OBJECT ORIENTED PROGRAMMING (OOP)

ASSIGNMENT 1

Name	Section	Roll No.
Muhammad Rafay	C	CT-24141
Haris Zamir	C	CT-24125
Muhammad Nizam	C	CT-24147
Muhammad Raza	C	CT-24138

Title:

Design a system that encrypts and decrypts files using different encryption algorithms (AES, RSA, DES) with a flexible algorithm selection mechanism.

Objective:

To encrypt and decrypt files with various different algorithms, each with different specialties and setbacks, and to decide which algorithm the user wants depending on their input. This is done using three different encryption/decryption algorithms which include **AES** (Advanced Encryption Standard), **RSA** (Rivest–Shamir–Adleman) and **DES** (Data Encryption Standard).

Approach:

- We create three different classes for each algorithm, namely **AES**, **RSA** and **DES**.
- We implement algorithmic logic code in each of the three classes to make it functional.
- In all the classes, we will name two protected methods **encrypt** and **decrypt** so that the user be redirected to the algorithm they desire through the parent class.
- We create the **CipherInterface** interface class which contains four pure protected virtual methods which are **encrypt**, **decrypt**, **preprocess** and **postprocess** along with a virtual destructor.
- Inside the **CipherInterface** class, we write the statements for the **encrypt** and **decrypt** method making them follow a pattern which follows preprocess > encrypt/decrypt > postprocess (each of these has a different definition in their respective classes) making it follow the Template design pattern.
- We make **CipherInterface** friends with **Cipher**, so that Cipher can access the protected methods in each class.
- We create the **Cipher** class which follows the Strategy design pattern with three public methods **encrypt**, **decrypt** and **setMode** (which takes a CipherInterface pointer as input)
- All classes (AES, DES, RSA) will inherit from the **CipherInterface** abstract class.

- To use them in the main program, we instantiate a **Cipher** object and pass a class object (AES, DES, RSA) into its constructor as reference. This will set that class object as the current strategy.
- To change strategies (cryptography type) we will use the **setMode** method available in the **Cipher** class object.

Design Patterns:

The design patterns used in the above program are:

- Factory Method
- Template Method

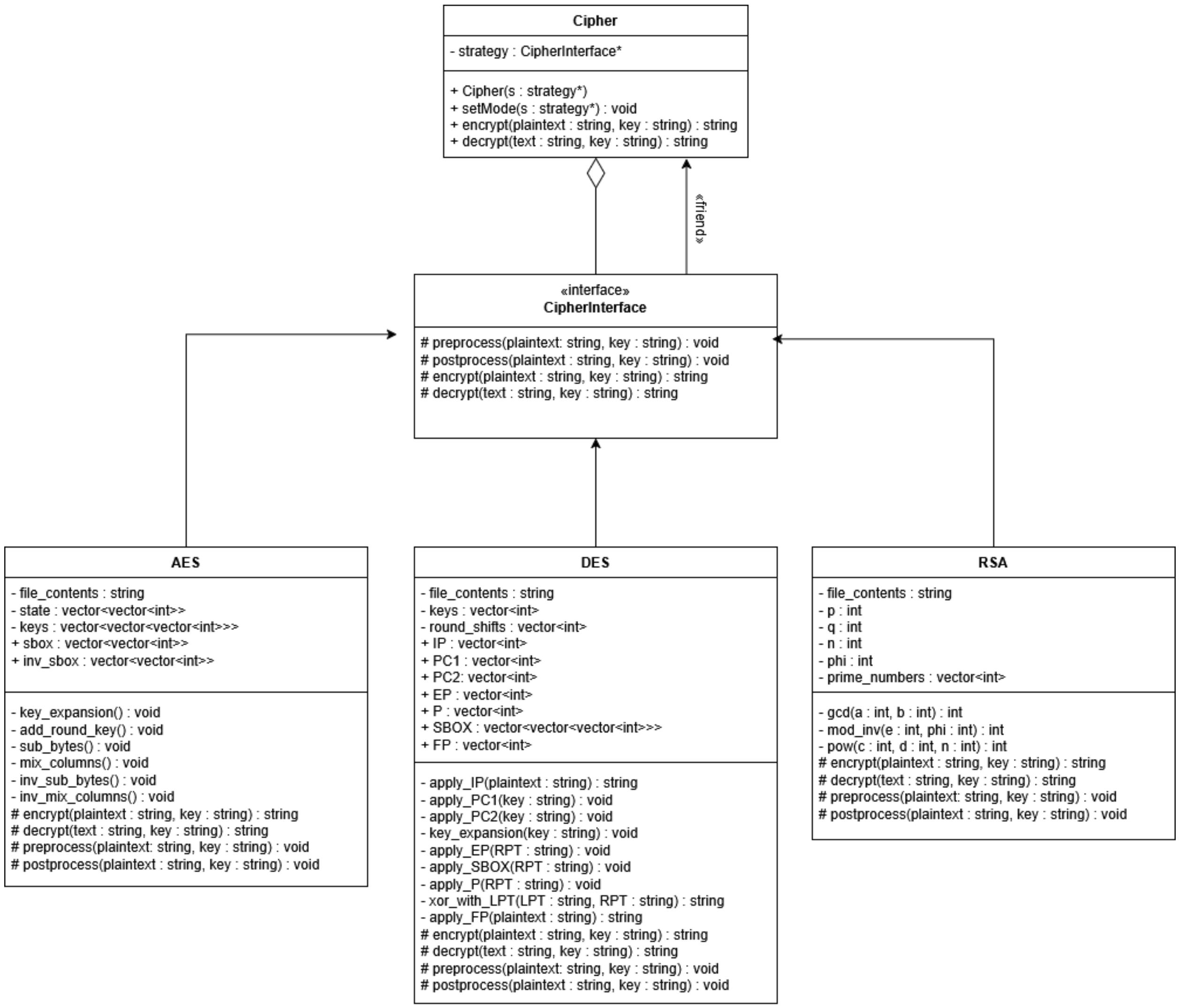
Since the **Cipher** class provides an interface for creating objects and also allows child classes to change the object type on runtime, it follows the **Factory Method** according to its definition:

"Provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created."

Similarly, since the **CipherInterface** class defines the structures of the algorithms (AES, DES, RSA) and executes specific steps with varying results based on the subclasses. It follows the **Template Method** according to its definition:

"Defines the structure of an algorithm in a base class but allows subclasses to provide specific implementations of certain steps."

UML Diagram:



How will we encrypt and decrypt files?

In the *main(int argc, char* argv[])* function, the user will pass files they want to encrypt or decrypt (the last argument will tell the program whether it will encrypt or decrypt the files passed). As we know, encryption sizes are limited based on the algorithm, so we will encrypt block by block until we reach the end of the file and store the result. The same process will be followed on decryption. More files can also be passed in the *argv* and the program will loop through them storing the result in new files.

Cryptography Algorithms

Let's understand the inner working of the three main cryptography algorithms to get a better idea at how they encrypt and decrypt information and what problem they solve.

Advanced Encryption Standard *(128-bit)*

MAKING OF ROUNDKEYS

1. Input a 16-byte key from the user and convert it to Hexadecimal. This will be Round Key 0

Round Key 0 (Initial Key):

```
Word 0: 4D 59 5F 53
Word 1: 45 43 52 45
Word 2: 54 5F 4B 45
Word 3: 59 31 32 33
```

2. 10 more Round Keys are made to conduct the encryption process, using the formulas below

For each new word `W[i]`:

- If `i` is a multiple of 4 (e.g., 4, 8, 12, ...):

```
W[i] = W[i-4] XOR SubBytes(RotWord(W[i-1])) XOR Rcon[i/4]
```

- Otherwise:

```
W[i] = W[i-4] XOR W[i-1]
```

1. **RotWord(W[3])**:
   - `W[3] = 59 31 32 33`
   - Rotate left by 1: `31 32 33 59`

Round	Rcon Value (Hex)
1	01 00 00 00
2	02 00 00 00
3	04 00 00 00
4	08 00 00 00
5	10 00 00 00
6	20 00 00 00
7	40 00 00 00
8	80 00 00 00
9	1B 00 00 00
10	36 00 00 00

S BOX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

ENCRYPTION

1. Input plain text from the user, convert it to 16 bytes, and convert to hexadecimal

```
Plaintext: HELLO WORLD
16 BYTE: HELLO WORLD     (5 SPACES)
HEXADECIMAL: 48 45 4C 4C 4F 20 57 4F 52 4C 44 20 20 20 20 20
```

2. Round 0

   1. XOR the plaintext with **Round Key 0**:

```
Plaintext:   48 45 4C 4C 4F 20 57 4F 52 4C 44 20 20 20 20 20
Round Key 0: 4D 59 5F 53 45 43 52 45 54 5F 4B 45 59 31 32 33
XOR Result:  05 1C 13 1F 0A 63 05 0A 06 13 0F 65 79 11 12 13
```

3. Round (1-9) (Repeat 9 times)

- Apply S Box on the result of previous round.

Result:

```
6B C5 7D 76 67 2B 6B 67 6F 7D 76 2B 6B C5 7D 76
```

- Shift Rows

```
[6B, C5, 7D, 76]
[67, 2B, 6B, 67]
[6F, 7D, 76, 2B]
[6B, C5, 7D, 76]
```

```
[6B, C5, 7D, 76]
[2B, 6B, 67, 67]
[76, 2B, 6F, 7D]
[76, 6B, C5, 7D]
```

- **Row 0**: No shift.
- **Row 1**: Shift left by 1 byte.
- **Row 2**: Shift left by 2 bytes.
- **Row 3**: Shift left by 3 bytes.

- Mix Columns:

0 1 2

Column 0: `[6B, 2B, 76, 76]`

Yellow values differ subject to column (The rest remain same)

The MixColumns operation for Column 0 is:

```
NewByte0 = (02 * 6B) XOR (03 * 2B) XOR (01 * 76) XOR (01 * 76)
NewByte1 = (01 * 6B) XOR (02 * 2B) XOR (03 * 76) XOR (01 * 76)
NewByte2 = (01 * 6B) XOR (01 * 2B) XOR (02 * 76) XOR (03 * 76)
NewByte3 = (03 * 6B) XOR (01 * 2B) XOR (01 * 76) XOR (02 * 76)
```

Updated Column 0:

After MixColumns, Column 0 becomes:

```
[6B]
[71]
[96]
[17]
```

Final State Matrix After MixColumns

After performing MixColumns on all columns, the updated state matrix is:

Copy

```
[6B, 77, E2, 5E]  (Row 0)
[71, C5, DF, DE]  (Row 1)
[96, 5E, DF, F0]  (Row 2)
[17, 96, 99, FB]  (Row 3)
```

- XOR with respective round key (in this case, round key 1)

```
State Matrix: [6B, 77, E2, 5E]
              [71, C5, DF, DE]
              [96, 5E, DF, F0]
              [17, 96, 99, FB]

Round Key 1:  [8B, 7A, 9C, 98]
              [CE, 39, CE, DD]
              [9A, 66, 85, 98]
              [C3, 57, B7, AB]
```

XOR

```
XOR Result:    [E0, 0D, 7E, C6]
               [BF, FC, 11, 03]
               [0C, 38, 5A, 68]
               [D4, C1, 2E, 50]
```

4. Round 10
   - Repeat Step 3 (Excluding MixColumns)
   - The result of Round 10 will be the final encrypted text

DECRYPTION

1. Starts with Round key 10 and ends with round key 1
2. Round 10

   1. XOR the ciphertext with **Round Key 10**:

Copy

```
Ciphertext:   3A 7F 2B 9C 1D 4E 8A 6F 0B 5D 3C 7E 9A 2F 4B 8C
Round Key 10: (Example) 36 1B 2C 3D 4E 5F 6A 7B 8C 9D AE BF CD DE EF F0
XOR Result:   0C 64 07 A1 53 11 E0 14 87 C0 92 C1 57 F1 A4 7C
```

3. Round 9-1 (Repeat 9 times)

   - Inverse Shift Rows

```
[0C, 64, 07, A1]
[53, 11, E0, 14]
[87, C0, 92, C1]
[57, F1, A4, 7C]
```

```
[0C, 64, 07, A1]
[14, 53, 11, E0]
[92, C1, 87, C0]
[7C, 57, F1, A4]
```

After InvShiftRows:

- **Row 0**: `[0C, 64, 07, A1]` (no shift).
- **Row 1**: `[14, 53, 11, E0]` (shift right by 1).
- **Row 2**: `[92, C1, 87, C0]` (shift right by 2).
- **Row 3**: `[7C, 57, F1, A4]` (shift right by 3).

- Apply Inverse S Box

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

```
[81, 8C, 38, F1]
[9B, 50, E3, A0]
[74, DD, EA, 1F]
[01, DA, 2B, 1D]
```

```
[0C, 64, 07, A1]
[14, 53, 11, E0]
[92, C1, 87, C0]
[7C, 57, F1, A4]
```

- XOR with respective Round Key

```
[81, 8C, 38, F1]
[9B, 50, E3, A0]
[74, DD, EA, 1F]
[01, DA, 2B, 1D]
```

Round Key 9

```
[8B, 7A, 9C, 98]
[CE, 39, CE, DD]
[9A, 66, 85, 98]
[C3, 57, B7, AB]
```

XOR

```
[0A, F6, A4, 69]
[55, 69, 2D, 7D]
[EE, BB, 6F, 87]
[C2, 8D, 9C, B6]
```

- Inverse Mix Column

Column 0: [0A, 55, EE, C2]

Yellow values differ subject to column (The rest remain same)

The Inverse Mix Column operation for Column 0 is

```
(0E * 0A) XOR (0B * 55) XOR (0D * EE) XOR (09 * C2)
(09 * 0A) XOR (0E * 55) XOR (0B * EE) XOR (0D * C2)
(0D * 0A) XOR (09 * 55) XOR (0E * EE) XOR (0B * C2)
(0B * 0A) XOR (0D * 55) XOR (09 * EE) XOR (0E * C2)
```

Copy

Updated Column 0:

```
[BF]
[89]
[89]
[89]
```

4. ROUND 0

- Repeat Step 4 (EXCEPT INVERSE MIX COLUMN PART)

Final State Matrix After Decryption

Assume the final state matrix after decryption is:

```
[48, 45, 4C, 4C]  (Row 0)
[4F, 20, 57, 4F]  (Row 1)
[52, 4C, 44, 20]  (Row 2)
[20, 20, 20, 20]  (Row 3)
```

Final State Matrix After Inverse Mix Columns

After performing Inverse Mix Columns on all columns, the updated state matrix is:

```
[BF, 89, 89, 89]  (Row 0)
[89, 89, 89, 89]  (Row 1)
[89, 89, 89, 89]  (Row 2)
[89, 89, 89, 89]  (Row 3)
```

5. Convert Hexadecimal to ASCII to achieve final decrypted text

Hex: `48 45 4C 4C 4F 20 57 4F 52 4C 44 20 20 20 20 20`

ASCII: `H E L L O W O R L D` (5 spaces at the end)

Data Encryption Standard *(56 – bit)*

MAKING OF SUBKEYS

1. Make a 16-digit Hexadecimal key (convert to 64-bit binary key)

    1. **64-bit Key**: `133457799BBCDFF1` (hex)
        - Binary: `00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001`

2. Rearrange bits according to PC-1 table, and split in half (This step turns the 64-bit key to a 56-bit key)

Actually 56

PC-1 TABLE (ORDER OF BITS)						
57,	49,	41,	33,	25,	17,	9,
1,	58,	50,	42,	34,	26,	18,
10,	2,	59,	51,	43,	35,	27,
19,	11,	3,	60,	52,	44,	36,
63,	55,	47,	39,	31,	23,	15,
7,	62,	54,	46,	38,	30,	22,
14,	6,	61,	53,	45,	37,	29,
21,	13,	5,	28,	20,	12,	4

Apply PC-1:

- Use the PC-1 table to rearrange the bits.
- Result: Two 28-bit halves (C_0 and D_0):

Copy

```
C₀: 1111000 0110011 0010101 0101111
D₀: 0101010 1011001 1001111 0001111
```

3. Left Shift and PC-2 (repeat 16 times for 16 rounds)

- Perform either 1 or 2 left shifts on the previous 28-bit halves (Ck-1, Dk-1) according to squared data (So for left shift for Round 2, left shift will be applied on C1, D1)

: **Left Shifts**

For each round (i = 1 to 16), perform a **left shift** on $\mathbf{C_{k-1}}$ and $\mathbf{D_{k-1}}$ to generate $\mathbf{C_k}$ and $\mathbf{D_k}$. The number of bits to shift depends on the round:

- **Rounds 1, 2, 9, 16**: Shift **1 bit**.
- **Other rounds**: Shift **2 bits**.

Example for Round 1:

1. $\mathbf{C_0}$: `1111000 0110011 0010101 0101111`
2. $\mathbf{D_0}$: `0101010 1011001 1001111 0001111`

- Perform a **1-bit left shift** on both $\mathbf{C_0}$ and $\mathbf{D_0}$:
    - $\mathbf{C_1}$: `1110000 1100110 0101010 1011111`
    - $\mathbf{D_1}$: `1010101 0110011 0011110 0011111`

- Combine Ck and Dk

After shifting, combine $\mathbf{C_k}$ and $\mathbf{D_k}$ into a 56-bit block:

Copy

```
C₁: 1110000 1100110 0101010 1011111
D₁: 1010101 0110011 0011110 0011111
Combined: 1110000 1100110 0101010 1011111 1010101 0110011 0011110 0011111
```

- Rearrange According to PC-2 Table (This step shortens the 56-bit key to 48-bit subkey)

PC-2 TABLE (ORDER OF BITS)

```
14, 17, 11, 24,  1,  5,
 3, 28, 15,  6, 21, 10,
23, 19, 12,  4, 26,  8,
16,  7, 27, 20, 13,  2,
41, 52, 31, 37, 47, 55,
30, 40, 51, 45, 33, 48,
44, 49, 39, 56, 34, 53,
46, 42, 50, 36, 29, 32
```

Using the PC-2 table, rearrange the 56-bit block to create the 48-bit subkey **K_1**:

```
K₁: 000110 110000 001011 101111 001111 000011 110010 101101
```

ENCRYPTION

1. Convert Chosen 8-character Text to Binary

- The plaintext is a 64-bit block (e.g., `"HELLOWOR"` in ASCII).
- Convert it to binary if necessary:

Copy

```
Plaintext: 01001000 01000101 01001100 01001100 01001111 01010111 01001111 01010010
```

2. Rearrange bits according to IP table, and split in half

IP TABLE

```
58, 50, 42, 34, 26, 18, 10, 2,
60, 52, 44, 36, 28, 20, 12, 4,
62, 54, 46, 38, 30, 22, 14, 6,
64, 56, 48, 40, 32, 24, 16, 8,
57, 49, 41, 33, 25, 17, 9, 1,
59, 51, 43, 35, 27, 19, 11, 3,
61, 53, 45, 37, 29, 21, 13, 5,
63, 55, 47, 39, 31, 23, 15, 7
```

Result After IP

After applying the IP table, the permuted block is:

Copy

```
Permuted Block: 11111111 10100000 01111110 01110010 00000000 00000000 01011101 11110000
```

Split into L_0 and R_0

- **L_0 (Left Half)**: `11111111 10100000 01111110 01110010`
- **R_0 (Right Half)**: `00000000 00000000 01011101 11110000`

Step 3 and 4 are to be repeated 16 times until round 16

3. Find L1 (ROUND 1)

Step 1: Compute L_1

- **L_1** is simply **R_0** from the previous round.

Copy

```
L₁ = R₀ = 00000000 00000000 01011101 11110000
```

4. Find R1 (ROUND 1)

- Apply E Table

```
E TABLE (Expa
32, 1, 2, 3, 4, 5,
4, 5, 6, 7, 8, 9,
8, 9, 10, 11, 12, 13,
12, 13, 14, 15, 16, 17,
16, 17, 18, 19, 20, 21,
20, 21, 22, 23, 24, 25,
24, 25, 26, 27, 28, 29,
28, 29, 30, 31, 32, 1
```

Final expanded $\mathbf{R_0}$:

```
000000 000000 000000 000000 001011 111011 111110 000000
```

- XOR with respective Sub-Key

XOR Calculation

Copy

```
Expanded R₀: 000000 000000 000000 000000 001011 111011 111110 000000
Subkey K₁:   000110 110000 001011 101111 001111 000111 000001 110010
XOR Result:  000110 110000 001011 101111 000100 111100 111111 110010
```

- Apply appropriate S boxes

- Split this 48-bit result into **8 groups of 6 bits**:

```
Group 1: 000110
Group 2: 110000
Group 3: 001011
Group 4: 101111
Group 5: 000100
Group 6: 111100
Group 7: 111111
Group 8: 110010
```

- **Group 1** → S-Box 1
- **Group 2** → S-Box 2
- **Group 3** → S-Box 3
- ...
- **Group 8** → S-Box 8

1. The **first and last bits** determine the **row**:
   - `00` → Row 0
   - `01` → Row 1
   - `10` → Row 2
   - `11` → Row 3
2. The **middle 4 bits** determine the **column** (0 to 15 in decimal).

Group 1 → S-Box 1

Input: `000110`

- **Row:** `0 0` → **0** (Decimal)
- **Column:** `0011` → **3** (Decimal)
- S1(0,3) = **1** (Decimal) → `0001`

Group 2 → S-Box 2

Input: `110000`

- **Row:** `1 0` → **2** (Decimal)
- **Column:** `1000` → **8** (Decimal)
- S2(2,8) = **5** (Decimal) → `0101`

Group 3 → S-Box 3

Input: `001011`

- **Row:** `0 1` → **1** (Decimal)
- **Column:** `0101` → **5** (Decimal)
- S3(1,5) = **4** (Decimal) → `0100`

Group 4 → S-Box 4

Input: `101111`

- **Row:** `1 1` → **3** (Decimal)
- **Column:** `0111` → **7** (Decimal)
- S4(3,7) = **8** (Decimal) → `1000`

Group 5 → S-Box 5

Input: `000100`

- **Row:** `0 0` → **0** (Decimal)
- **Column:** `0010` → **2** (Decimal)
- S5(0,2) = **4** (Decimal) → `0100`

Group 6 → S-Box 6

Input: `111100`

- **Row:** `1 0` → **2** (Decimal)
- **Column:** `1110` → **14** (Decimal)
- S6(2,14) = **0** (Decimal) → `0000`

Group 7 → S-Box 7

Input: `111111`

- **Row:** `1 1` → **3** (Decimal)
- **Column:** `1111` → **15** (Decimal)
- S7(3,15) = **12** (Decimal) → `1100`

Group 8 → S-Box 8

Input: `110010`

- **Row:** `1 0` → **2** (Decimal)
- **Column:** `1001` → **9** (Decimal)
- S8(2,9) = **6** (Decimal) → `0110`

```
            DES S-BOX 1

        COLUMN →
        | 0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
--------+----------------------------------------------------------------
ROW  0  | 14  04  13  01  02  15  11  08  03  10  06  12  05  09  00  07
ROW  1  | 00  15  07  04  14  02  13  01  10  06  12  11  09  05  03  08
ROW  2  | 04  01  14  08  13  06  02  11  15  12  09  07  03  10  05  00
ROW  3  | 15  12  08  02  04  09  01  07  05  11  03  14  10  00  06  13
```

Only S-box 1 included for clarity

Final Result

```
0001 0101 0100 1000 0100 0000 1100 0110
```

- Apply P Table

P TABLE

```
16,  7, 20, 21,
29, 12, 28, 17,
 1, 15, 23, 26,
 5, 18, 31, 10,
 2,  8, 24, 14,
32, 27,  3,  9,
19, 13, 30,  6,
22, 11,  4, 25
```

The permutated output is:

```
00000000 00010111 01000000 01110011
```

- XOR With L0 to achieve R1

XOR Operation

Perform XOR bit by bit:

```
Copy
Permuted Result: 00000000 00010111 01000000 01110011
L₀:             11111111 10100000 01111110 01110010
XOR Result (R₁): 11111111 10110111 00111110 00000001
```

5. Combine L16 and R16

```
Copy
Final Block: R₁₆ + L₁₆
```

(Note: The order is $\mathbf{R_{16}}$ first, then $\mathbf{L_{16}}$.)

```
Copy
R₁₆: 11111111 10110111 00111110 00000001
L₁₆: 00000000 00000000 01011101 11110000
```

Combined final block:

```
Copy
11111111 10110111 00111110 00000001 00000000 00000000 01011101 11110000
```

- Apply IP^{-1} Table

IP^{-1} Table

```
40, 8, 48, 16, 56, 24, 64, 32,
39, 7, 47, 15, 55, 23, 63, 31,
38, 6, 46, 14, 54, 22, 62, 30,
37, 5, 45, 13, 53, 21, 61, 29,
36, 4, 44, 12, 52, 20, 60, 28,
35, 3, 43, 11, 51, 19, 59, 27,
34, 2, 42, 10, 50, 18, 58, 26,
33, 1, 41, 9, 49, 17, 57, 25
```

DECRYPTION

1. Convert cipher text to Binary

`59 54 5C 4C 5E 56 4A 52` →

`01011001 01010100 01011100 01001100 01011110 01010110 01001010 01010010`

2. Rearrange bits according to IP table, and split in half

IP TABLE							
58,	50,	42,	34,	26,	18,	10,	2,
60,	52,	44,	36,	28,	20,	12,	4,
62,	54,	46,	38,	30,	22,	14,	6,
64,	56,	48,	40,	32,	24,	16,	8,
57,	49,	41,	33,	25,	17,	9,	1,
59,	51,	43,	35,	27,	19,	11,	3,
61,	53,	45,	37,	29,	21,	13,	5,
63,	55,	47,	39,	31,	23,	15,	7

`L16 (Left 32 bits): 11001100000000001100110011110000`

`R16 (Right 32 bits): 10101010111100001010101000001111`

3. Compute Lk and Rk till L0 and R0 are achieved (Repeated 16 times)

ROUND 16

1. **Expand R16 (32 → 48 bits)** using **E-table**:

   `R16 = 10101010111100001010101000001111`

   → Expanded: `010101 010101 011110 000101 010101 010000 001111 110101`

2. **XOR with K16**:

   `K16 = 111000111000111000111000111000111000111000111000`

   → XOR Result: `101101 101101 100110 111101 101101 101000 110111 001101`

3. **S-Box Substitution (6-bit → 4-bit per S-box)**:
   - **First 6 bits (** `101101` **)** → S1 → Row `11` (3), Column `0110` (6) → `12` (`1100`)
   - Repeat for all 8 S-boxes → `1100 1010 0110 1101 1001 0101 1111 0010`

4. **P-Box Permutation (32-bit output)**:

   → `1010 1101 0100 1100 0011 0101 0110 1001`

5. **XOR with L16**:

   `L16 = 11001100000000001100110011110000`

   → `01100111 01001100 00000001 00001001` → **New R15**

6. **New L15 = R16**

   `L15 = 10101010111100001010101000001111`

(Repeat for all 16 rounds, using K15, K14, ..., K1.)

4. Combine L0 and R0

After Round 1 (using K1), recombine **L0** and **R0**:

`Final Block = L0 + R0 = 11001100000000001100110011110000 + 10101010111100001010101000001111`

5. Apply IP-1 Table to achieve decrypted text in hexadecimal

IP^{-1} Table

```
40, 8, 48, 16, 56, 24, 64, 32,
39, 7, 47, 15, 55, 23, 63, 31,
38, 6, 46, 14, 54, 22, 62, 30,
37, 5, 45, 13, 53, 21, 61, 29,
36, 4, 44, 12, 52, 20, 60, 28,
35, 3, 43, 11, 51, 19, 59, 27,
34, 2, 42, 10, 50, 18, 58, 26,
33, 1, 41, 9, 49, 17, 57, 25
```

```
48 45 4C 4C 4F 57 4F 52
```

6. Convert the hexadecimal to ASCII to achieve Final Decrypted Text

Final Result:

The hexadecimal `48 45 4C 4C 4F 57 4F 52` converts to:

`HELLOWOR`

Rivest–Shamir–Adleman *(9-bit)*

MAKING OF PUBLIC AND PRIVATE KEYS

1. Select two prime numbers, p and q

Step 1: Choose Two Prime Numbers

pick small prime numbers:

$$p = 17, \quad q = 19$$

2. Multiply p and q

Step 2: Compute n

$$n = p \times q = 17 \times 19 = 323$$

3. Multiply (p-1) with (q-1)

Step 3: Compute Euler's Totient Function $\phi(n)$

$$\phi(n) = (p-1) \times (q-1) = (17-1) \times (19-1) = 16 \times 18 = 288$$

4. Choose e (Public key), where e and ϕ(n) must not have a common factor (other than 1)

Step 4: Choose Public Key Exponent e

Greatest Common Devisor

- e should be **relatively prime** to $\phi(n)$, meaning **GCD(e, 288) = 1**.
- Let's pick $e = 5$ (a common choice).

5. Calculate d (Private key), by performing Modular Inverse

Step 5: Compute Private Key d

- d is the **modular inverse** of e modulo $\phi(n)$.

- We solve for d:

$$d = e^{-1} \mod 288$$

- Using the **Extended Euclidean Algorithm**, we find:

$$d = 173$$

6. Final Keys

✅ **Final Keys:**

- **Public Key** = $(e, n) = (5, 323)$
- **Private Key** = $(d, n) = (173, 323)$

ENCRYPTION

1. Convert text to ASCII

Letter	ASCII
H	72
E	69
L	76
L	76
O	79
SPACE	32
W	87
O	79
R	82
L	76
D	68

2. Encrypt each letters ASCII value (must be less than n, else make smaller blocks so that it is less than n), using formula and public key

The **RSA encryption formula** is:

$$c = m^e \mod n$$

We'll encrypt each **ASCII value** using:

- $e = 5$
- $n = 323$

🔹 **Encrypt each letter:**

$$c = m^5 \mod 323$$

1️⃣ **Encrypt** 72:

$$72^5 \mod 323 = 194$$

2️⃣ **Encrypt** 69: And rest of the letters...

$$69^5 \mod 323 = 69$$

3. Encrypted Text in Decimal

RSA-Encrypted: `194, 69, 87, 87, 248, 243, 17, 248, 152, 87, 171`

4. Convert to Hexadecimal for final Encrypted Text

🔹 **Final Hexadecimal Output:**

```nginx
C2 45 57 57 F8 F3 11 F8 98 57 AB
```

DECRYPTION

1. Reclaim decimal RSA encrypted text

RSA-Encrypted: `194, 69, 87, 87, 248, 243, 17, 248, 152, 87, 171`

2. Decrypt each number using decryption formula and private key

RSA decryption formula:

$$m = c^d \mod n$$

Where:

- c = ciphertext number
- $d = 173$ (private key)
- $n = 323$

We decrypt each number:

Decrypt Each Block

1️⃣ **Decrypt** 194:

$$194^{173} \mod 323 = 72$$

(H)

2️⃣ **Decrypt** 69:

$$69^{173} \mod 323 = 69$$

(E) And rest of the blocks...

3. Final Decrypted data

✅ **Final Decrypted Message:**

HELLO WORLD 🎉

Conclusion:

In conclusion, we need a class interface along with a context class (in this case **Cipher**) that will switch between the required strategies (cryptography algorithms) based on the user's input. And since each encryption and decryption follows a certain number of steps defined in the class interface, the following program includes both the Template and Strategy design pattern.