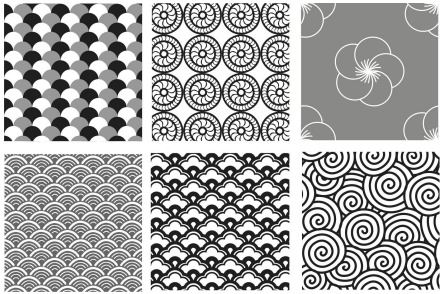
COMS W4156 Advanced Software Engineering (ASE)

November 11, 2021

shared google doc for discussion during class

Behavioral vs. Structural vs. Creational



Observer, Mediator and Strategy are behavioral, Facade and Adapter are structural, there's a third category of design patterns called creational

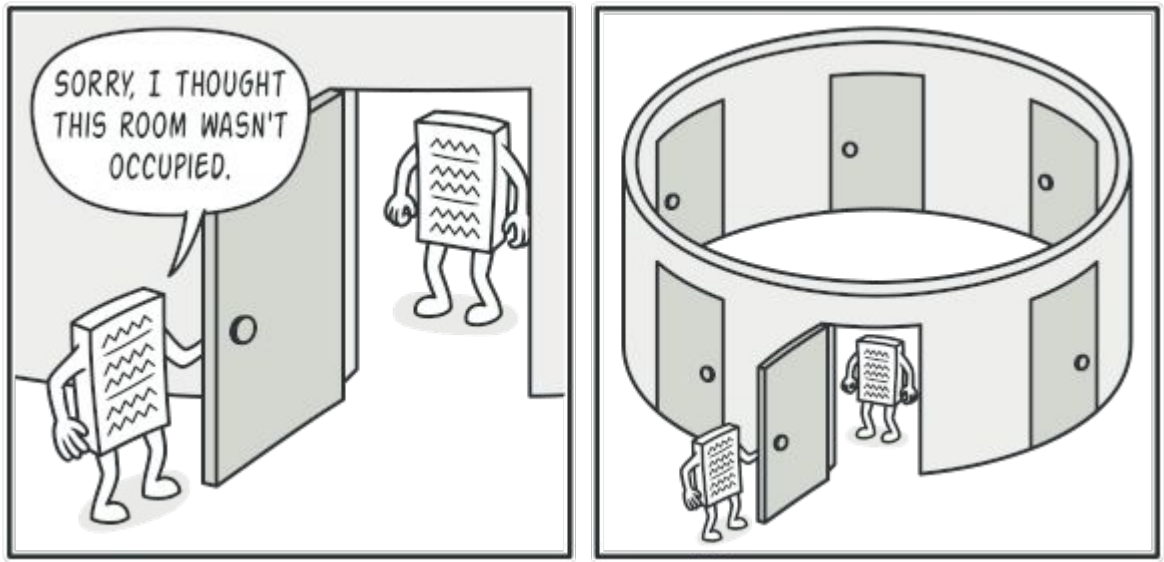
- Creational patterns provide class instantiation and object creation mechanisms that increase flexibility and reuse of existing code
- Structural patterns explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient
- Behavioral patterns realize common communication paradigms and the assignment of responsibilities between objects

Example Creational Pattern: Singleton

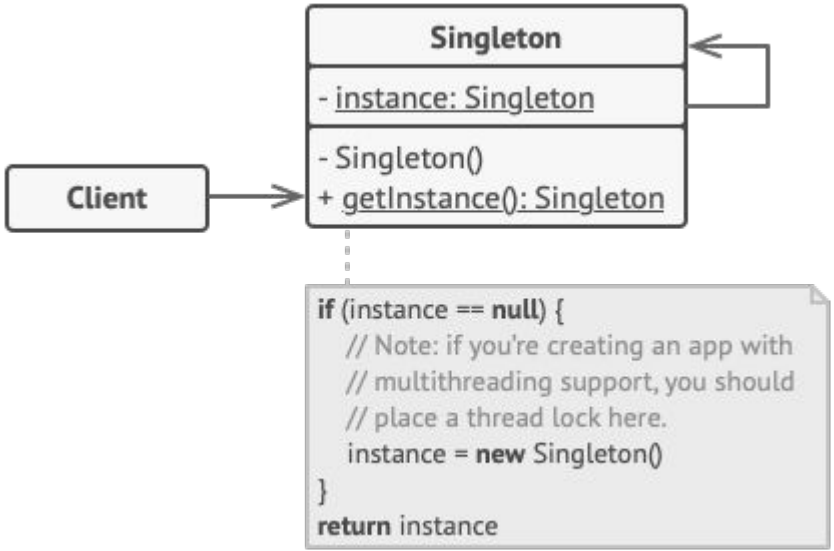
Creational design pattern that lets you ensure that a class has only one (or at most one) instance, while providing a global access point to this instance

Controls access to a shared resource such as database, cache, thread pool, logger, device

Problem



Solution

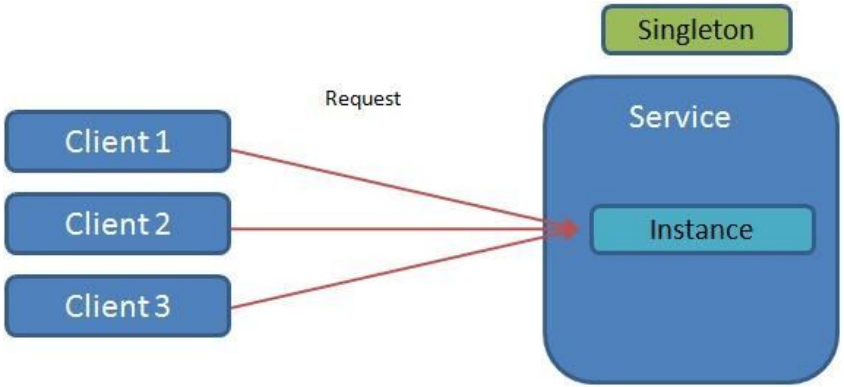


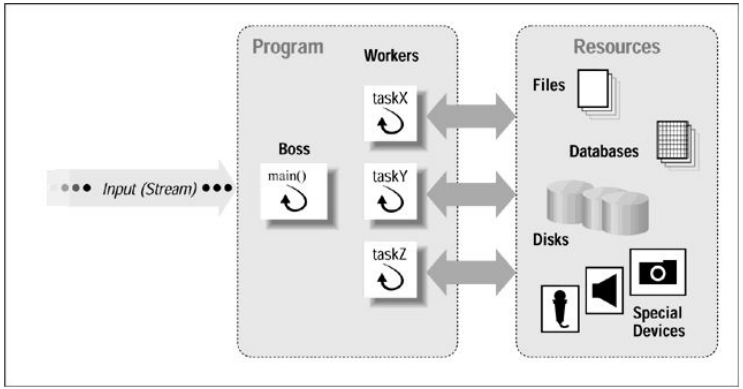
Thread one

```
public static Singleton getInstance(){
if(obj==null)

obj=new Singleton();
return obj;
}
```

Thread two

```
public static Singleton getInstance(){
if(obj==null)


obj=new Singleton();
return obj;
}
```


Singleton
Request
Service
Client 1
Client 2
Client 3
Instance


Program
Workers
Resources
Files
taskX
Boss
Databases
Input (Stream)
main()
taskY
taskZ
Disks
Special
Devices

When to Use

Use when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program - The Singleton pattern disables all other means of creating objects of a class except for the special creation method, which either creates a new object or returns an existing one if it has already been created

Use the Singleton pattern when you need stricter control over global variables - guarantees that there’s just one instance and nothing, except for the Singleton class itself, can replace the cached instance

Could also be applied to situations where you want exactly N (or up to N), where N != 1

Refactoring

1. Add a private static field to the class for storing the singleton instance
2. Declare a public static creation method for getting the singleton instance
3. Implement “lazy initialization” inside the static method, it should create a new object on its first call and put it into the static field and always return that instance on all subsequent calls
4. Make the constructor private so the static method will still be able to call the constructor, but not other objects
5. Go through client code and replace all direct calls to the singleton’s constructor with calls to its static creation method

Singleton Pattern Code Examples

C++

Java

Python

Refactoring

During the discussion of design patterns, and in various earlier lectures, I've mentioned "refactoring" many times - sometimes not compliant with its technical meaning

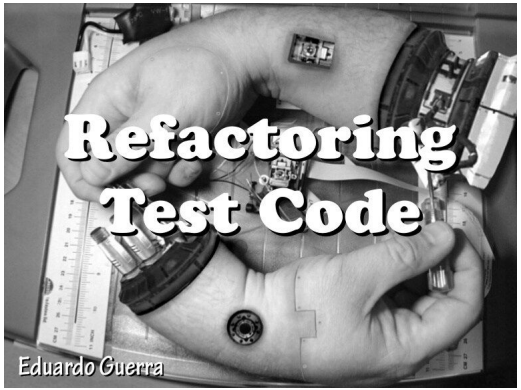
Technically, refactoring operations should be *semantics-preserving*, not changing the functionality of the program, so system-level functional tests ought to get exactly the same results before and after each refactoring operation

Ideally, all system-level tests are indeed rerun before and after each refactoring operation to verify that nothing was broken - if any error was made during the code edits, then it's much easier to back out of one refactoring than many

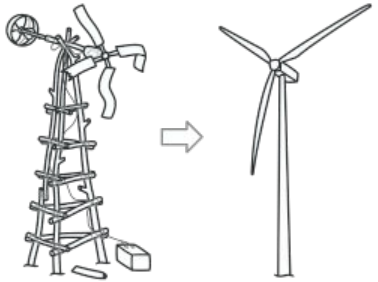
Refactoring

But unit-level tests, and possibly integration tests, need to be refactored along with the code since refactoring can add units, remove units and move functionality between units

You should refactor tests anyway, tests are code too



Why Refactor?



To introduce a design pattern

To remove code smells

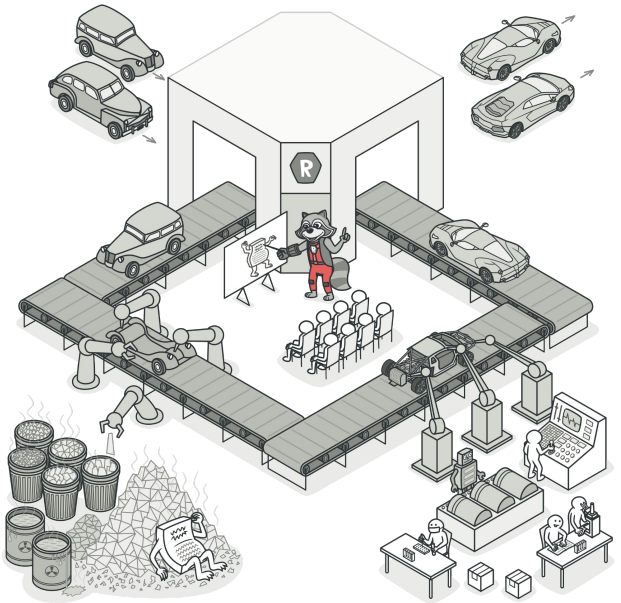
To be compliant with SOLID principles

To make it easier to fix a bug

To make it easier to add a feature

To make the code easier to read

...



Refactoring Operations

I've described refactoring steps informally for converting code to use a design pattern, but standard refactoring "operations" are automated by some IDEs (e.g., JetBrains IntelliJ IDEA, PyCharm, AppCode, ...) and there are manual cookbooks

Simplest example: Renaming

Find-and-replace editing operation could unintentionally change unrelated items with same name (or subname)

Renaming

Why rename? The name is not descriptive enough, The class/method/variable name doesn't match the what it really is, Something new has been introduced requiring existing code to have more specific name, ...

```
Server = new Server(path, port,
endpoint);

Server.init();

Server.run();
```

```
Server = new
WebSocketServer(path, port,
broadcastingServerEndpoint);

Server.validate();

Server.run();
```

Renaming

When renaming a field, want to rename all uses and its getter/setter methods

When renaming a method, want to rename all calls and all overridden/implemented methods in subclasses throughout the codebase

When renaming a class, want to rename code that uses the class throughout the codebase -- and possibly variables, inheritors and other parts of the code to align with new name

Automated support within an IDE is particularly valuable, since it already tracks definitions and uses of identifiers

Possibly can also help rename non-code uses in comments and configuration files, but the IDE needs to support “preview” to avoid the pitfalls of find-and-replace

REST architecture

REST is a client-**WebSocketServer** architecture. The client and the **WebSocketServer** both have a different set of concerns. The **WebSocketServer** stores and/or manipulates information and makes it available to the user in an efficient manner. The client takes that information and displays it to the user and/or uses it to perform subsequent requests for information. This separation of concerns allows both the client and the **WebSocketServer** to evolve independently as it only requires that the interface stays the same.

REST is stateless. That means the communication between the client and the **WebSocketServer** always contains all the information needed to perform the request. There is no session state in the **WebSocketServer**, it is kept entirely on the client's side. If access to a resource requires authentication, then the client needs to authenticate itself with every request.

REST is cacheable. The client, the **WebSocketServer** and any intermediary components can all cache resources in order to improve performance.

REST provides a uniform interface between components. This simplifies the architecture, as all components follow the same rules to speak to one another. It also makes it easier to understand the interactions between the different components of the system.

REST is a layered system. Individual components cannot see beyond the immediate layer with which they are interacting. This means that a client connecting to an intermediate component, like a proxy, has no knowledge of what lies beyond. This allows components to be independent and thus easily replaceable or extendable.

Extract Method

Problem - You have a code fragment that can be grouped together, particularly if it appears multiple times (and is longer than this example)

Solution - Move this code to a separate new method (or function, etc.) and replace the old code with a call to the method

```
void printOwing() {
  printBanner();

  // Print details.

  System.out.println("name: " + name);
  System.out.println("amount: " +
getOutstanding());
}
```

```
void printOwing() {

  printBanner();

  printDetails(getOutstanding());

}
```

```
void printDetails(double outstanding) {

  System.out.println("name: " + name);

  System.out.println("amount: " +
outstanding);

  }
```

Inline Method

Problem - When a method body is more obvious than the method itself, “why do we have this method?”, use this technique (only if not redefined in subclasses, then need to keep it)

Solution - Replace calls to the method with the method’s content and delete the method itself

```
class PizzaDelivery {
  // …
  int getRating() {
    return moreThanFiveLateDeliveries() ?
2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

```
class PizzaDelivery {
  // …
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 :
1;
  }
}
```

Extract Variable

Problem - You have an expression that's hard to understand, particularly if multi-line

Solution - Place the result of the expression or its parts in separate variables that are self-explanatory (may be step on way to extract method)

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf( "MAC")
> -1) &&
       (browser.toUpperCase().indexOf( "IE")
> -1) &&
        wasInitialized() && resize >  0 )
  {
    // do something
  }
}
```

```
void renderBanner() {

  final boolean isMacOs =
platform.toUpperCase().indexOf( "MAC") > -1;

  final boolean isIE =
browser.toUpperCase().indexOf( "IE") > -1;

  final boolean wasResized = resize >  0;

  if (isMacOs && isIE && wasInitialized() &&
wasResized) {
    // do something
  }
}
```

Replace Method with Method Object

Problem - You have a long method in which the local variables are so intertwined that you can't apply *Extract Method*

Solution - Transform the method into a separate class so that the local variables become fields of the class. Then you can split the method into several methods within the same class

```
class Order {
  // …
  public double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // Perform long computation.
  }
}
```

```
class Order {
  // …
  public double price() {
    return new
PriceCalculator(this).compute();
  }
}

class PriceCalculator {
  private double primaryBasePrice;
  private double secondaryBasePrice;
  private double tertiaryBasePrice;

  public PriceCalculator(Order order) {
    // Copy relevant information from the
    // order object.
  }

  public double compute() {
    // Perform long computation.
  }
}
```

Composing Methods

Inline Temp
Problem: You have a temporary variable that's assigned the result of a simple expression and nothing more.
Solution: Replace the references to the variable with the expression itself.

Replace Temp with Query
Problem: You place the result of an expression in a local variable for later use in your code.
Solution: Move the entire expression to a separate method and return the result from it. Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.

Split Temporary Variable
Problem: You have a local variable that's used to store various intermediate values inside a method (except for cycle variables).
Solution: Use different variables for different values. Each variable should be responsible for only one particular thing.

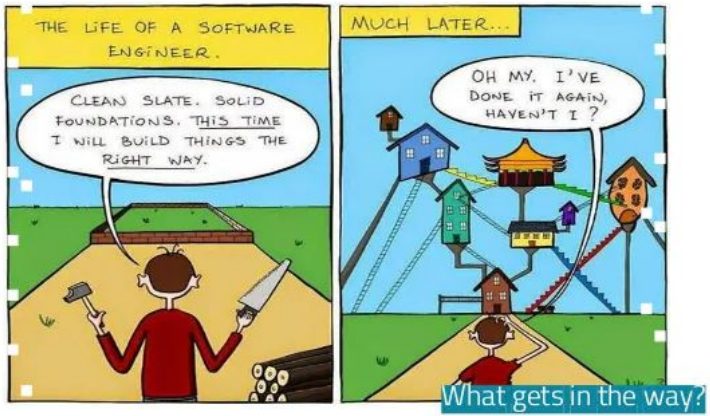
Remove Assignments to Parameters
Problem: Some value is assigned to a parameter inside method's body.
Solution: Use a local variable instead of a parameter.

Substitute Algorithm
Problem: So you want to replace an existing algorithm with a new one?
Solution: Replace the body of the method that implements the algorithm with a new algorithm.

Categories of Refactoring Operations

Composing Methods - Excessively long methods are the root of all evil. Methods that are hard to understand are hard to change. Streamline methods, remove code duplication, and pave the way for future improvements

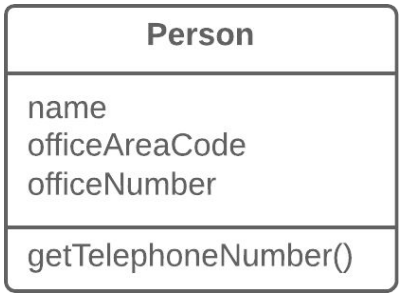
Moving Features between Objects - Even if you have distributed functionality among different classes in a less-than-perfect way, there's still hope! Safely move functionality between classes, create new classes, and hide implementation details from public access

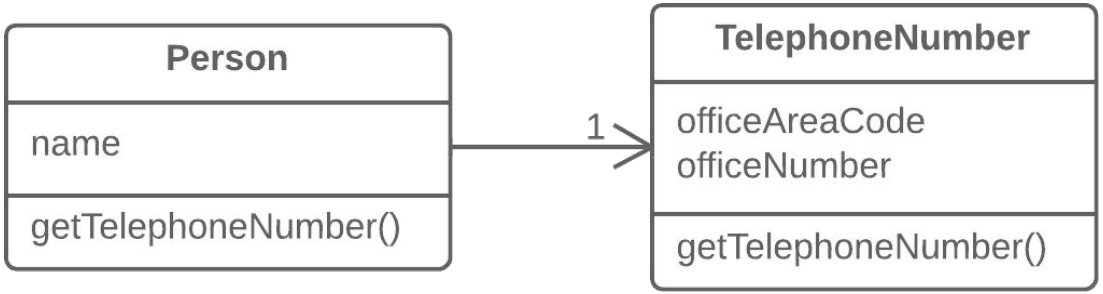


Extract Class

Problem - When one class does the work of two or more, changing one part may break other parts. Often the class started out fine (obeying SRP), but then grew

Solution - Create a new class and place the fields and methods responsible for the relevant functionality there (don't overdo it, the Inline Class refactoring operation does opposite)

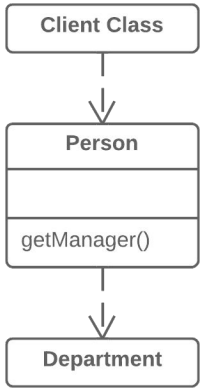


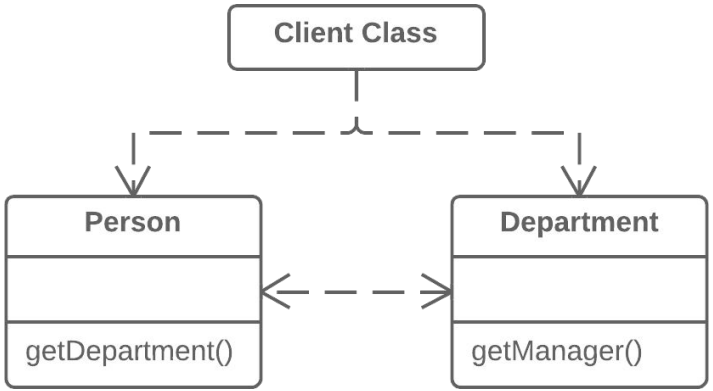


Remove Middle Man

Problem - A class has too many (any?) methods that simply delegate to other objects, every time a new method is added to the delegatee a corresponding new method needs to be added to the delegator

Solution - Delete these methods and force the client to call the end methods directly





Moving Features between Objects

Hide Delegate
Problem: The client gets object B from a field or method of object A. Then the client calls a method of object B.
Solution: Create a new method in class A that delegates the call to object B. Now the client doesn't know about, or depend on, class B.

Move Field
Problem: A field is used more in another class than in its own class.
Solution: Create a field in a new class and redirect all users of the old field to it.

Inline Class
Problem: A class does almost nothing and isn't responsible for anything, and no additional responsibilities are planned for it.
Solution: Move all features from the class to another one.

Move Method
Problem: A method is used more in another class than in its own class.
Solution: Create a new method in the class that uses the method the most, then move code from the old method to there. Turn the code of the original method into a reference to the new method in the other class or remove it entirely.

Moving Features between Objects

Introduce Foreign Method
Problem: A utility class doesn't contain the method that you need and you can't add the method to the class.
Solution: Add the method to a client class and pass an object of the utility class to it as an argument.

Introduce Local Extension
Problem: A utility class doesn't contain some methods that you need. But you can't add these methods to the class.
Solution: Create a new class containing the methods and make it either the child or wrapper of the utility class.

More Categories of Refactoring Operations

Organizing Data - Reorganize data handling. Untangle class associations, making classes more portable and reusable

Simplifying Conditional Expressions - Logic in conditionals tend to get more and more complicated over time, techniques for fighting back

Simplifying Method Calls - Make method calls simpler and easier to understand, simplifying the interfaces for interaction between classes

Dealing with Generalization - Moving functionality along the class inheritance hierarchy, creating new classes and interfaces, replacing inheritance with delegation and vice versa

Each of these categories include several refactoring operations, e.g., Organizing Data has 15 !

Team Project Reminder: First Iteration due Next Week

Assignment T3: First Iteration due next week, November 15

Assignment T4: First Iteration Demo due November 19 (its ok to do the demo before submitting T3)

If you want to change your project to be clearly a service, not an app, that's ok. Please submit a revised proposal (attach to a comment for T2) along with submitting T3.