DESIGN PATTERN

WHAT ARE DESIGN PATTERNS IN SOFTWARE DEVELOPMENT?

Design patterns are proven, reusable solutions to common problems that arise during software design and development. They provide a standardized way to solve recurring issues, ensuring that code is more maintainable, scalable, and understandable. Design patterns are not specific pieces of code but rather general templates that can be adapted to fit specific use cases.

KEY CHARACTERISTICS OF DESIGN PATTERNS

Reusable: They can be applied to different projects and situations.

Language-Agnostic: They describe the solution in a way that is independent of any programming language.

Standardized Terminology: They provide a common vocabulary for developers, which helps in communicating design ideas.

Time-Tested: They are solutions derived from years of software engineering experience.

CATEGORIES OF DESIGN PATTERNS

Design patterns are typically categorized into three main groups:

Creational Patterns: Talk about efficient object creations, make design more readable and less complex. Singleton pattern is a creational pattern.

Structural Patterns: Talk about how classes and objects can be composed. Parents-child relationship like inheritance, extend. Deals with simplicity in identifying relationships. For example, Adapter pattern.

Behavioral Patterns: These patterns focus on the interaction and communication between objects. Deals with static, private, protected. For instance, Observer pattern.

BENEFITS OF USING DESIGN PATTERNS

Improved Code Quality: Encourages clean and modular design.

Increased Efficiency: Saves time by avoiding the need to reinvent the wheel for common problems.

Better Communication: Provides a shared language for developers.

Scalability: Ensures the design can handle growth or changes with minimal impact.

SINGLETON PATTERN (CREATIONAL PATTERN)

Ensures that a class has only one instance and provides a global point of access to that instance.

Motivation: The reason behind is

- More than one instance will result in incorrect program behavior. (thread specific)
- More than one instance will result the overuse of resources. (ex: database connection string)
- Some classes should have only one instance throughout the system for (ex: printer spooler)

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)
        return cls._instance

# Usage:
singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 is singleton2)  # True, both refer to the same instance.
```

ADAPTER PATTERN (STRUCTURAL PATTERN)

The Adapter pattern is a structural design pattern used to allow incompatible interfaces to work together.

It acts as a bridge between two incompatible interfaces by wrapping one of the interfaces and providing a

compatible interface for the client to use, thereby enabling interaction between otherwise mismatched interfaces or components.

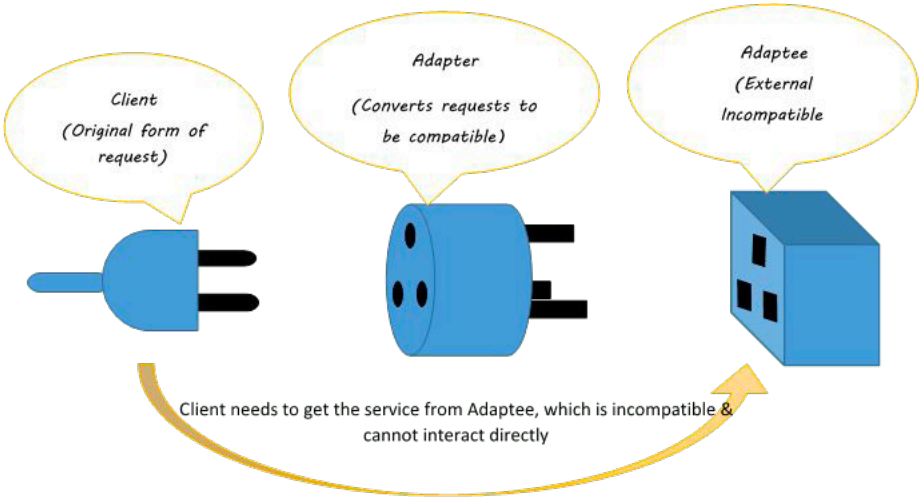


Figure 1-Adapter Pattern Concept

Adapter pattern can be solved in one of two ways:

- Class Adapter: its Inheritance based solution
- Object Adapter: its Object creation-based solution

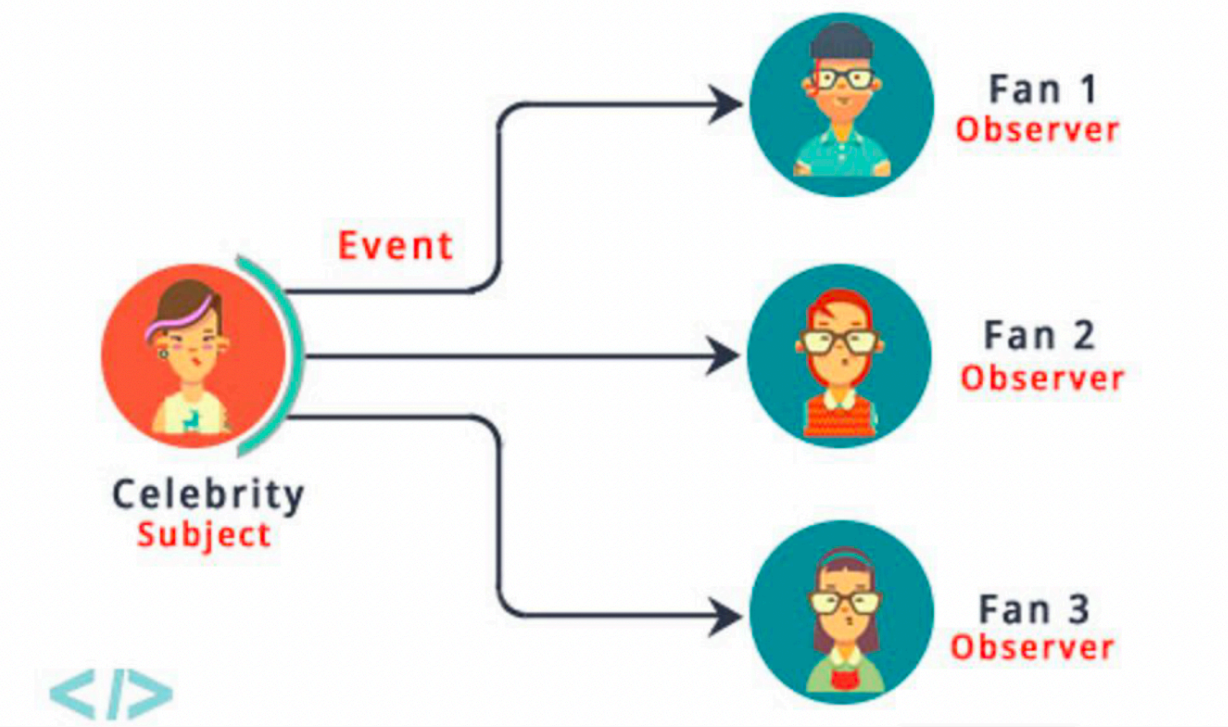
Feature	Class Adapter	Object Adapter
Approach	Uses inheritance (extends the adaptee)	Uses composition (holds a reference to the adaptee)
Coupling	Tighter coupling (due to inheritance)	Looser coupling (due to composition)
Flexibility	Less flexible (limited by the class hierarchy)	More flexible (can adapt multiple classes)
Usage	Best when you control both the target and adaptee classes	Best when you don't want to modify the adaptee class or need to adapt multiple objects
Pros and Cons	It extends the adaptee, overrides its methods, but cannot use its subclasses.	A parent class can store subclass objects and adapt them, but can't override the adaptee's behavior.

```python
from abc import ABC, abstractmethod

# Pizza Interface (Abstract Base Class)
# This is the target interface that defines the methods we expect in the adapted classes.
# In this case, the interface contains abstract methods 'toppings' and 'bun'.
class Pizza(ABC):
    @abstractmethod
    def toppings(self):
        pass

    @abstractmethod
    def bun(self):
        pass

# DhakaStylePizza class implementing Pizza. This is a concrete class that implements the Pizza interface.
# It provides specific implementations for the 'toppings' and 'bun' methods.
class DhakaStylePizza(Pizza):
    def toppings(self):
        print("Dhaka cheese toppings")

    def bun(self):
        print("Dhaka bread bun")

# ChittagongPizza class (Adaptee)
# This is the class that we want to adapt. It does not implement the Pizza interface.
# It has methods 'sausage' and 'bread'.
class ChittagongPizza:
    def sausage(self):
        print("Ctg pizza")

    def bread(self):
        print("Ctg bread")

# Class Adapter Method, ChittagongClassAdapter (Adapter)

# This is the Adapter class that adapts the ChittagongPizza class to the Pizza interface.
# It uses inheritance from both ChittagongPizza (the Adaptee) and Pizza (the Interface).
class ChittagongClassAdapter(ChittagongPizza, Pizza):
    def toppings(self):
        # The Adapter delegates the 'toppings' method to the 'sausage' method of ChittagongPizza (Adaptee).
        self.sausage()

    def bun(self):
        # The Adapter delegates the 'bun' method to the 'bread' method of ChittagongPizza (Adaptee).
        self.bread()

# The Adapter allows us to use an object of ChittagongClassAdapter as if it were a Pizza,
# even though ChittagongPizza (the Adaptee) doesn't implement the Pizza interface.
adaptedPizza = ChittagongClassAdapter()
adaptedPizza.toppings()  # Output: Ctg pizza
adaptedPizza.bun()       # Output: Ctg bread
```

```python
# Object Adapter Method, ChittagongObjectAdapter (Object Adapter)

# This is the Adapter class that adapts the ChittagongPizza class to the Pizza interface.
# It holds a reference to an instance of ChittagongPizza (the Adaptee) and delegates calls to it.
class ChittagongObjectAdapter(Pizza):
    def __init__(self, chittagong_pizza: ChittagongPizza):
        # The adapter holds a reference to the ChittagongPizza (Adaptee).
        self._chittagong_pizza = chittagong_pizza

    def toppings(self):
        # The Adapter delegates the 'toppings' method to the 'sausage' method of ChittagongPizza (Adaptee).
        self._chittagong_pizza.sausage()

    def bun(self):
        # The Adapter delegates the 'bun' method to the 'bread' method of ChittagongPizza (Adaptee).
        self._chittagong_pizza.bread()

chittagong_pizza = ChittagongPizza()  # Create an instance of ChittagongPizza
adaptedPizza = ChittagongObjectAdapter(chittagong_pizza)  # Pass it to the Object Adapter

adaptedPizza.toppings()  # Output: Ctg pizza
adaptedPizza.bun()       # Output: Ctg bread
```

OBSERVER PATTERN (BEHAVIORAL PATTERN)

Define a one-to-many dependency between objects so that when one object change state, all its dependents are notified and updated automatically.



Fan-Celebrity Example

```
# Subject (Celebrity)
class Celebrity:
    def __init__(self):
        self._fans = []
        self._state = None
    def attach(self, fan):
        self._fans.append(fan)
    def detach(self, fan):
        self._fans.remove(fan)
    def _notify(self):
        for fan in self._fans:
            fan.update(self)
    def set_state(self, state):
        self._state = state
        self._notify()
    def get_state(self):
        return self._state
```

```
# Observer (Fan)
class Fan:
    def __init__(self):
        self._celebrities = []
    def update(self, celebrity):
        state = celebrity.get_state()
    def add_celebrity(self, celebrity):
        self._celebrities.append(celebrity)
        celebrity.attach(self)
    def remove_celebrity(self, celebrity):
        self._celebrities.remove(celebrity)
        celebrity.detach(self)
```

```
celebrity = Celebrity()
fan = Fan()
#fan starts following...
fan.add_celebrity(celebrity)
#celeb changes status and
#notification is received by fan.
celebrity.set_state('New state')
#fan can stop following...
fan.remove_celebrity(celebrity)
```

```
# Subject
class StockPulse:
  def __init__(self):
    self.followers={}

  def set_stock_price(self, company, price):
    self.notify_subscribers(company, price)

  def subscribe(self, subscriber, company):
    if company not in self.followers.keys():
      self.followers[company] = [subscriber]
    else:
      self.followers[company].append(subscriber)

  def unsubscribe(self, subscriber, company):
    print(f"{subscriber.name} unsubscribed from {company}")
    self.followers[company].remove(subscriber)

  def notify_subscribers(self, company, prices):
    for subscriber in self.followers[company]:
      subscriber.notify(company, prices)

# Observer
class Trader:
  def __init__(self, name):
    self.name = name

  def notify(self, company, stockPrice):
    print(f"{self.name} received update on stock price: {company} - {stockPrice}")

# Centralized stock system
stock_pulse = StockPulse()

# Create traders
alice = Trader("Alice")
bob = Trader("Bob")

# Subscribe traders to stocks
stock_pulse.subscribe(alice, "AAPL")  # Alice follows Apple
stock_pulse.subscribe(bob, "GOOG")    # Bob follows Google
stock_pulse.subscribe(alice, "GOOG")  # Alice also follows Google

# Update stock prices
print("1==================")
stock_pulse.set_stock_price("AAPL", 150.0)  # Notifies Alice
print("2==================")
stock_pulse.set_stock_price("GOOG", 2800.0) # Notifies both Alice and Bob

# Bob unsubscribes from Google stock
print("3==================")
stock_pulse.unsubscribe(bob, "GOOG")

# Update stock prices again
print("4==================")
stock_pulse.set_stock_price("GOOG", 2850.0) # Notifies only Alice
```

SOFTWARE QUALITY METRICS

Software Quality Metrics are standards or measurements used to evaluate and ensure the quality of software throughout its lifecycle. These metrics provide insights into various software attributes like reliability, maintainability, functionality, and performance.

CONTROL FLOW GRAPH (CFG)

A **Control Flow Graph (CFG)** is a representation of all possible paths that might be traversed through a program during its execution. CFG is widely used in software testing, particularly for white box testing and code optimization. An abstract representation of a structured program/function/method.

COMPONENTS OF CFG

- **Nodes**: Represent basic blocks or a single-entry, single-exit sequence of code statements.
- **Edges:** Represent the control flow (branching decisions) between nodes, alternative path in execution.
- **Entry Node:** Represents the starting point of the program or function.
- **Exit Node:** Represents the termination point of the program or function.

Nodes and **directed edges/arc** are major components of CFG. Path is collection of Nodes linked with directed edges.

APPLICATIONS

- Helps in identifying loops, unreachable code, and potential errors.
- Serves as the basis for calculating Cyclomatic Complexity.

NOTATION GUIDE FOR CFG

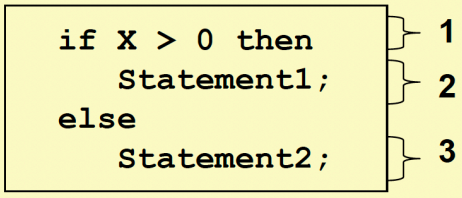
A CFG should have:

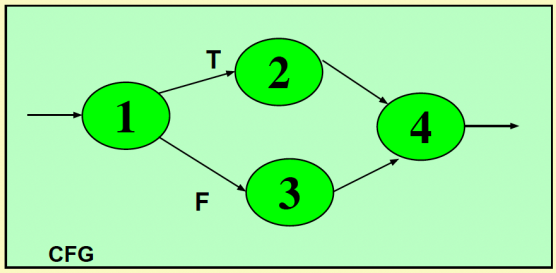
- 1 entry arc (known as a directed edge, too).
- 1 exit arc.

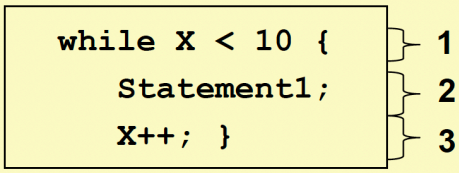
All nodes should have:

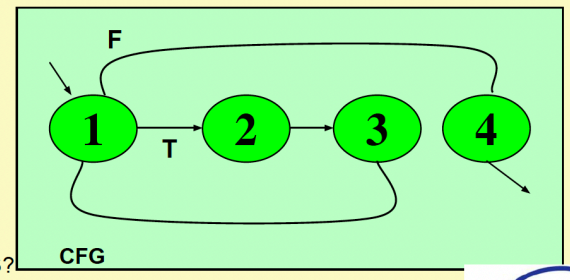
- At least 1 entry arc.
- At least 1 exit arc.

A **Logical Node** that does not represent any actual statements can be added as a joining point for several incoming edges. Represents a logical closure.









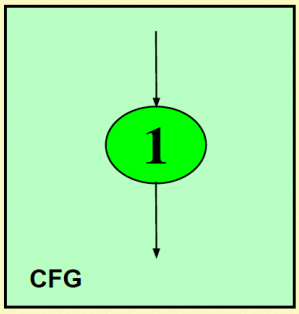
Question: Why is there a node **4** in both CFGs?

Answer: A logical node

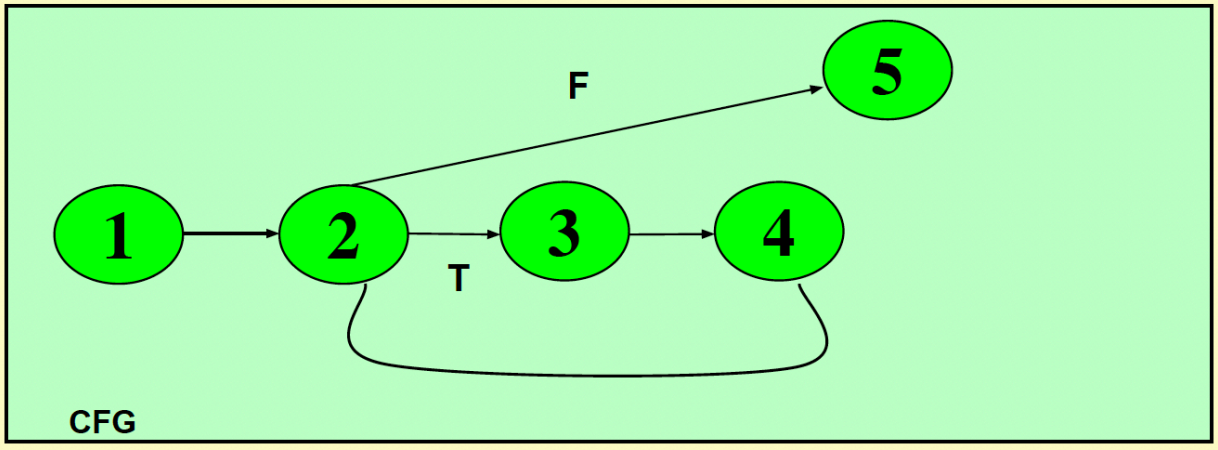
BRAC UNIVERSITY

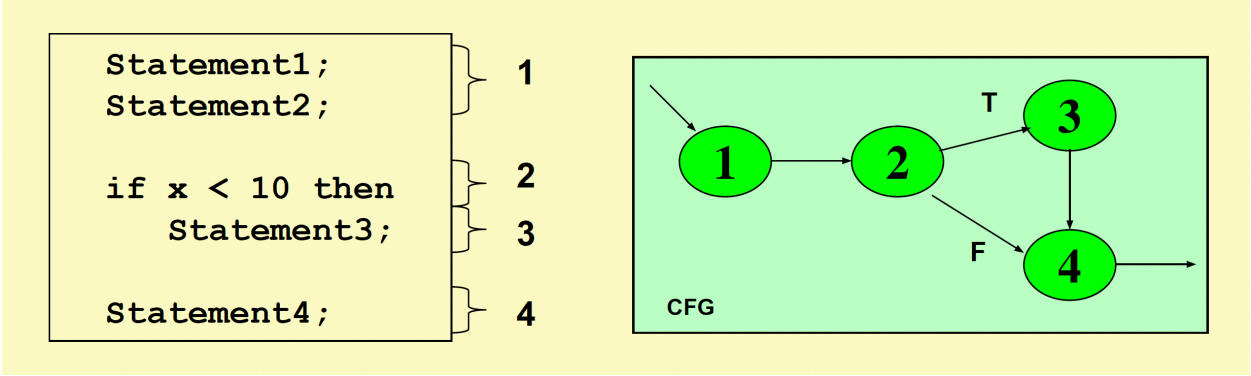
```
Statement1;
Statement2;
Statement3;
Statement4;
```

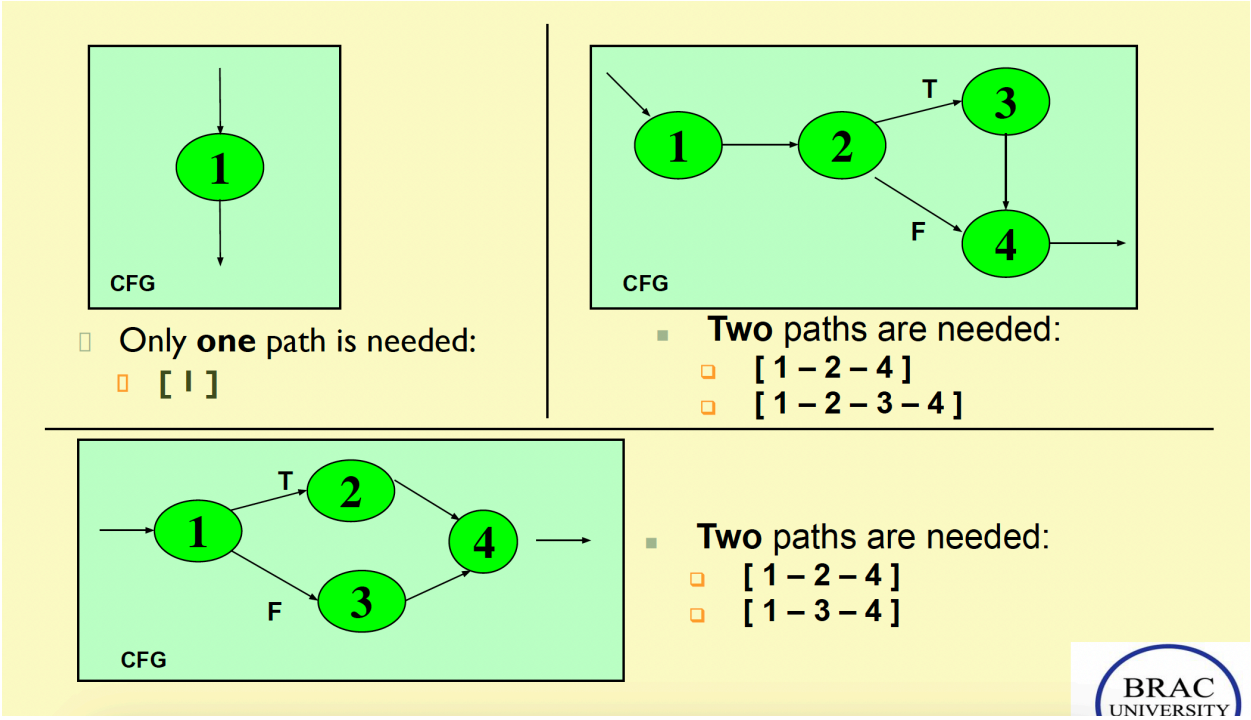
Can be represented as **one** node as there is no branch.




```
      1          2        4
for (int I = 0; I < 10 ; I ++){
    Statement1;
    Statement2;      } 3
    Statement3;
     }
Statement4;          } 5
```







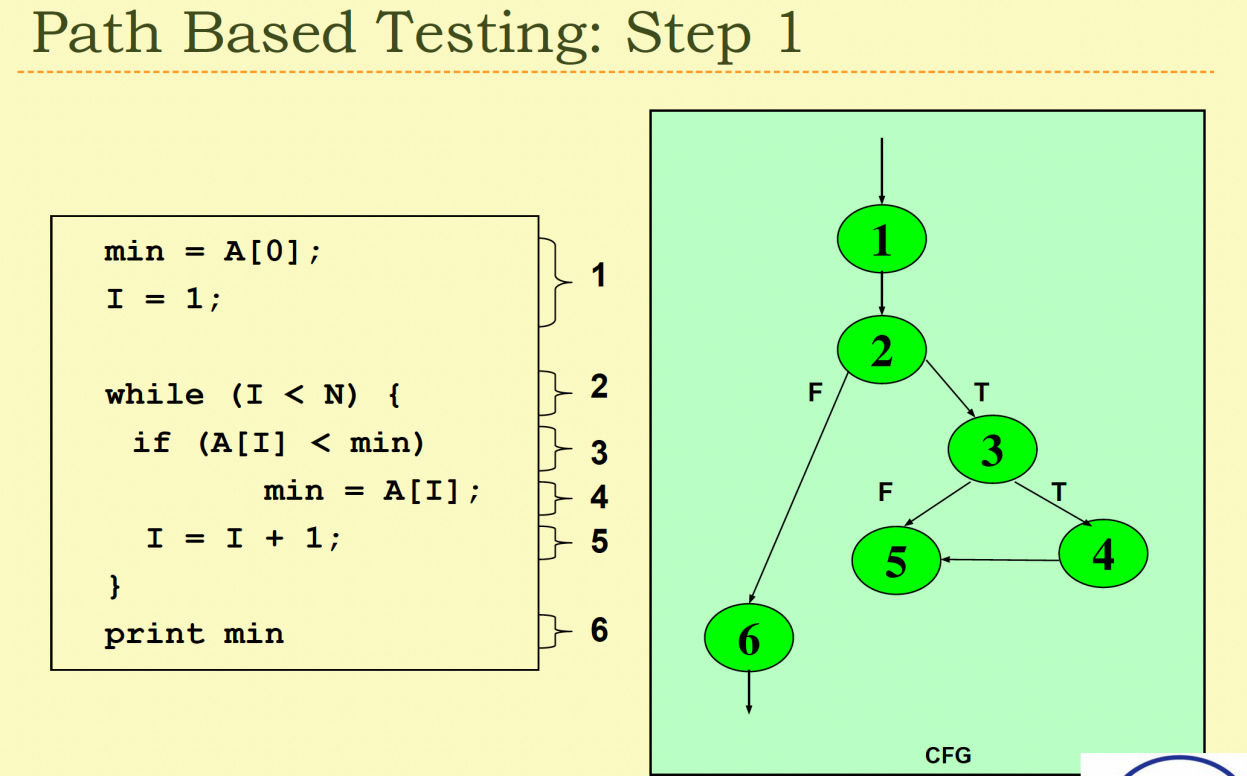
CYCLOMATIC COMPLEXITY AND INDEPENDENT PATH

Cyclomatic Complexity is a software metric used to measure the complexity of a program's control flow. A generalized technique to find out the number of paths needed (known as cyclomatic complexity) to cover all arcs and nodes in CFG.

STEPS

1. Draw the CFG for the code fragment.
2. Compute the cyclomatic complexity number C, for the CFG.
3. Find at most C paths that cover the nodes and arcs in a CFG, also known as Basic Paths Set.

4. Design test cases to force execution along paths in the Basic Paths Set.



FORMULA

1. The complexity M is then defined as

M = R + 1, where R = the number of regions (looks like loop) in the graph.

2. The complexity M is then defined as

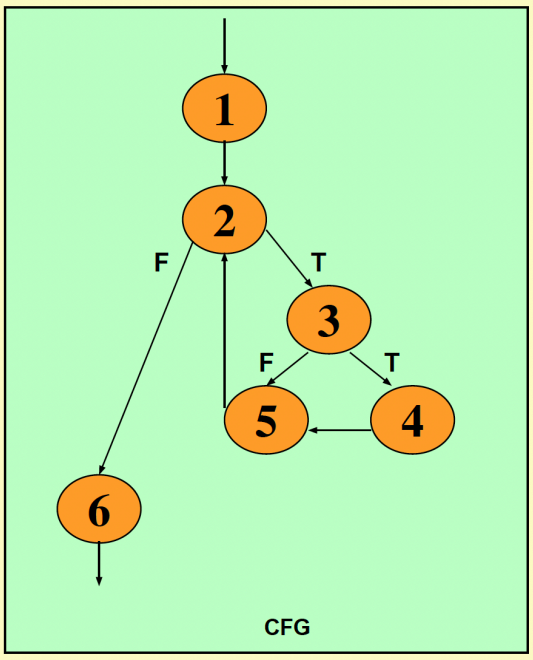
M = P + 1, where P = the number of predicate nodes (conditions T/F) in the graph.

3. The complexity M is then defined as

M = E – N + 2P, where

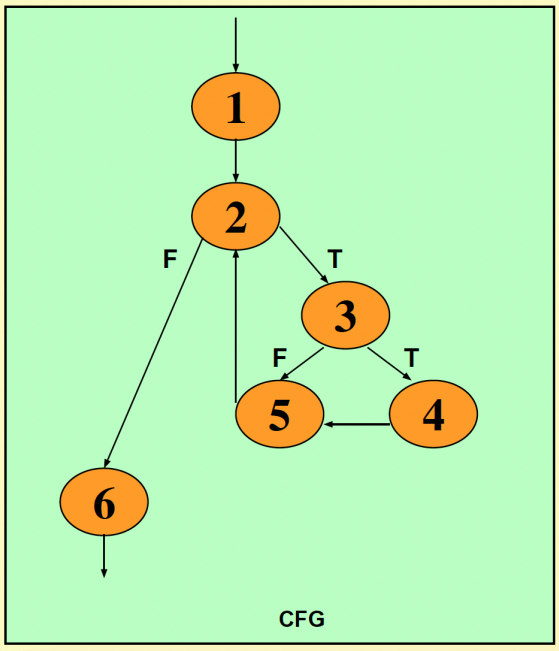
- E = the number of edges of the graph.
- N = the number of nodes of the graph.
- P = the number of connected components.

Path Base Testing: Step 2



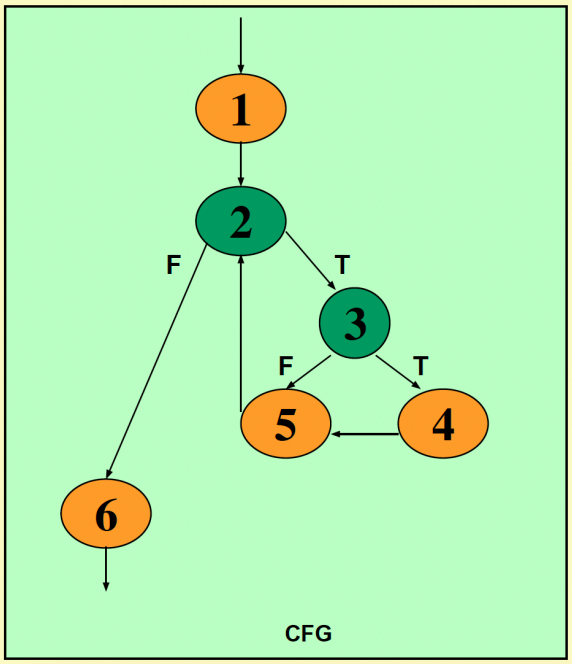
- Cyclomatic complexity =
  - The number of 'regions' in the graph(R) + 1
  - **M = R + 1**

Path Base Testing: Step 2



- Cyclomatic complexity, M =
  - The number of 'regions' in the graph(R) + 1
  - = 2 +1 = 3

Path Base Testing: Step 2

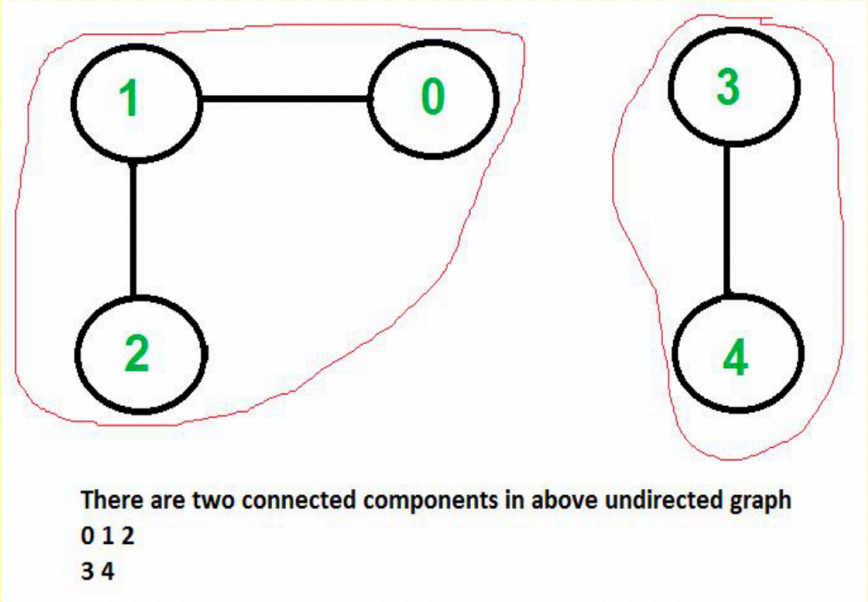


- M = Number of 'predicate' node (P) + 1
- In this example:
  - Predicates, P = 2
    - (Node 2 and 3)
  - Cyclomatic Complexity, M
    = 2 + 1
    = 3

BRAC

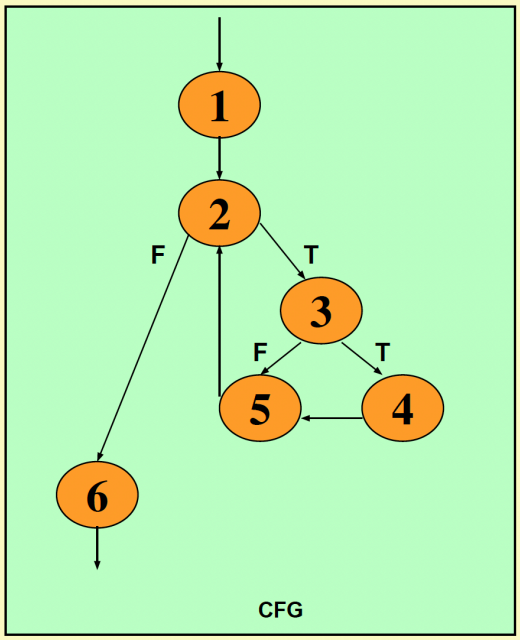
Path Base Testing: Step 2

Connected Components in a Graph



BRAC

Path Base Testing: Step 2



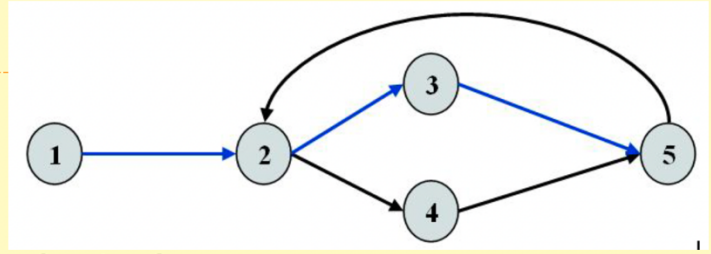
- Cyclomatic complexity, M = **E – N + 2P**
- E, edges = 7
- N, nodes = 6
- P, connected components = 1
- = 7 – 6 + (2x1)
- = 3

INDEPENDENT PATH

An Independent Path is a path through the graph that introduces at least one new edge not covered by any previously selected paths. Must move along at least one arc that has not been yet traversed (an unvisited arc). The objective is to cover all statements in a program by independent paths.

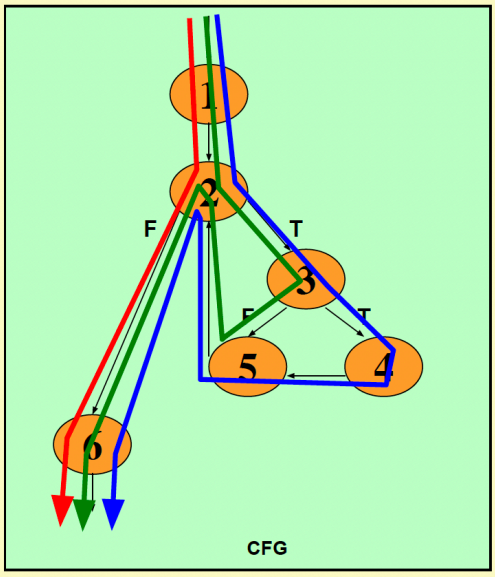
- The number of independent paths to discover <= Cyclomatic complexity number, M.
- The set of independent paths is called Basic Path Set.

Example



- M = Regions + 1 = 2 + 1 = 3
- 1-2-3-5 can be the first independent path; 1-2-4-5 is another; 1-2-3-5-2-4-5 is one more.
- Alternatively, if we had identified 1-2-3-5-2-4-5 as the first independent path, there would be no more independent paths.
- The number of independent paths therefore can vary according to the order we identify them.

Path Base Testing: Step 3



- Cyclomatic complexity = 3.
- Need at most **3** independent paths to cover the CFG.
- In this example:
  - [1 – 2 – 6]
  - [1 – 2 – 3 – 5 – 2 – 6]
  - [1 – 2 – 3 – 4 – 5 – 2 – 6]

BRAC

Path Base Testing: Step 4

- Prepare a test case for each independent path.
- In this example:
  - Path: [1 – 2 – 6]
    - Test Case: A = { 5, …}, N = 1
    - Expected Output: 5

```
min = A[0];                    1
I = 1;

while (I < N) {                2
  if (A[I] < min)              3
        min = A[I];            4
     I = I + 1;                5
}
print min                      6
```

SPECIALIZATION INDEX (SIX)

The Specialization Index metric measures the extent to which subclasses override their ancestors' classes. This index is the ratio between the number of overridden methods and total number of methods in a Class, weighted by the depth of inheritance for this class.

The metric provides a percentage, where the class contains at least one operation. For a root class, the specialization indicator is zero. Nominal range is between 0 % and 120 %.

The ... variable	Represents the ...
DIT	Depth of inheritance
NMA	The number of operations added to the inheritance
NMI	The number of inherited operations
NMO	The number of overridden methods

Formula :

$$SIX = \frac{NMO \times DIT}{NMO + NMA + NMI}$$

```
Class Person{
 void read();
 void display();
}
Class Student extends Person{
 void read();
void display();
Void getAverage();
}
Class GraduateStudent extends Student{
 void read();
void display();

Void workStatus();

}
```

$$SIX = \frac{2 \times 2}{2 + 1 + 1} \times 100\% = 100\%$$

CODE SMELL & REFACTORING

REFACTORING

Refactoring is the process of restructuring existing code without changing its external behavior. It is done to improve the internal structure of the code, making it more maintainable, readable, and efficient.

KEY PRINCIPLES OF REFACTORING

Preserve Behavior: Refactoring should not alter the external functionality of the code.

Incremental Changes: Make small, incremental improvements to ensure the system remains stable.

Automated Tests: Having a reliable suite of tests ensures that refactoring doesn't break the existing functionality.

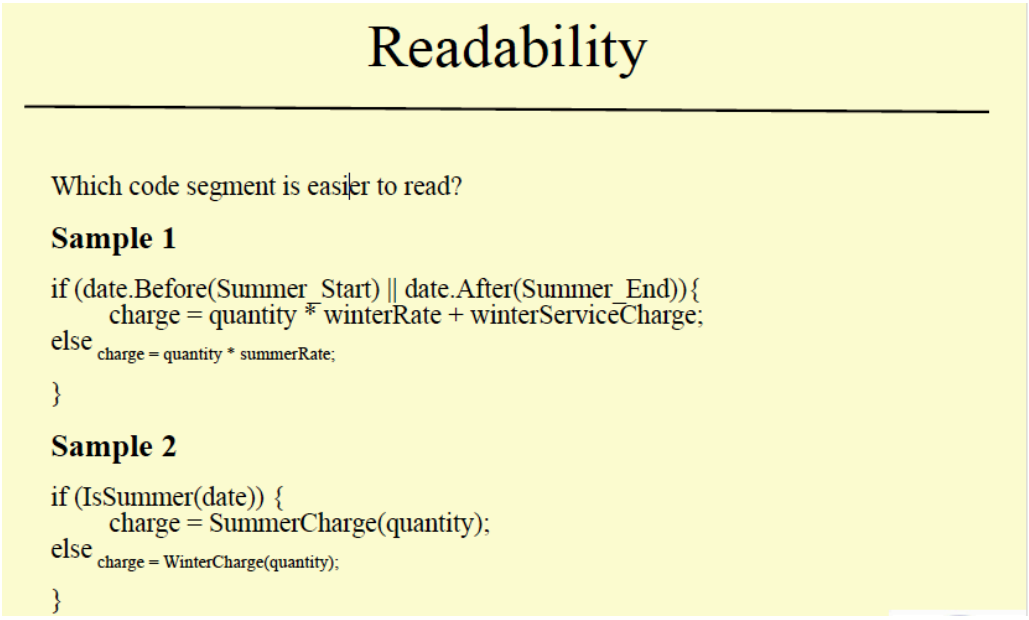
BENEFITS OF REFACTORING

Improved Readability: Cleaner code is easier to read and understand.

Enhanced Maintainability: Easier to fix bugs and add features.

Reduced Technical Debt: Minimizes the cost of changes in the future.

Improved Performance: Although not the primary goal, refactoring can lead to more efficient code.



WHEN TO REFACTOR

During Code Reviews: Fix code smells and improve readability.

Before Adding Features: Simplify the code to support new functionality.

After Fixing Bugs: Make the code robust and prevent similar issues.

When Code Is Hard to Understand: Simplify and clarify complex areas.

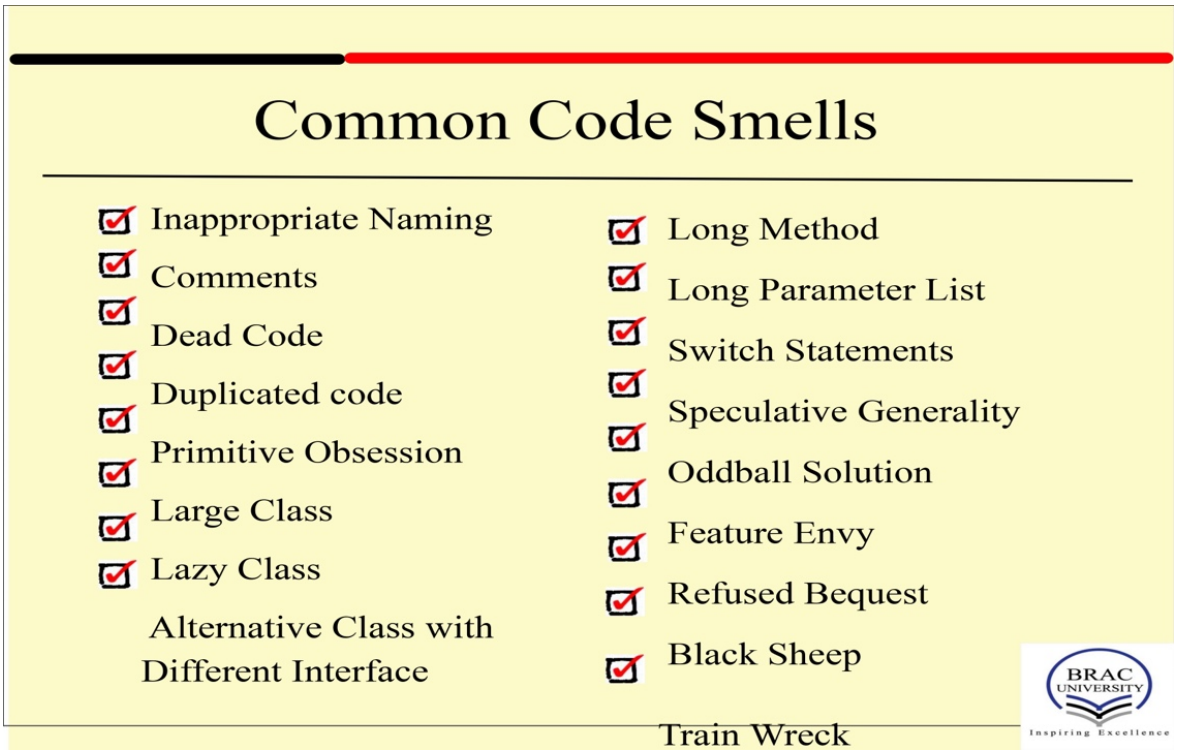
When Code Duplication Exists: Eliminate redundant logic.

Changing Requirements: Adapt the code for new needs.

CODE SMELL

A code smell refers to any characteristic in the source code of a program that may indicate a deeper problem. It doesn't necessarily break the code but suggests that the code could be improved. Code smells often indicate areas where the code is poorly designed or not easily maintainable. Recognizing code smells is a skill that helps developers write cleaner, more efficient, and maintainable code.

COMMON TYPES OF CODE SMELLS



Inappropriate Naming

- Poorly chosen names that don't describe purpose or function.
- Makes code harder to understand.

Comments

- Overused, outdated, or redundant comments instead of clear code.
- Suggests the code itself is unclear or overly complex.

Dead Code

- Code that is no longer used or executed.
- Creates clutter and confusion.

Duplicated Code

- Repetition of the same code in multiple places.
- Leads to inconsistency and harder maintenance.

Primitive Obsession

- Overuse of primitive types instead of custom classes or objects.
- Reduces code expressiveness and increases complexity.

Large Class

- A class with too many responsibilities.
- Violates the Single Responsibility Principle.

Lazy Class

- A class that doesn't do enough to justify its existence.
- Could be merged or removed.

Alternative Classes with Different Interfaces

- Two classes performing similar tasks with inconsistent interfaces.
- Makes switching between them or extending functionality harder.

Long Method

- A method that is too long and difficult to read or understand.
- Suggests the need for method extraction.

Long Parameter List

- Excessive parameters in a method, making it hard to use.
- Could benefit from grouping related parameters into objects.

Switch Statements

- Repeated switch or if conditions instead of polymorphism.
- Indicates poor design flexibility.

Speculative Generality

- Overly abstract code designed for potential future use that never materializes.
- Leads to unnecessary complexity.

Oddball Solution

- Unique, inconsistent ways of solving similar problems.
- Breaks cohesion and uniformity.

Feature Envy

- A class or method overly dependent on another class's data.
- Suggests poor encapsulation.

Refused Bequest

- A subclass doesn't use or need inherited methods or properties.
- Indicates a flawed inheritance hierarchy.

Black Sheep

- Poorly written or out-of-place code that doesn't fit the rest of the system.
- Stands out as inconsistent or substandard.

Train Wreck

- Multiple method calls chained together.
- Leads to brittle and hard-to-read code.

Code Smell - Inappropriate Naming

- Names given to variables (fields) and methods should be clear and meaningful.
- A variable name should say exactly what it is.
  - Which is better?
    - private string s; OR **private string salary;**
- A method should say exactly what it does.
  - Which is better?
    - public double calc (double s)
    - **public double calculateFederalTaxes (double salary)**

BRAC

Code Smell - Comments

- Comments are often used as deodorant
- Comments represent a *failure to express an idea in the code*. Try to make your code self-documenting or intention-revealing
- When you feel like writing a comment, first try "to refactor so that the comment becomes superfluous
- **Remedies:**
  - Extract Method
  - Rename Method
  - Introduce Assertion

Comment: "Grow the Array" smells

```
public class MyList
  {
    int INITIAL_CAPACITY = 10;
    bool m_readOnly;
    int m_size = 0;
    int m_capacity;
    string[] m_elements;

    public MyList()
    {
      m_elements = new string[INITIAL_CAPACITY];
      m_capacity = INITIAL_CAPACITY;
    }

    int GetCapacity() {
      return m_capacity;
    }
}
```

```
void AddToList(string element)
{
  if (!m_readOnly)
  {
    int newSize = m_size + 1;
    if (newSize > GetCapacity())
    {
      // grow the array
      m_capacity += INITIAL_CAPACITY;
      string[] elements2 = new string[m_capacity];
      for (int i = 0; i < m_size; i++)
        elements2[i] = m_elements[i];

      m_elements = elements2;
    }
    m_elements[m_size++] = element;
  }
}
```

BRAC

Comment Smells Make-over

```
void AddToList(string element)
{
  if (m_readOnly)
    return;
  if (ShouldGrow())
  {
    Grow();
  }
  StoreElement(element);
}
```

```
private void Grow()
{
  m_capacity += INITIAL_CAPACITY;
  string[] elements2 = new string[m_capacity];
  for (int i = 0; i < m_size; i++)
    elements2[i] = m_elements[i];

  m_elements = elements2;
}
```

```
private void StoreElement(string element)
{
  m_elements[m_size++] = element;
}
```

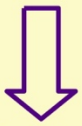
```
private bool ShouldGrow()
{
  return (m_size + 1) > GetCapacity();
}
```

BRAC UNIVERSITY
Inspiring Excellence

Rename Method

Extract Method

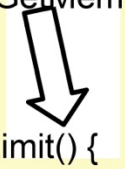
```
void PrintOwning(double amount){
    PrintBanner();

    // print details
    System.Console.Out.WriteLine("name: "+ name);
    System.Console.Out.WriteLine("amount: "+ amount);
}
```



```
void PrintOwning(double amount){
    PrintBanner();
    PrintDetails(amount);
}

void PrintDetails(double amount){
    System.Console.Out.WriteLine("name: "+ name);
    System.Console.Out.WriteLine("amount: "+ amount);
}
```

Introduce Assertion

```
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :
    _primaryProject.GetMemberExpenseLimit();
}
```



```
double getExpenseLimit() {
    Assert(_expenseLimit != NULL_EXPENSE || _primaryProject != null,
    "Both Expense Limit and Primary Project must not be null");

    return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :
        _primaryProject.GetMemberExpenseLimit();
}
```


Code Smell - Long Method

- A method is long when it is too hard to quickly comprehend.
- Long methods tend to hide behavior that ought to be shared, which leads to duplicated code in other methods or classes.
- Good OO code is easiest to understand and maintain with shorter methods with good names
- **Remedies:**
  - Extract Method
  - Replace Temp with Query
  - Introduce Parameter Object
  - Preserve Whole Object
  - Decompose Conditional

Long Method Example

```
private String toStringHelper(StringBuffer result)
{
   result.append("<");
   result.append(name);
   result.append(attributes.toString());
   result.append(">");
   if (!value.equals(""))
      result.append(value);
   Iterator it = children().iterator();
   while (it.hasNext())
   {
      TagNode node = (TagNode)it.next();
      node.toStringHelper(result);
   }
   result.append("</");
   result.append(name);
   result.append(">");
   return result.toString();
}
```

Example Html tag:
<name> Jannet Jhonson </name>

Long Method Makeover (Extract Method)

```
private String toStringHelper(StringBuffer result)
{
   writeOpenTagTo(result);
   writeValueTo(result);
   writeChildrenTo(result);
   writeEndTagTo(result);
   return result.toString();
}

private void writeOpenTagTo(StringBuffer result)
{
   result.append("<");
   result.append(name);
   result.append(attributes.toString());
   result.append(">");
}
private void writeEndTagTo(StringBuffer result)
{
   result.append("</");
   result.append(name);
   result.append(">");
}
```

```
private void writeValueTo(StringBuffer result)
{
   if (!value.equals(""))
      result.append(value);
}

private void writeChildrenTo(StringBuffer result)
{
   Iterator it = children().iterator();
   while (it.hasNext())
   {
      TagNode node = (TagNode)it.next();
      node.toStringHelper(result);
   }
}
```

BRAC UNIVERSITY
Inspiring Excellence

Replace Temp with Query

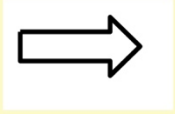
```
Method1(){
  double basePrice = _quanity * _itemPrice;
if(basePrice > 1000) {
   return basePrice * 0.95
  }
else{
   return basePrice*0.98
  }
}
 Method2(){
  double basePrice = _quanity * _itemPrice;
  return basePrice + 100;
}
```

What if the basePrice calculation equation changes ??
-- We would need to change two lines in the code

Smell: Long Method

Replace Temp with Query

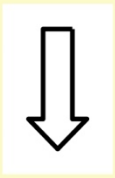
```
Method1(){
  double basePrice = _quanity * _itemPrice;
if(basePrice > 1000) {
   return basePrice * 0.95
  }
else{
   return basePrice*0.98
  }
}
```



```
Method1(){
if(getBasePrice() > 1000) {
    return getBasePrice() * 0.95;
}
else {
     return getBasePrice() * 0.98;
 }

}
double getBasePrice()  {
     return _quanitiy * itemPrice;
}
```

Replace Temp with Query

```
Method2(){
 double basePrice = _quanity * _itemPrice;
 return basePrice + 100;
}
```



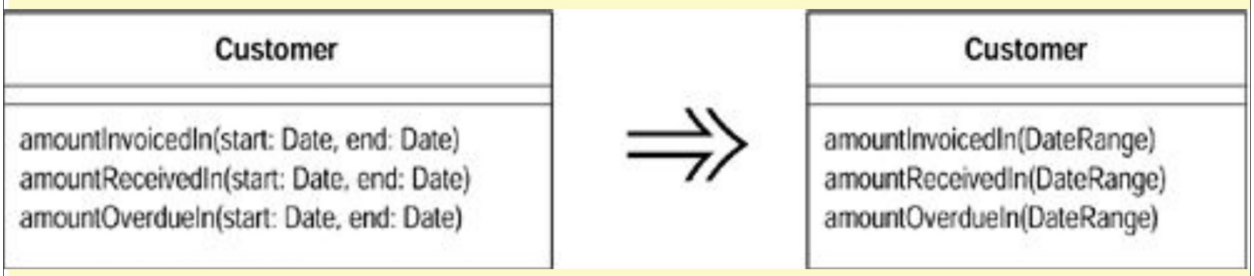
```
double getBasePrice()  {
    return _quanitiy * itemPrice;
}
```

```
Method2(){
 return getBasePrice() + 100;
}
```


Introduce Parameter Object

```
int MethodTooManyParameter (Date start, Date end, int value, string
                                    month, string yearStart, string yearEnd)
 {
        // method body
 }
```

Introduce Parameter Object




```
Class DateRange{
    Date start;
    Date end;
}
```


Preserve Whole Object

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

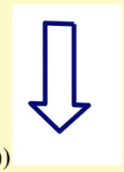
```
withinPlan = plan.withinRange(daysTempRange());
```

Decompose Conditional

You have a complicated conditional (if-then-else) statement.

Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
  charge = quantity * _winterRate + _winterServiceCharge;
 else charge = quantity * _summerRate;
```



```
if (notSummer(date))
  charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

Example of Conditional Complexity

```
public bool ProvideCoffee(CoffeeType coffeeType)
{
        if(_change < _CUP_PRICE || !AreCupsSufficient || !IsHotWaterSufficient || !IsCoffeePowderSufficient)
        {
                return false;
        }
        if((coffeeType == CoffeeType.Cream || coffeeType == CoffeeType.CreamAndSugar) && !IsCreamPowderSufficient)
        {
                return false;
        }
        if((coffeeType == CoffeeType.Sugar || coffeeType == CoffeeType.CreamAndSugar) && !IsSugarSufficient)
        {
                return false;
        }

        _cups--;
        _hotWater -= _CUP_HOT_WATER;
        _coffeePowder -= _CUP_COFFEE_POWDER;
        if(coffeeType == CoffeeType.Cream || coffeeType == CoffeeType.CreamAndSugar)
        {
                _creamPowder -= _CUP_CREAM_POWDER;
        }
        if(coffeeType == CoffeeType.Sugar || coffeeType == CoffeeType.CreamAndSugar)
        {
                _sugar -= _CUP_SUGAR;
        }

        ReturnChange();
        return true;
}
```

Code Smell- Long Parameter List

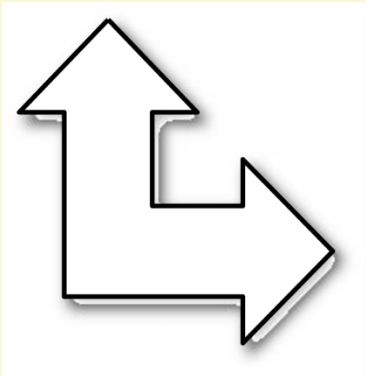
- Methods that take too many parameters produce client code that is awkward and difficult to work with.
- **Remedies:**
  - Introduce Parameter Object
  - Replace Parameter with Method
  - Preserve Whole Object

Example

```
private void createUserInGroup() {
   GroupManager groupManager = new GroupManager();
   Group group = groupManager.create(TEST_GROUP, false,
     GroupProfile.UNLIMITED_LICENSES, "",
                  GroupProfile.ONE_YEAR, null);
   user = userManager.create(USER_NAME, group, USER_NAME, "jack",
              USER_NAME, LANGUAGE, false, false, new Date(),
                  "blah", new Date());
}
```

Introduce Parameter Object

Customer
AmoutInvoicedIn(Date start, Date end) AmoutRecivedIn(Date start, Date end) AmoutOverdueIn(Date start, Date end)



Customer
AmoutInvoicedIn(DateRange range) AmoutRecivedIn(DateRange range) AmoutOverdueIn(DateRange ran

BRAC

Replace Parameter with Method

```
public double getPrice() {
   int basePrice = _quantity * _itemPrice;
   int discountLevel;
   if (_quantity > 100)
     discountLevel = 2;
   else
     discountLevel = 1;
   double finalPrice = discountedPrice (basePrice, discountLevel);
   return finalPrice;
}

private double discountedPrice (int basePrice, int discountLevel) {
   if (discountLevel == 2)
     return basePrice * 0.1;
   else
     return basePrice * 0.05;
}
```

BRAC UNIVERSITY
Inspiring Excellence

Replace Parameter with Method

```
public double getPrice() {
   int basePrice = _quantity * _itemPrice;
   int discountLevel = getDiscountLevel();
   double finalPrice = discountedPrice (basePrice, discountLevel);
   return finalPrice;
}

private int getDiscountLevel() {
   if (_quantity > 100) return 2;
   else return 1;
}
private double discountedPrice (int basePrice, int discountLevel) {
   if (getDiscountLevel() == 2) return basePrice * 0.1;
   else return basePrice * 0.05;
}
```

Smell: Long Parameter List

Replace Parameter with Method

```
public double getPrice() {
   int basePrice = _quantity * _itemPrice;
   int discountLevel = getDiscountLevel();
   double finalPrice = discountedPrice (basePrice);
   return finalPrice;
}

private double discountedPrice (int basePrice) {
   if (getDiscountLevel() == 2) return basePrice * 0.1;
   else return basePrice * 0.05;
}
```

Preserve Whole Object

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

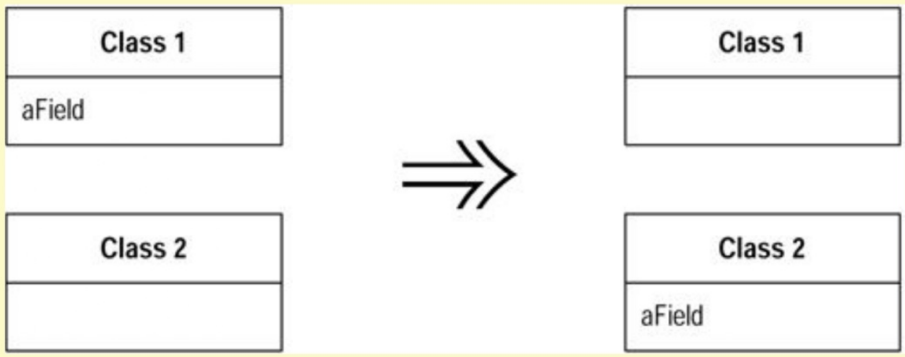
Feature Envy

- A method that seems more interested in some other class than the one it is in.
- Data and behavior that acts on that data belong together. When a method makes too many calls to other classes to obtain data or functionality, Feature Envy is in the air.
- **Remedies:**
  - Move Field
  - Move Method
  - Extract Method

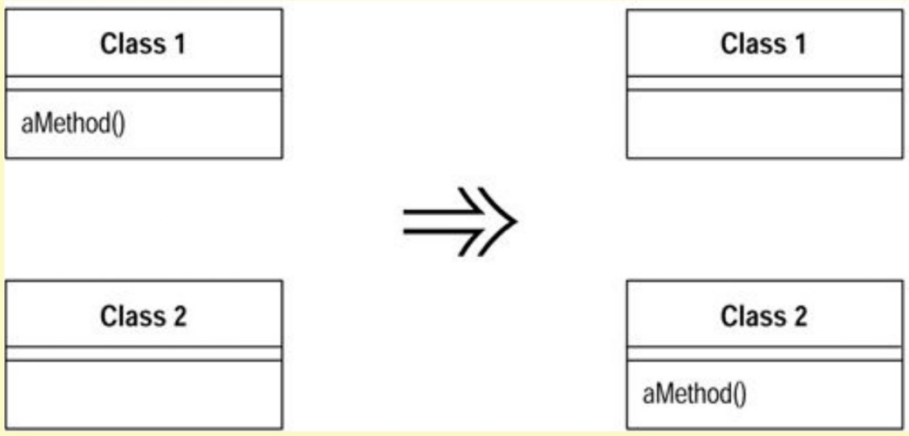
Example

```
Public class CapitalStrategy{
      double capital(Loan loan)
      {
         if (loan.getExpiry() == NO_DATE && loan.getMaturity() != NO_DATE)
            return loan.getCommitmentAmount() * loan.duration() * loan.riskFactor();

         if (loan.getExpiry() != NO_DATE && loan.getMaturity() == NO_DATE)
         {
            if (loan.getUnusedPercentage() != 1.0)
               return loan.getCommitmentAmount() * loan.getUnusedPercentage() *
      loan.duration() * loan.riskFactor();
            else
               return (loan.outstandingRiskAmount() * loan.duration() * loan.riskFactor()) +
                  (loan.unusedRiskAmount() * loan.duration() * loan.unusedRiskFactor());
         }

         return 0.0;
      }
}
```


Move Field



Move Method



Duplicated Code

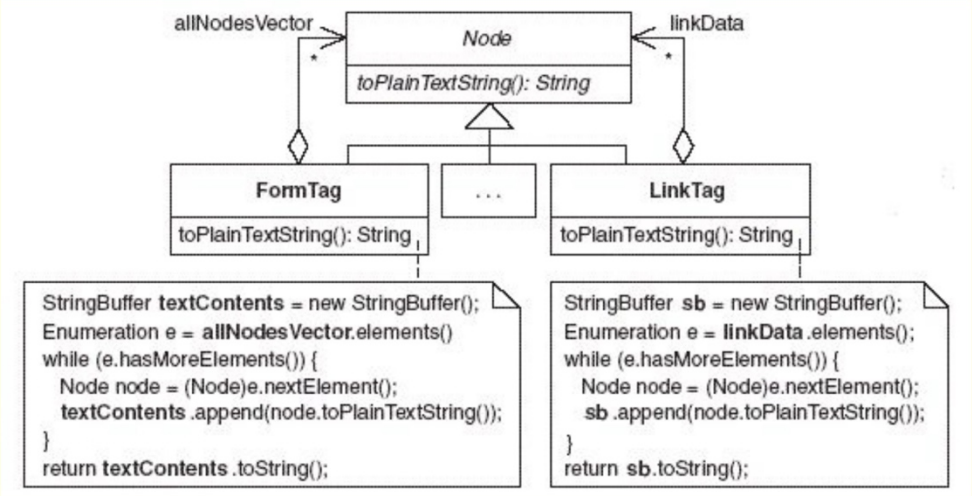
- The *most pervasive and pungent smell* in software
- There is obvious or blatant duplication
  - Such as copy and paste
- There are subtle or non-obvious duplications
  - Similar algorithms
- **Remedies**
  - Extract Method
  - Pull Up Field
  - Form Template Method
  - Substitute Algorithm

Ctl+C Ctl+V Pattern

```
public static MailTemplate getStaticTemplate(Languages language) {
        MailTemplate mailTemplate = null;
        if(language.equals(Languages.English)) {
                mailTemplate = new EnglishLanguageTemplate();
        } else if(language.equals(Languages.French)) {
                mailTemplate = new FrenchLanguageTemplate();
        } else if(language.equals(Languages.Chinese)) {
                mailTemplate = new ChineseLanguageTemplate();
        } else {
                throw new IllegalArgumentException("Invalid language type specified");
        }
        return mailTemplate;
}

public static MailTemplate getDynamicTemplate(Languages language, String content) {
        MailTemplate mailTemplate = null;
        if(language.equals(Languages.English)) {
                mailTemplate = new EnglishLanguageTemplate(content);
        } else if(language.equals(Languages.French)) {
                mailTemplate = new FrenchLanguageTemplate(content);
        } else if(language.equals(Languages.Chinese)) {
                mailTemplate = new ChineseLanguageTemplate(content);
        } else {
                throw new IllegalArgumentException("Invalid language type spec
        }
        return mailTemplate;
}
```

BRAC UNIVERSITY
Inspiring Excellence

Example Of Obvious Duplication



```
private void AddOrderMaterials(int iOrderId)
{


    if (iOrderType == 1)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 2)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        OrderMaterial oOrderMaterialCream = new OrderMaterial();
        oOrderMaterialCream.MaterialId = 2;
        oOrderMaterialCream.OrderId = iOrderId;
        oOrderMaterialCream.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCream);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 3)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        OrderMaterial oOrderMaterialSugar = new OrderMaterial();
        oOrderMaterialSugar.MaterialId = 3;
        oOrderMaterialSugar.OrderId = iOrderId;
        oOrderMaterialSugar.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialSugar);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 4)
```

BRAC UNIVERSITY
Inspiring Excellence

Levels of Duplication

Literal Duplication

Same for loop in 2 places

Semantic Duplication

1st Level - For and For Each Loop

2nd Level - Loop v/s Lines repeated

```
for(int i : asList(1,3,5,10,15))
stack.push(i);

v/s

for(int i=0;i<5;i++){
  stack.push(asList(i));
}
```

```
stack.push(1); stack.push(3);
stack.push(5); stack.push(10);
stack.push(15);

v/s

for(int i : asList(1,3,5,10,15))
stack.push(i);
```

Data Duplication

Some constant declared in 2 classes (test and production)

Conceptual Duplication

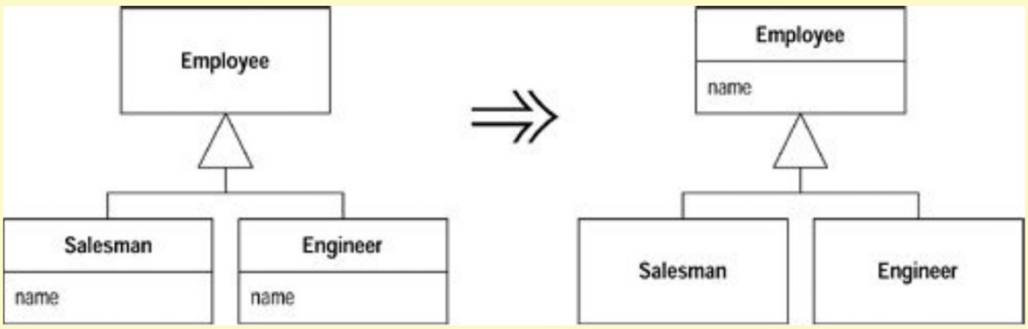
2 Algorithm to Sort elements (Bubble sort and Quick sort)

Logical Steps - Duplication

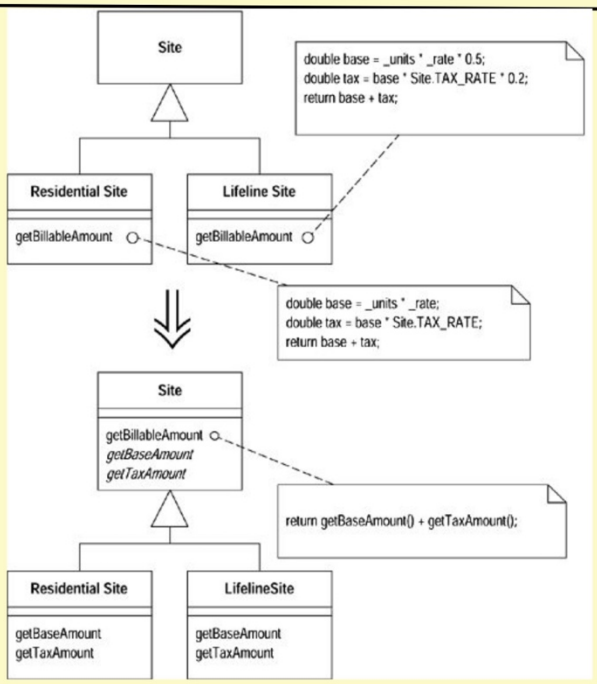
Same set of steps repeat in different scenarios.

Ex: Same set of validations in various points in your applications

Pull Up Field



Form Template Method



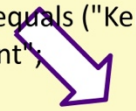
Substitute Algorithm

```
String foundPerson(String[] people){
for (int i = 0; i < people.length; i++) {
    if (people[i].equals ("Don")){
        return "Don";
    }
    if (people[i].equals ("John")){
        return "John";
    }
    if (people[i].equals ("Kent")){
        return "Kent";
    }
}
return ""; }
```

Smell: Duplicate Code

Substitute Algorithm

```
String foundPerson(String[] people){

for (int i = 0; i < people.length; i++) {
    if (people[i].equals ("Don")){
        return "Don";
    }
    if (people[i].equals ("John")){
        return "John";
    }
    if (people[i].equals ("Kent")){
        return "Kent";
    }
}
return "";
}
```



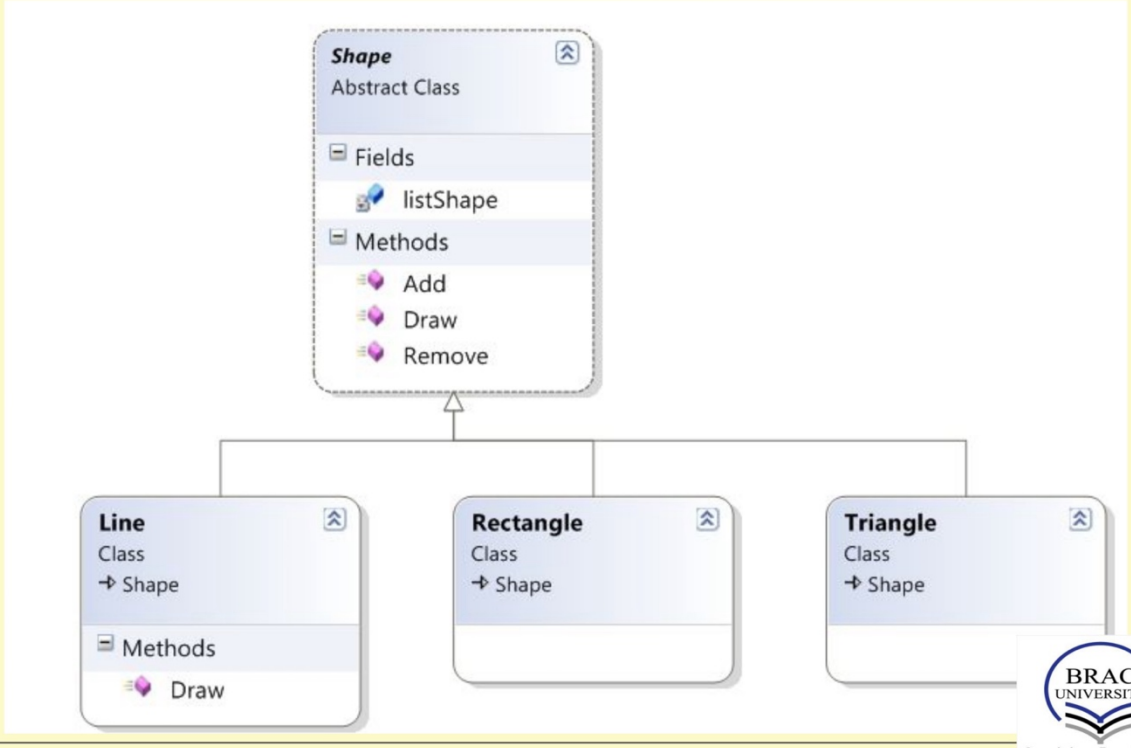
```
String foundPerson(String[] people){
    List candidates = Arrays.asList(new String[]  {"Don",
"John", "Kent"});
    for (String person : people)
        if (candidates.contains(person))
            return person;
    return "";
}
```

Refused Bequest

- This rather potent odor results when *subclasses inherit code that they don't want*. In some cases, a subclass may "refuse the bequest" by providing a *do-nothing implementation* of an inherited method.
- Remedies
  - Push Down Field
  - Push Down Method

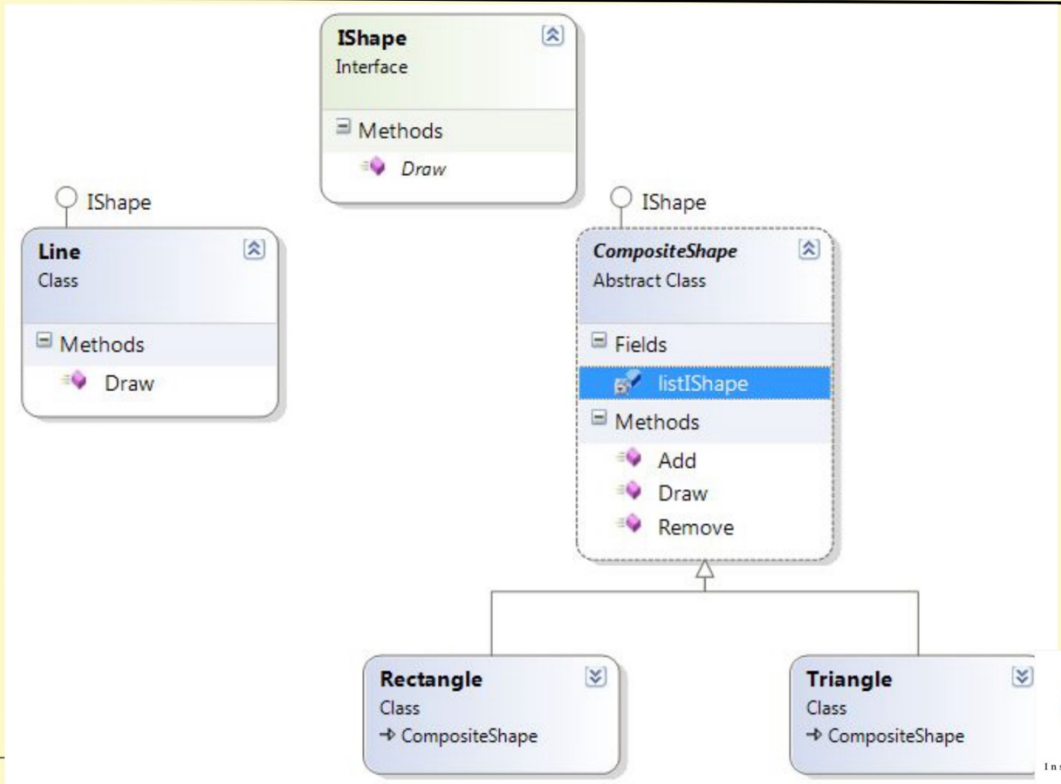
Smell: Refused Bequest

Example of Refused Bequest



BRAC UNIVERSITY
Inspiring Excellence

Refused Bequest Make Over



References

[F] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 2000

[K] Kerievsky, Joshua. *Refactoring to Patterns*. Boston, MA: Addison-Wesley, 2005