OBJECT ORIENTED DESIGN PATTERNS

Design Patterns are

- general repeatable solutions to a commonly occurring problems in software design.

Design Patterns are

- NOT premade software solutions
- NOT just for Java

Design Principles

- Design patterns are based on standard software design principles
  - Open Closed Principle
  - Dependency Inversion Principle
  - Interface Segregation Principle
  - Single Responsibility Principle
  - Liskov’s Substitution Principle

Design Patterns

- Fall into three broad categories
  - Creational Patterns -- provide ways to instantiate single objects or groups of related objects
  - Behavioral Patterns -- define the manners of communication between classes and objects
  - Structural Patterns -- provide a manner to define relationships between classes or objects

Creational Patterns

- Deal with the creation of objects and used when the basic means of object creation
  - could be problematic
  - or increase code complexity

Common Creational Patterns

- Builder
- Prototype
- Singleton
- Factory

Singleton

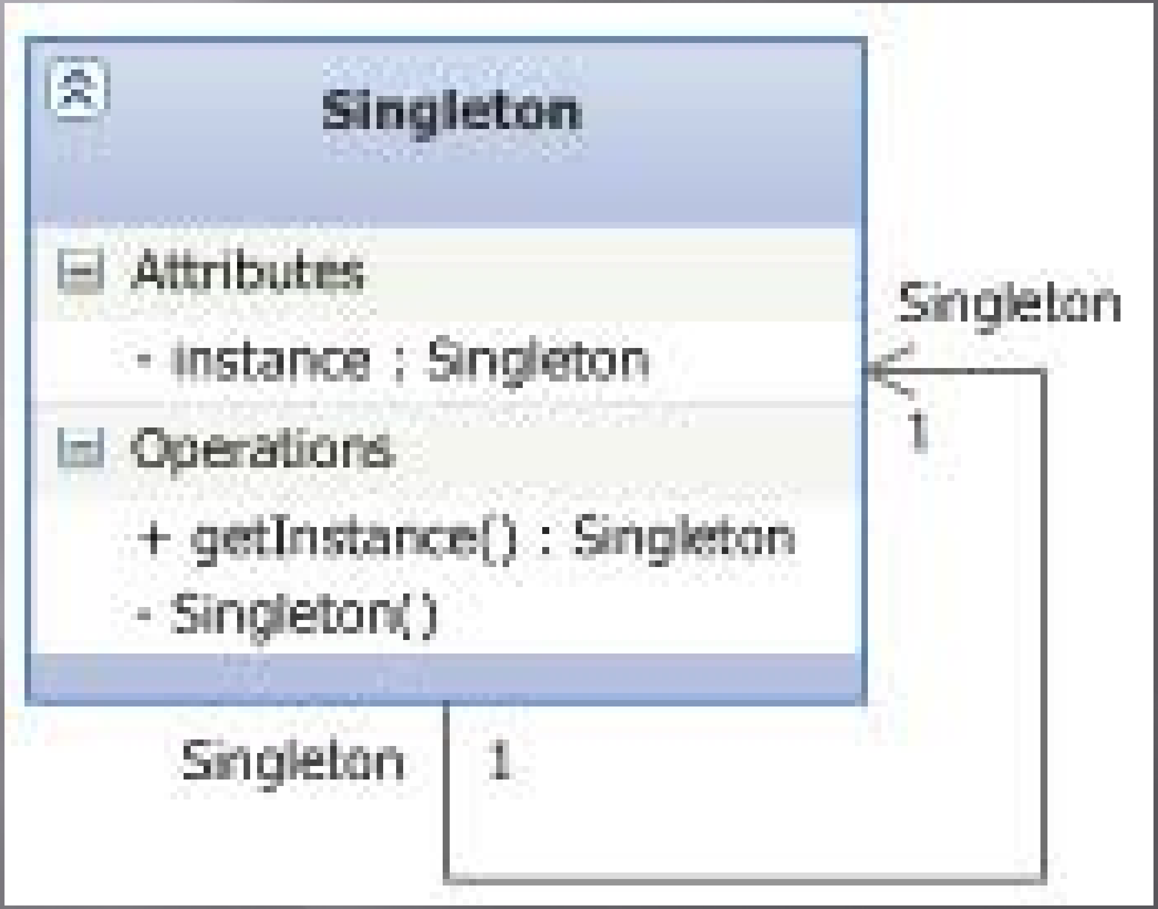
- Simple pattern
- Used to ensure that a class can only have one concurrent instance. Whenever additional objects of a singleton class are required, the previously created, single instance is provided.
- Creates centralized management of internal or external resources and provides a global point of access

Singleton

- Addresses these problems
  - How to create a class with only one instance?
  - How can that instance be accessed easily?
  - How can a class control its instantiation?

- Allow the programmers to hide the constructor of the class by defining a public static operation that returns just one instance of the class.

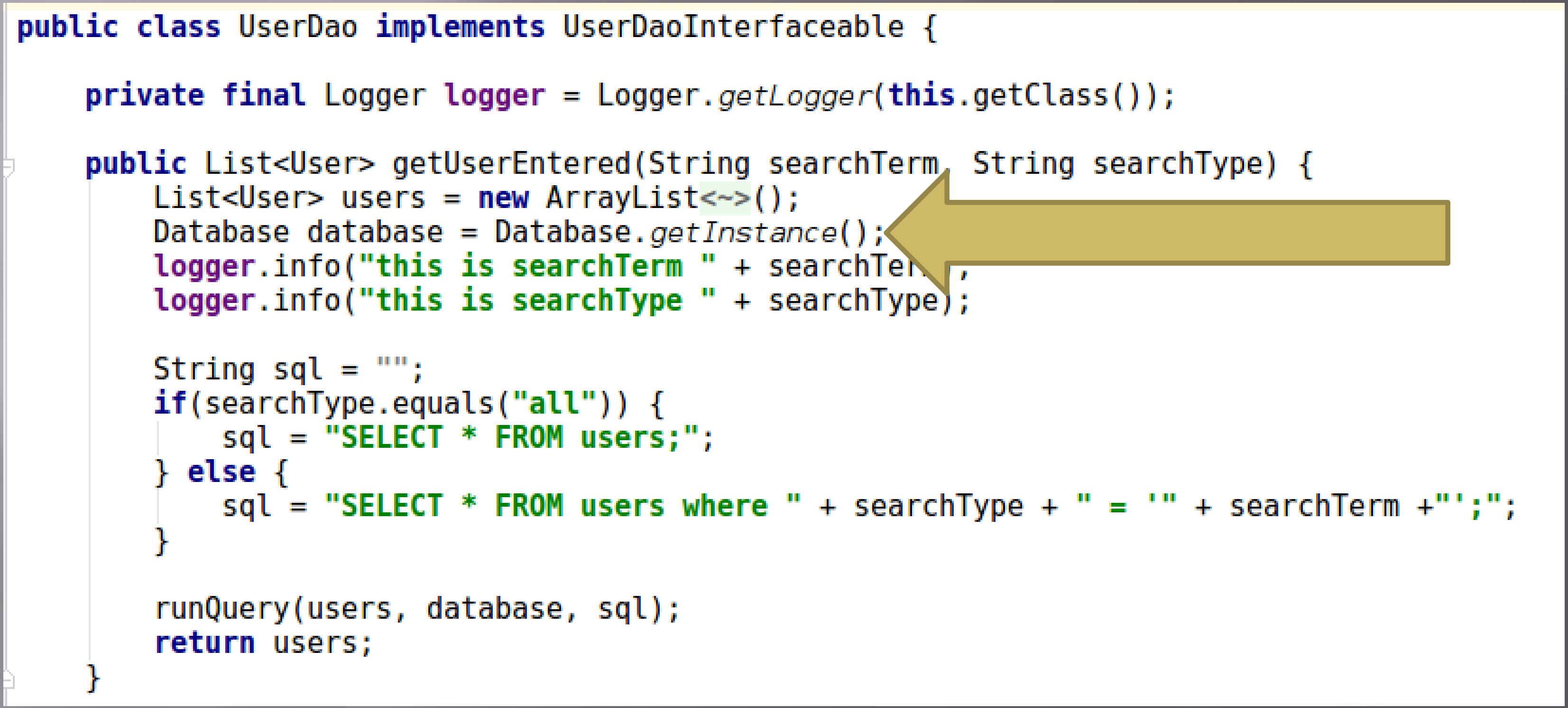
UML Diagram


Singleton
Attributes
- instance : Singleton
Operations
+ getInstance() : Singleton
- Singleton()
Singleton
1
Singleton
1

So what’s the point?

- Restricts access to constructor
- Creates only a single instance of object
- Insures that any code accessing the object will see the same instance variables in that object
- Eliminates unnecessary object creation
- Conserves system resources

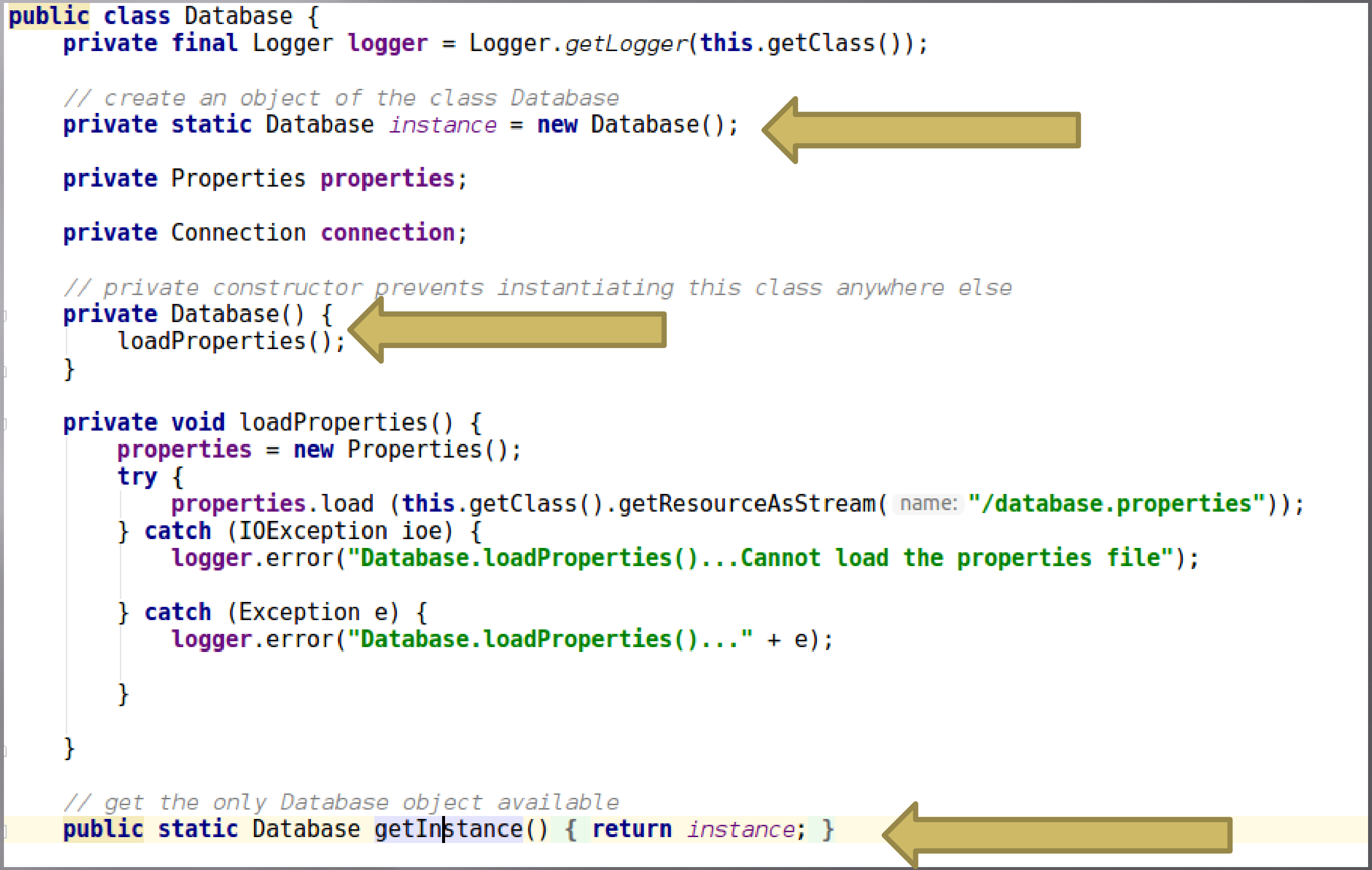
An Example . . .



```
public class UserDao implements UserDaoInterfaceable {

    private final Logger logger = Logger.getLogger(this.getClass());

    public List<User> getUserEntered(String searchTerm, String searchType) {
        List<User> users = new ArrayList<~>();
        Database database = Database.getInstance();
        logger.info("this is searchTerm " + searchTerm);
        logger.info("this is searchType " + searchType);

        String sql = "";
        if(searchType.equals("all")) {
            sql = "SELECT * FROM users;";
        } else {
            sql = "SELECT * FROM users where " + searchType + " = '" + searchTerm +"';";
        }

        runQuery(users, database, sql);
        return users;
    }
```

Uses Database database = Database.getInstance();
Rather than Database database = new DataBase();

Meanwhile back in the Database class . . .



```java
public class Database {
    private final Logger logger = Logger.getLogger(this.getClass());

    // create an object of the class Database
    private static Database instance = new Database();

    private Properties properties;

    private Connection connection;

    // private constructor prevents instantiating this class anywhere else
    private Database() {
        loadProperties();
    }

    private void loadProperties() {
        properties = new Properties();
        try {
            properties.load (this.getClass().getResourceAsStream( name: "/database.properties"));
        } catch (IOException ioe) {
            logger.error("Database.loadProperties()...Cannot load the properties file");

        } catch (Exception e) {
            logger.error("Database.loadProperties()..." + e);

        }

    }

    // get the only Database object available
    public static Database getInstance() { return instance; }
```

Things of note– 1) Constructor is private, 2) object is constructed when instance variable created, 3) access to the object instance is by a method

Volunteers???

Summary of Singleton

- Allows for the creation of a single instance of an object
- Since there is only one instance of the object, the consistency of instance variables is assured in multithreaded applications
- System resources are conserved by minimizing creation of new objects (for example database access for online resources)

Singleton Possible Flaws

- Can be used where it is unnecessary
- Can create unneeded restrictions
- Introduces a global state into an application

Factory Pattern

- Has three common related types
  - Factory
  - Factory Method,
  - Abstract Factory Method

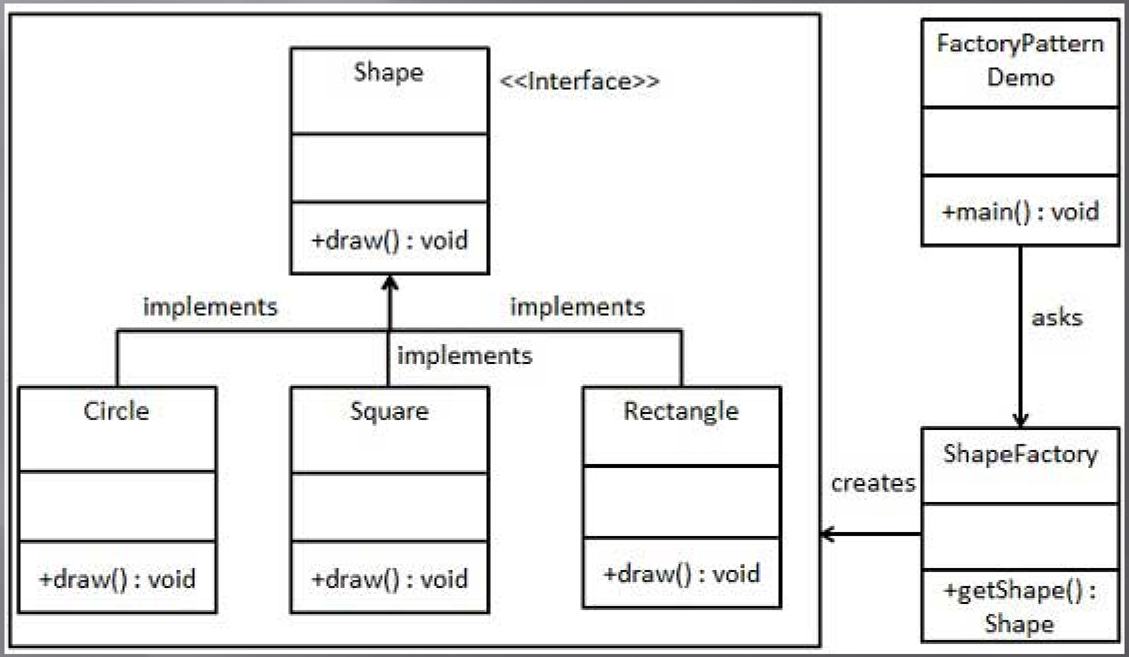
Factory Pattern

- Most commonly used pattern
- Creates object without exposing the creation logic
- Allows access to new object using a common interface

Factory

- Used when a super class with multiple sub-classes needs to return a sub-class based on input
- Takes the responsibility of instantiation of a class from client program

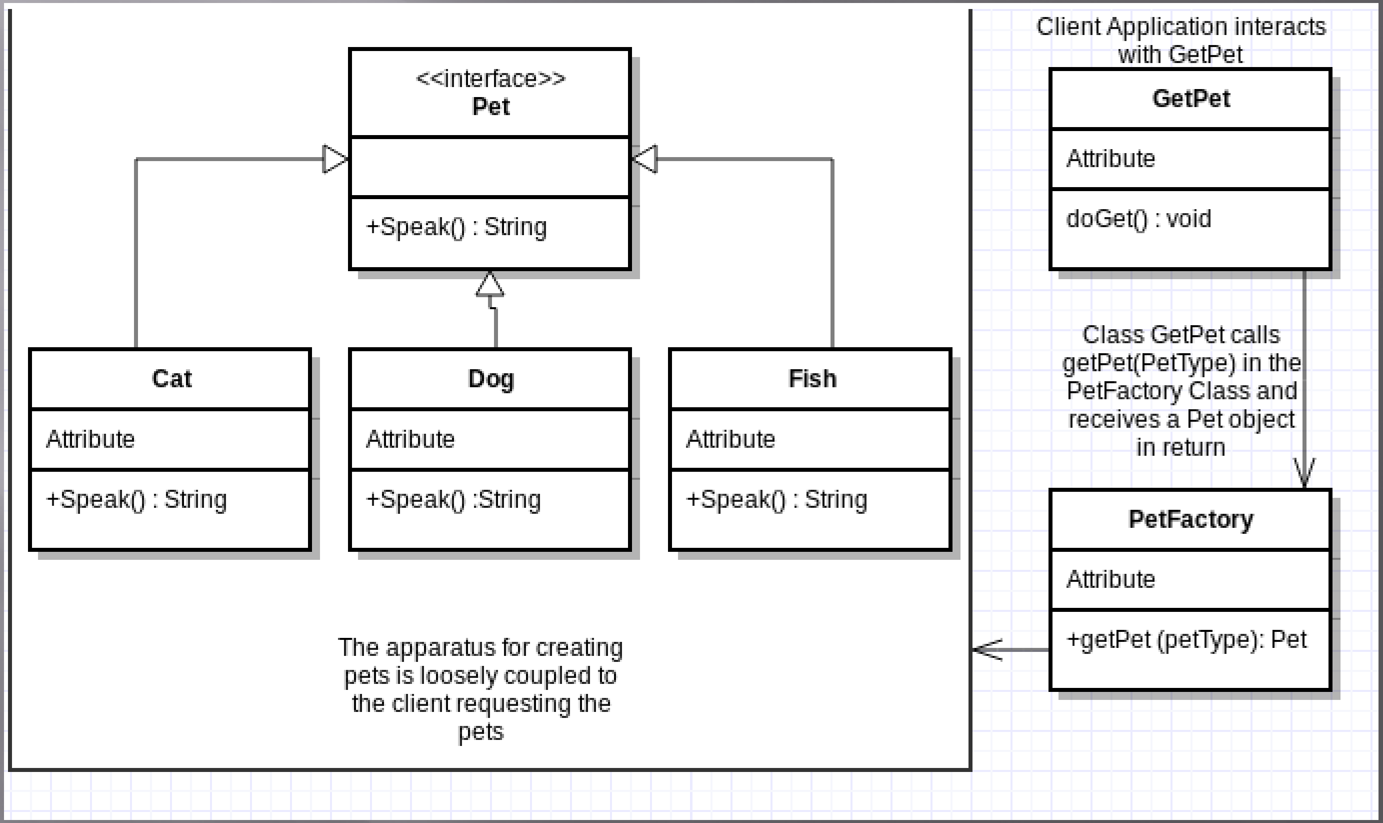
UML Diagram


Shape
<<Interface>>
+draw() : void
implements
implements
implements
Circle
+draw() : void
Square
+draw() : void
Rectangle
+draw() : void
FactoryPattern Demo
+main() : void
asks
ShapeFactory
+getShape() : Shape
creates

Why Use Factory?

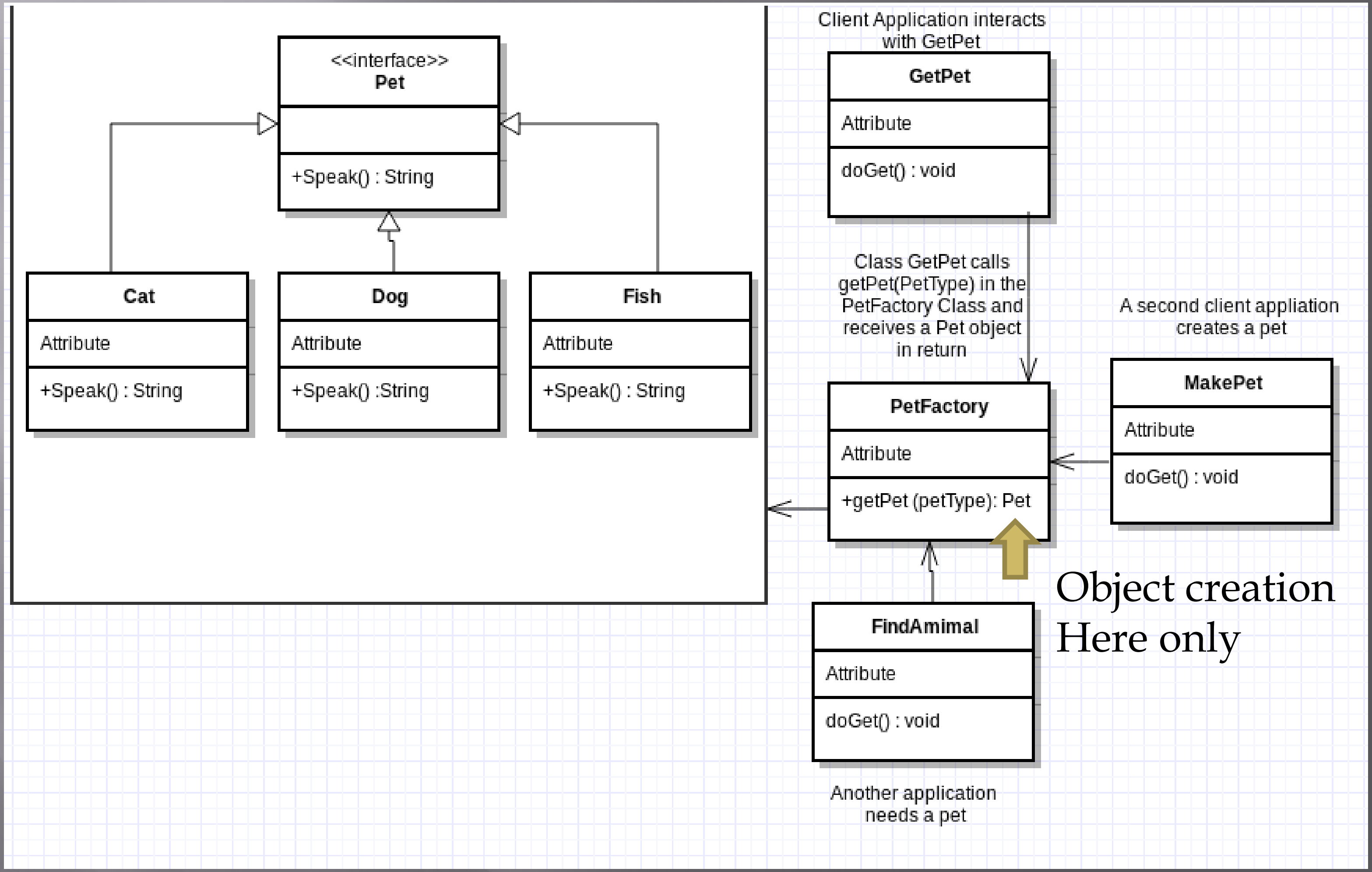
- Provides a way to code for interface rather than implementation.
- Removes object instantiation from client code.
  - Making code more robust
  - More loosely coupled
  - Easier to extend

Pet Factory UML

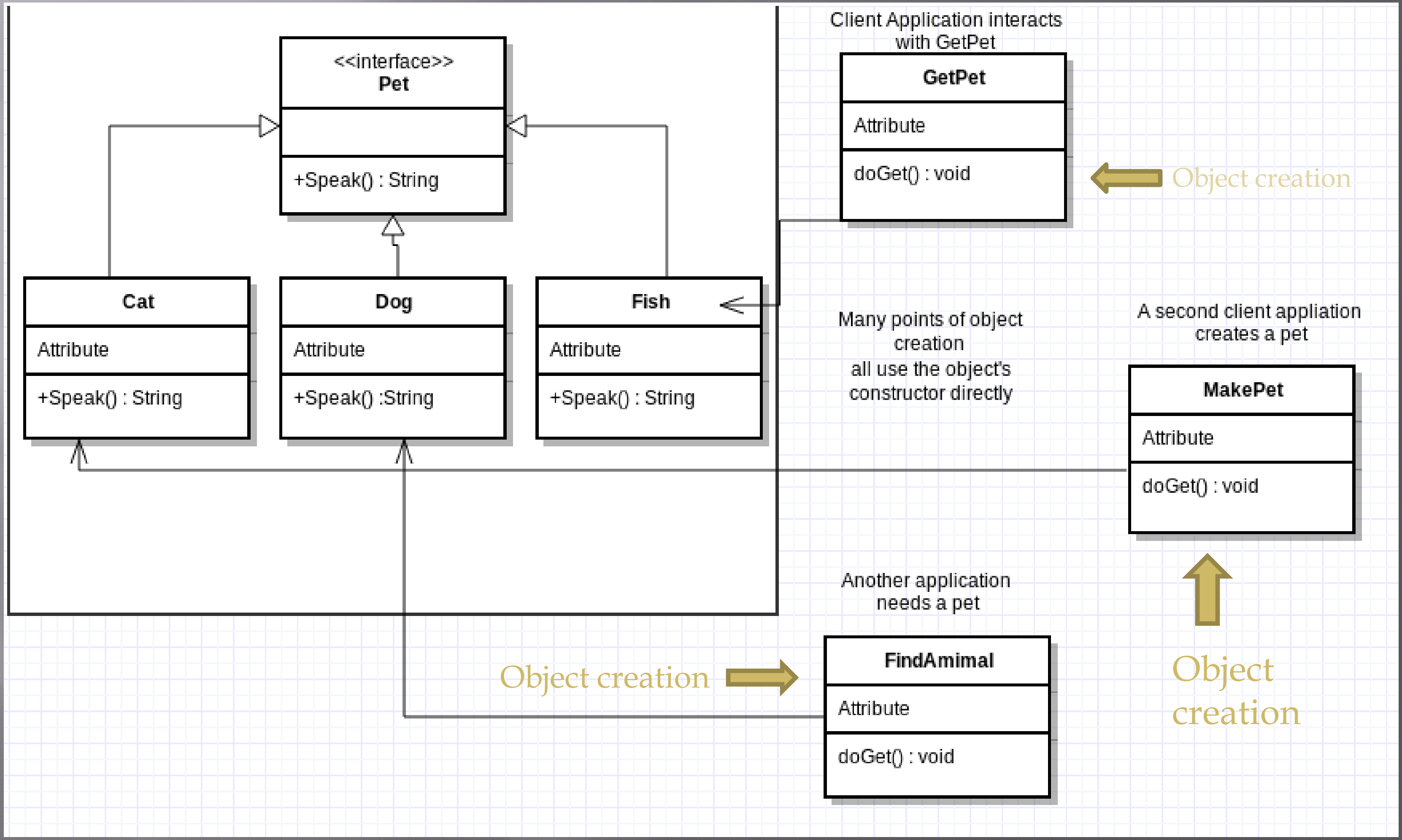


<<interface>>
Pet
+Speak() : String
Cat
Attribute
+Speak() : String
Dog
Attribute
+Speak() :String
Fish
Attribute
+Speak() : String
The apparatus for creating pets is loosely coupled to the client requesting the pets
Client Application interacts with GetPet
GetPet
Attribute
doGet() : void
Class GetPet calls getPet(PetType) in the PetFactory Class and receives a Pet object in return
PetFactory
Attribute
+getPet (petType): Pet

Why Does it Matter?



Without Factory



Volunteers???

A Simple Demo
Pet Factory