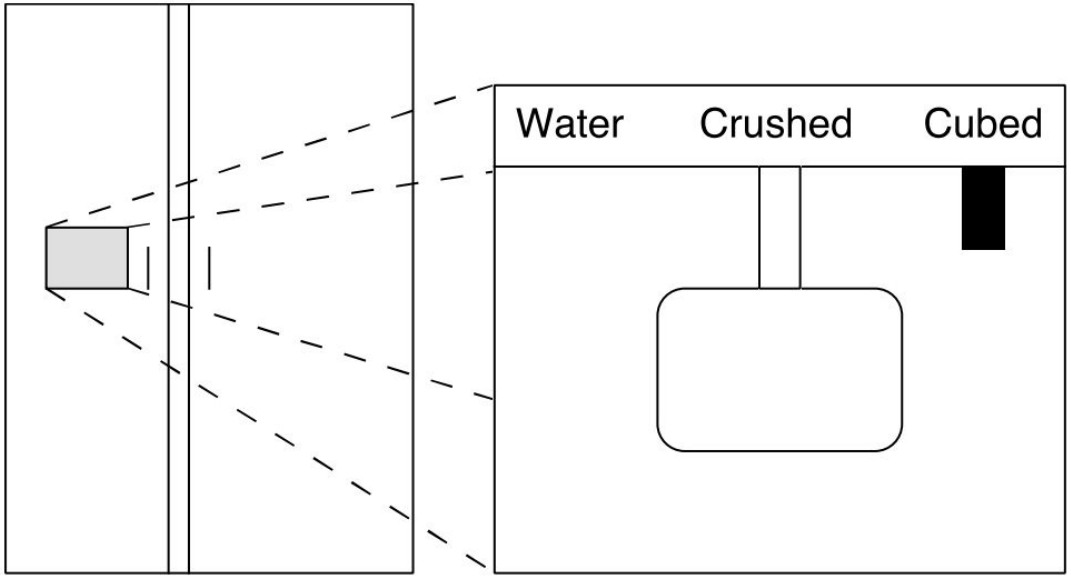
CSCI 2120:
Software Design & Development II

OOP: Design By Contract

Systems Thinking

- A ***system*** is any part of anything that you want to think about as an indivisible unit
- An ***interface*** is a description of the "boundary" between a system and everything else, that also describes how to think about that system as a unit
- A ***subsystem*** (***component***) is a system that is used inside, i.e., as a part of, another system — a relative notion!

Example: Ice/Water Dispenser



Select water, crushed ice, or cubed ice.
Place a glass against the pad and push.

People's Roles wrt Systems

- A ***client*** is a person (or a role played by some agent) viewing a system "from the outside" as an indivisible unit
- An ***implementer*** is a person (or a role played by some agent) viewing a system "from the inside" as an assembly of subsystems/components

Describing Behavior: Part 1

- One side of the coin: ***information hiding*** is a technique for describing system behavior in which you *intentionally leave out* "internal implementation details" of the system

Describing Behavior: Part 2

- Other side of the coin (and a necessary consequence of information hiding): ***abstraction*** is a technique in which you create a *valid cover story* to counteract the effects of hiding some internal implementation details
  - Presumably the hidden information is relevant to the system behavior, so even if you hide it you still need to account for its presence!

Overview of Design-by-Contract

- Also known as ***programming-to-the-interface***
- Articulated clearly only in the 1980s
- Design-by-contract has become *the standard policy* governing "separation of concerns" across modern software engineering
- This is how software components are really used…

Structure of a Method Contract

- Each method has:
  - A ***precondition*** (***requires clause***) that characterizes the responsibility of the program that ***calls*** (***uses***) that method (client code)
  - A ***postcondition*** (***ensures clause***) that characterizes the responsibility of the program that ***implements*** that method (implementation code in the method body)

Meaning of a Method Contract

- If its precondition is true when a method is called, then the method will ***terminate*** — return to the calling program — and the postcondition will be true when it does return
- If its precondition is not true when a method is called, then the method may do anything (including not terminate)

Responsibilities and Rewards

- Responsibility: Making sure the ***precondition*** is true when a method is called is the responsibility of the ***client***
- Reward: The client may assume the postcondition is true when the method returns

Responsibilities and Rewards

- Responsibility: Making sure the ***postcondition*** is true when a method returns is the responsibility of the ***implementer***
- Reward: The implementer may assume the precondition is true when the method is called

Javadoc

- The standard documentation technique for Java is called ***Javadoc***
- You place special ***Javadoc comments*** enclosed in `/** ... */` in your code, and the ***javadoc tool*** generates nicely formatted web-based documentation from them

APIs

- The resulting documentation is known as the ***API (application programming interface)*** for the Java code to which the Javadoc tags are attached

- The API for the Oracle components is at: https://docs.oracle.com/en/java/javase/17/docs/api/index.html

APIs

- The resulting documentation is known as the ***API (application programming interface)*** for the Java code to which the Javadoc tags are attached
- The API for [illegible] at:

  http://web.cse.ohi[illegible]

The word ***interface*** has two related but distinct meanings:

- a unit of Java code that contains Javadoc comments used to produce documentation
- the resulting documentation

Javadoc

- Javadoc is a system for generating documentation from Java code.
- The *javadoc* binary utility has been included in the JDK since the initial release.
- Provides an easy-to-use way to document code that in a standard, uniform way.

Using Javadoc

- Javadoc content is always included within special comment blocks.

```
/**
      [Javadoc stuff here…]
 */
```

Example

```
/**
 * A class that demonstrates Javadoc.
 *
 * Here is a more detailed paragraph describing this class.
 */
public class MyClass {

        /**
         * This method does a lot of things.
         */
        public void myMethod(){
                /* Do lots of things here. */
        }
}
```

Example of a Contract

```
/**
 * ...
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return th approximate square root of x
 * @requires
 * x > 0  and  epsi     > 0
 * @ensures <pre>
 * sqrt >= 0  and
 * [sqrt is within relati
 *  of the actual square
 * </pre>
 */
private static double sqrt(double x,
    double epsilon)
```

The ***Javadoc tag*** `@param` is needed for each formal parameter; you describe the parameter's role in the method.

Example of a Contract

```
/**
 * ...
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @requires
 * x > 0  and  ep          > 0
 * @ensures <pre>
 * sqrt >= 0  and
 * [sqrt is within relati
 *  of the actual square
 * </pre>
 */
private static double sqrt(double x,
    double epsilon)
```

The ***Javadoc tag*** `@return` is needed if the method returns a value; you describe the returned value.

Example o

```
/**
 * ...
 * @param x number to ta
 * @param epsilon all
 * @return the ar      imate
 * @requires
 * x > 0  and  epsilon > 0
 * @ensures <pre>
 * sqrt >= 0  and
 * [sqrt is within relative error epsilon
 *  of the actual square root of x]
 * </pre>
 */
private static double sqrt(double x,
    double epsilon)
```

The ***Javadoc tag*** `@requires` introduces the precondition for the `sqrt` method.

Example o

```
/**
 * ...
 * @param x number to tak
 * @param epsilon allo
 * @return the appr    ate
 * @requires
 * x > 0  and  epsilon > 0
 * @ensures <pre>
 * sqrt >= 0  and
 * [sqrt is within relative error epsilon
 *  of the actual square root of x]
 * </pre>
 */
private static double sqrt(double x,
    double epsilon)
```

The ***Javadoc tag*** `@ensures` introduces the postcondition for the `sqrt` method.

Example o

```
/**
 * ...
 * @param x number to tak
 * @param epsilon allow
 * @return the approxi  te
 * @requires
 * x > 0  and  epsilon > 0
 * @ensures <pre>
 * sqrt >= 0  and
 * [sqrt is within relative error epsilon
 *  of the actual square root of x]
 * </pre>
 */
private static double sqrt(double x,
    double epsilon)
```

Javadoc comments may contain HTML-like tags; e.g., <pre> … </pre> means spacing and line-breaks are retained in generated documentation.

Example Contract (Abbreviated)

```
/**
 * ...
 * @requires
 * x > 0  and  epsilon > 0
 * @ensures
 * sqrt >= 0
 * [sqrt is wi
 *  of the act
 */
private static
    double epsilon)
```

This is the precondition, indicating that the *arguments* passed in for the *formal parameters* `x` and `epsilon` both must be positive before a client may call `sqrt`.

Example Contract (Abbreviated)

```
/**
 * ...
 * @requires
 * x > 0  and
 * @ensures
 * sqrt >= 0  and
 * [sqrt is within relative error epsilon
 *  of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

This is the postcondition, indicating that the *return value* from `sqrt` is non-negative and … what does the rest say?

Using a Method Contract

- A static method's contract refers to its formal parameters, and (only if it returns a value, not `void`) to the name of the method (which stands for the return value)
- To determine whether the precondition and postcondition are true for a particular client call:
  - The model values of the *arguments* are substituted for the respective formal parameters
  - The model value of the *result returned by the method* is substituted for the method name

A Method Body

```
private static double sqrt(double x,
    double epsilon) {
  assert x > 0.0 :
    "Violation of: x > 0";
  assert epsilon > 0.0 :
    "Violation of: epsilon > 0";
  // rest of body: compute the square root
}
```

A Method Body

```
private static double sqrt(double x,
    double epsilon) {
  assert x > 0.0 :
    "Violation of: x > 0";
  assert epsilon > 0.0 :
    "Violation of: epsilon > 0";
  // rest of body
}
```

The **assert** statement in Java checks whether a condition (an ***assertion***) is true; if it is not, it stops execution and reports the message after the colon.

A Method Body

```
private static double sqrt(double x,
    double epsilon) {
  assert x > 0.0 :
    "Violation of: x > 0";
  assert epsilon > 0.0 :
    "Violation of: epsilon > 0";
  // rest of body
}
```

But why are there **`assert`** statements in this method body to *check* what the implementer is supposed to *assume*?

Checking a Precondition

- During ***software development***, it is a **best practice** to check assumptions with `assert` when it is easy to do so
  - This checking can be turned on and off (on by using the "-ea" argument to the JVM)
  - When turned off, `assert` is documentation
- Preconditions generally are easy to check; postconditions generally are not easy to check

A Misconception

- A common misconception is that using `assert` statements to check preconditions contradicts design-by-contract principles
- It does not, because the advice is not to ***deliver*** software with assertion-checking turned on, but rather to ***develop*** software with assertion-checking turned on — to help catch *your* mistakes, not the client's!

JavaDocs

Javadoc Tags

- Javadoc uses **tags** to annotate certain properties of classes or methods:
- Class Tags:

Tag	Usage
@author [name]	Tag the author(s) of the class.
@version [version]	Show the version of the class.
@since [text]	When the method/class was introduced.
@see [reference]	List reference to another class/method.

Javadoc Tags (cont.)

- Method Tags:

Tag	Usage
@param [name] [desc.]	Describes a parameter.
@return [desc.]	Describes the returned value.
@throws [exception] [desc.]	Describes an exception that may be thrown.
@deprecated	Indicates the method is deprecated.
{@inheritDoc}	Automatically inserts super class Javadoc properties.

Styling Javadoc

- HTML Tags can be used in Javadoc comments:

Tag	Usage
<p>	Starts a new paragraph.
<code>…</code>	Formats part of the Javadoc as code.

Generating Javadoc

- You can generate HTML from your Javadoc'd source code by invoking the following tool:

javadoc -d <dir> <java-file-1> …

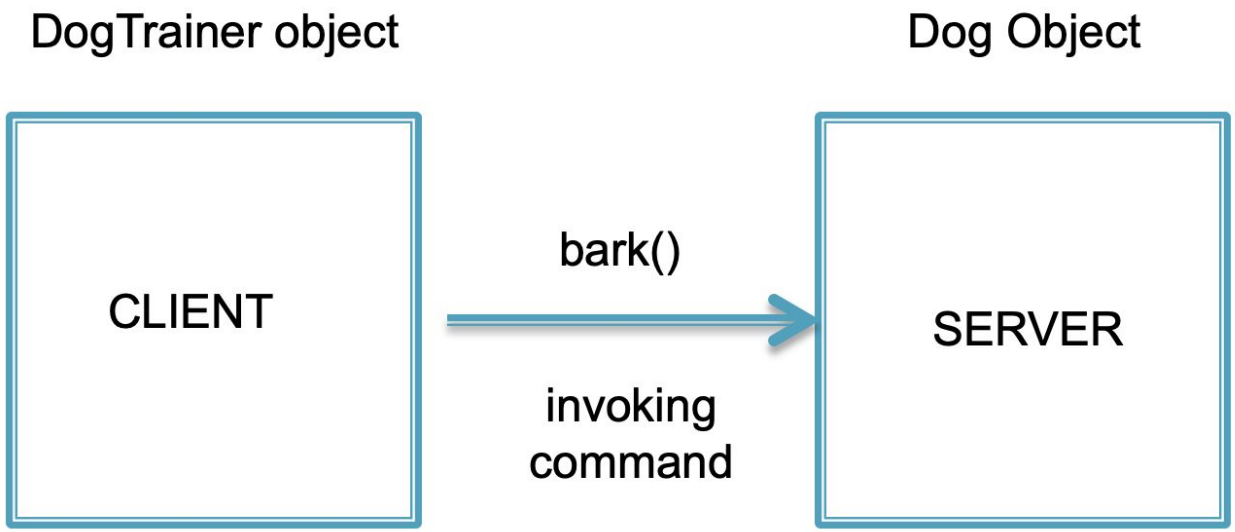
Client / Server Model

- We can think of the invocation of a method as a transaction between:
  - CLIENT: the portion of the code that calls the method
  - SERVER: the portion of the code that executes the method.
  - For example:

    Suppose we have a class Dog that has a method bark() and we also have a class DogTrainer that invokes the bark() method of a Dog object.

  - CLIENT: DogTrainer object
  - SERVER: Dog object that is running its bark method

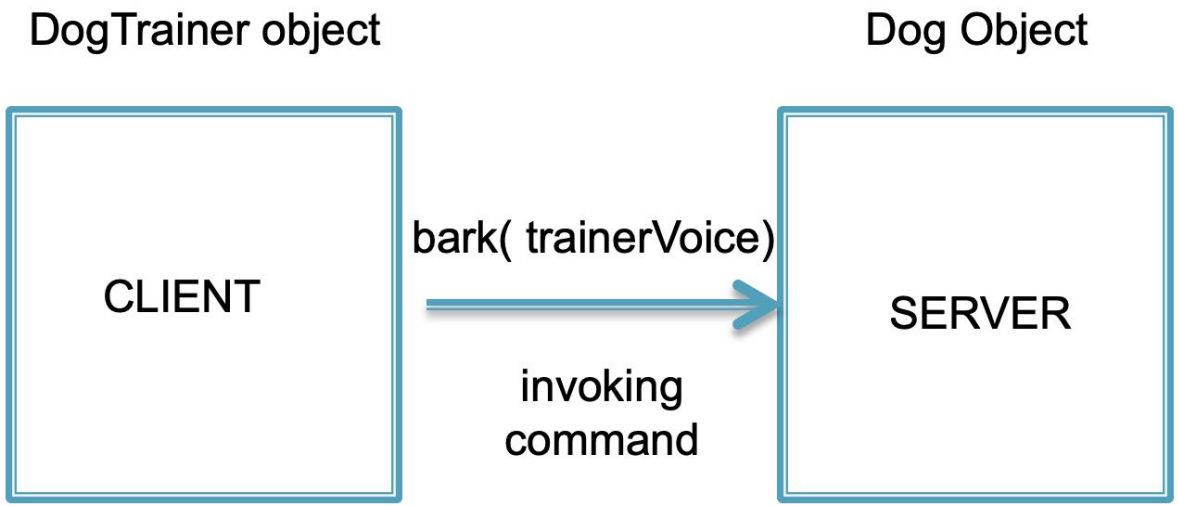
Client / Server Model



Client / Server Model

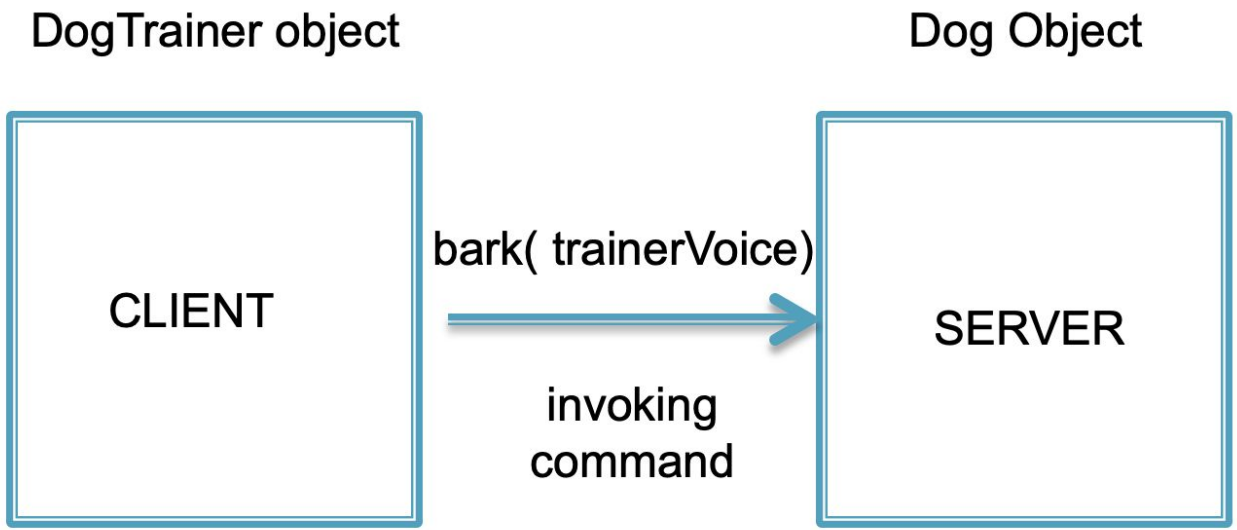
- What is the point of thinking about it this way?
  - It has to do with checking that the invocation of the method "make sense" in terms of the data that goes in and the data that comes out
  - Remember: Data "goes in" *via* parameters and "comes out" *via* a return value
  - Let's modify our example a bit and imagine that there is a parameter to the bark method that represents how forcefully that command is given (i.e., the decibel level of the trainer's voice)

Client / Server Model



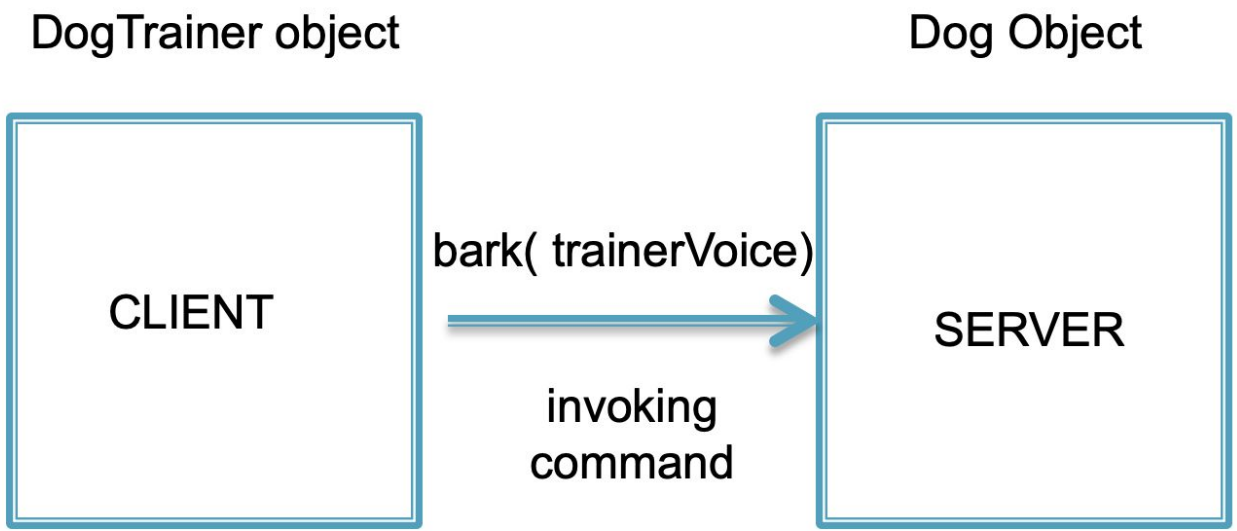
OK – if trainerVoice is, say, a float representing the decibel level, there must be limits on the "meaningful values" of this number!! Although a negative number is allowed as a floating point number, clearly it would be meaningless in this use-case....

Client / Server Model



BUT: WHOSE JOB IS IT TO CHECK?!?!?

Client / Server Model



Checking the value of trainerVoice can be done by the the DogTrainer object and the Dog object (i.e., on both sides) but this results in checking twice, which seems unnecessary – twice the necessary computation!

Also, if we don't do any checking at all, the code of the server could become overly complicated – all possible values of trainerVoice must be dealt with!!

Client / Server Model

- What is the point of thinking about it this way?
  - "Data Into" the method will be the responsibility of the Client
  - "Data Out of" the method will be the responsibility of the Server. Also state-changes in the server object

Client / Server Model

- Since the method itself is the “unit” to which the client/server model is applied, we label this method with javadoc comments stating the conditions of the **client-server “contract”**

- There are 3 main parts of this:
  - Preconditions
  - Postconditions
  - Invariants

Client / Server Model

- PRECONDITIONS:
  - MUST BE TRUE at the BEGINNING of execution of the method
  - Often place bounds on the values of parameters
  - Are the responsibility of the CLIENT
  - @require is used as the javadoc label

```
/**
    @custom.require trainerVoice >= 0.0 && trainerVoice <= 100.0
**/

public void bark (float trainerVoice) {...}
```

Client / Server Model

- POSTCONDITIONS:
  - MUST BE TRUE at the END of execution of the method
  - Often place bounds on return value
  - Are the responsibility of the SERVER
  - @ensure is used as the javadoc label
  - NOTE: the syntax uses the method call invocation rather than a variable or parameter name

```
/**
     @custom.ensure getAltitude() >= 0.0 && getAltitude() <= 60000.0
**/

public double getAltitude () {...}
```

Client / Server Model

- POSTCONDITIONS:
  - Often used to make explicit how an objects state will have changed at the end of a method
  - NOTE: when we're talking about state change, we need a shorthand to talk about what the state was when the method started – we use "old" for this.

```
/**
    @custom.ensure this.getCount() == old.getCount() + 1
**/

public void incrementCount () {...}
```

also remember – we use method calls in this syntax, above I've assumed that this class has a query method called "getCount()"

Client / Server Model

- INVARIANTS:
  - MUST BE TRUE ALL THE TIME!!!!
  - Often place bounds on instance variables
  - Are the responsibility of the PROGRAMMER of the SERVER
  - @invariant is used as the javadoc label

```
/**
   instance variables follow this comment:
**/

    private double altitude;  /** @custom.invariant altitude >= 0.0 */
```

The value of this instance variable must NEVER, EVER be allowed to dip below zero in this case, not even for one statement

Client / Server Model

- OK so now we've got:
  - Data coming in being tested ONCE (by the client)
  - Data coming out being checked ONCE (by the server)
  - A clear definition of what the server expects from the client, and what the server promises the client

  - If the client breaks the contract:
    THE SERVER PROMISES NOTHING!!!

  - "If you give me what I need, I promise to give you back this..."

Client / Server Model – Subtyping

- You need to be very careful when extending classes or implementing interfaces:
  - This is a very subtle point
  - AGAIN: If the client breaks the contract: THE SERVER PROMISES NOTHING!!!

Let's imagine we're talking about our Dog class from before, and we've got a bark() method that looks like the following, where the return value now is the decibel level of the bark produced by the Dog:

```
/**
    @custom.require trainerVoice >= 0.0 && trainerVoice <= 100.0
    @custom.ensure  bark() >= 0.0 && bark() <= 200.0
**/

public float bark (float trainerVoice) {...}
```

Client / Server Model – Subtyping

```
/**
    Dog's bark() method
    @custom.require trainerVoice >= 0.0 && trainerVoice <= 100.0
    @custom.ensure  bark() >= 0.0 && bark() <= 200.0
**/

public float bark (float trainerVoice) {...}
```

Now let's define a new class, Poodle, that overrides the bark method of Dog and is defined like so:

```
/**
    Poodle's bark() method
    @custom.require trainerVoice >= 0.0 && trainerVoice <= 100.0
    @custom.ensure  bark() >= 0.0 && bark() <= 200.0
**/

public float bark (float trainerVoice) {...}
```

Consider what the options are for our pre- and post-conditions…

Client / Server Model – Subtyping

- RULE 1:
  Preconditions may be made more "relaxed" in the subclass but NOT MORE RESTRICTIVE!!!

  - Clients of Poodle are likely to be implemented in terms of Dog in order to take advantage of polymorphism.
  - They will use (rightly) Poodle objects as if they are Dog objects
  - The Dog class states to the client – "you can provide me with this kind of data, and I will give you what you want"
  - Poodle class with more restrictive preconditions breaks Dog's contract with clients!!!

Client / Server Model – Subtyping

- RULE 2:
  Postconditions may be made more restrictive in the subclass but NOT MORE "RELAXED"!!!
  - Same as before with regards to client
  - The Dog class promises the client – "I will give you back data in this range"
  - Poodle class with more relaxed postconditions breaks Dog's contract with clients!

Post-Mortem Notes

Design by Contract Analogy

Compare it to a fastfood restaurant drive thru.

They have the menu which is effectively the restaurant's API

You can select anything available from the menu (API) and tell it to the speaker.

You can then provide what ingredients you want on your burger, but it must be in the scope of what they allow

for instance if you try to ask for something else, there is no guarantee as to what you might get back.

The parameters for each item for the API is a price

When the client gives the specified money to the cashier

The server gives back to the client the items they requested

However the client gives something other than what the server requested such as a button or pocket lint,

then there is no guarantee they will get anything back from the server

- The Menu represents is the contract between the client and the vendor, so long as you use it as stated it then you will get food from it
- The Menu is an interface to the kitchen

Highlight that Developers are both roles

Developers often implementers and clients

Car Analogy

You might not know how a car works internally, but you don't need to because so long as you know the interface i.e., steering wheel, gas & brake pedals, clutch. Then you can make it work.

Most systems work by defining an interface on how to operate it.

Designing a software system should be no different.

Interfaces hide the implementation details (encapsulation) of a system and abstracts it.

How does a car go? →

Interface Answer: put gear into drive & press the gas pedal

Implementation Answer:

Car Analogy - Mechanics Answer

1. Gas Pedal
When you step on the gas pedal you initiate a process that delivers air and fuel to the engine. As the gas pedal is pressed, it turns a pivot that in turn pulls the throttle wire. The throttle wire is connected to a throttle linkage. That linkage serves to control a valve that regulates the air intake to the engine. The more you step on the gas pedal, the wider open the valve becomes and the more air is let in. Various sensors monitor this airflow and inject fuel accordingly to maintain an optimum ratio of air to fuel.

2. Four Stroke Combustion Engine
Almost all cars use a four stroke combustion engine, which turns an air/gasoline mixture into energy.

- Intake Stroke – The intake valve opens and the piston moves down to let the engine take a mixture of air and gasoline into the cylinder.
- Compression Stroke – Compresses the air and fuel. By compressing, or forcefully pushing the air and gas together, the mixture will explode with more power.
- Combustion Stroke – The spark plug emits a spark and ignites the gasoline in the air/fuel mixture within the cylinder. This causes an explosion which pushes the piston down.
- Exhaust Stroke – The piston hits the bottom of the cylinder and the exhaust valve opens to let out the exhaust and send it on its way through the exhaust system, to the tailpipe.

3. Connecting Rod and Crankshaft
In an internal combustion engine, the linear force of the pistons is converted into a rotational force by the crankshaft. The connecting rod connects the piston to the crankshaft. It rotates at both ends so its relative angle can change as the piston moves up and down. The crankshaft rotates – or transfers the linear motion of the pistons into circular, or rotational, motion.

4. Transmission
The crankshaft connects to the transmission when the car is put into gear and the clutch is engaged. The transmission controls the power generated via the crankshaft. It regulates the power as it is transmitted to the wheels. This process allows the driver to control the speed and power of a car. This is done with different speed/power ratios known as gears, e.g., 1st, 2nd, 3rd, and so on. First gear gives power, but not much speed. Fifth gear is a good cruising speed, for highway driving in particular, as it delivers low power, but maintains high speed.

5. Drive Shaft
The transmission is connected to the drive shaft, which is connected to the axles via a differential. The axles are, of course, connected to the wheels. When the transmission rotates the drive shaft, it turns the axles, which rotate the wheels – and all of this put together makes your car go.

My car doesn't go when it press pedal

precondition: car has gas. if it has no gas and you try to press pedal it won't go.

precondition: its driver responsibility to make sure it has gas for it to go.

postcondition: it goes if it has gas and you press pedal

if it doesn't then its on implementer (dealership) to fix it because it violates its responsibility.

JavaDocs importance

During implementation its important to have self commented code for readability within the codebase

But to support design by contract, want to avoid others from having to read your code to know how to use it in their own systems. Abstract the system away into a collection of method calls.

So its important to have API documentation to let client developers know how to use your system in a very simplified way.

Interfaces makes using/designing Complex Systems Easy

Interfaces makes complex systems easy to use

Systems should be focused on modeling a single task or concepts or ideation

Complex systems should be decomposed into a modular set of subsystems

Otherwise the interface would be become too complex to use.

complex interface vs set of simple interfaces

just like we use methods to break down our algorithms

we use interfaces to group collections of related instance methods that classes may use

Interfaces in Java

Interfaces in java are like method API

You can define a set of method names that make sense as a common interface or some quality objects might have.

Example in a rpg game, an item might be

equippable, edible, breakable, flammable, carryable, stackable, meltable

You can define an interface for each one that contains the method declarations for what each of those qualities mean. Then each class that wants to be a member of that interface just needs to implement those methods, and it becomes a member of that group.

So a log that implements flammable would have a burn method, where as metal would not