MANIFESTO BY A SURVIVALIST CULT LEADER WHO'S FOR SOME REASON OBSESSED WITH MEMORY ALLOCATION.
NOT MORE THAN ONCE, I BET.
I CAN GET SOMEONE ELSE TO REVIEW MY CODE.

Code Style


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
?
THROW IT ALL OUT AND START OVER.
GOOD CODE

Code Style


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.