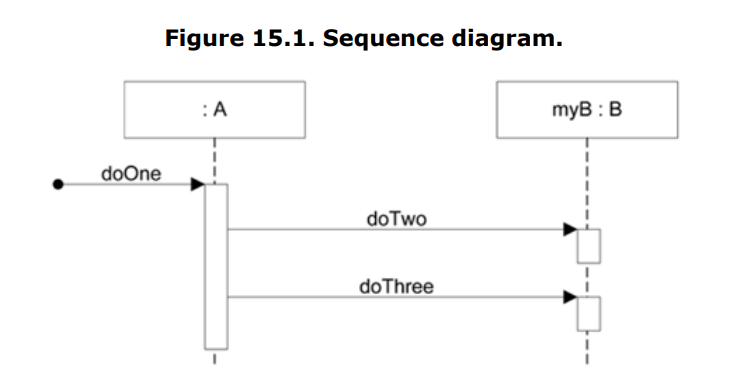
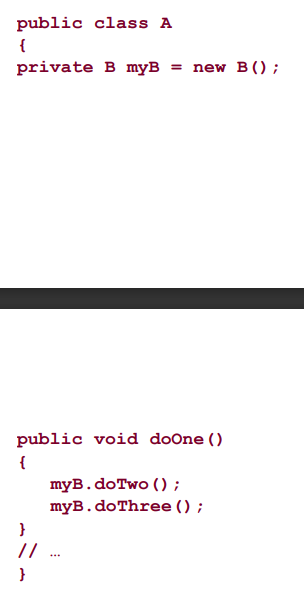
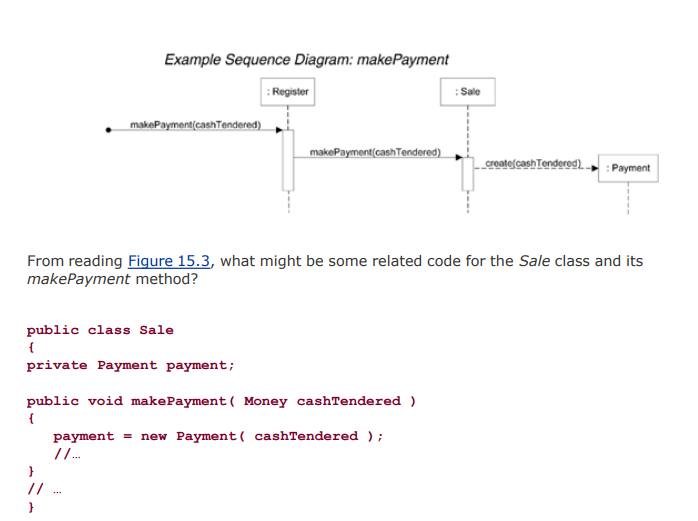
Sequence Diagram:

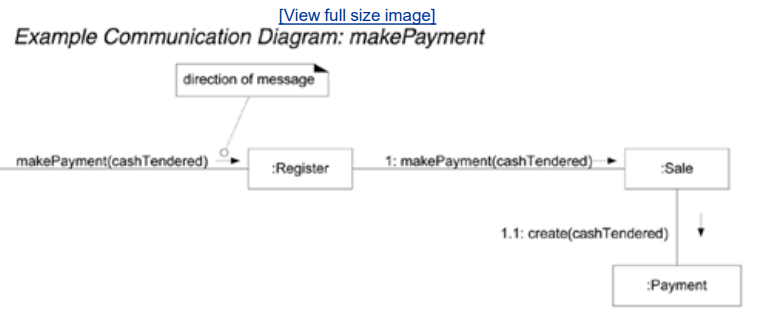


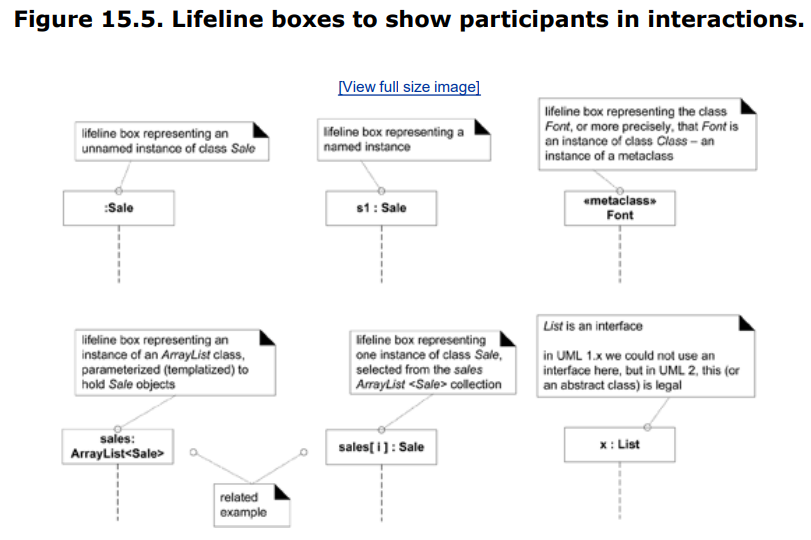


Sequence e.g.

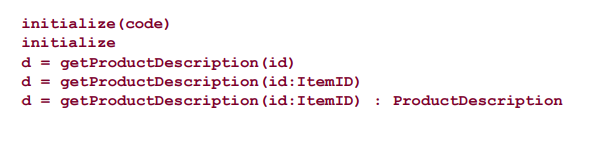


Create(cash Tendered) means an object of payment class created with constructor accepting cash tendered value. Pichla arrow shows agli class ka function but called in previous class. Isme register ki make payment ka function bana, usme sale ka obj banke sale ka makepayment call hua. Phir agay same.

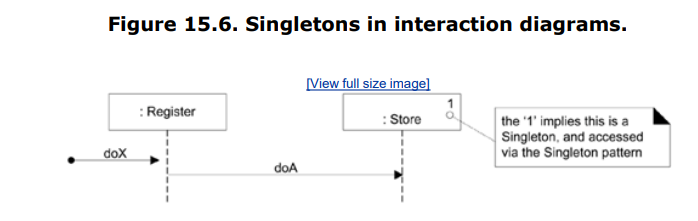


Notations for sequence:  


\* ArrayList and sales[i]



Extra info: getproddesc called, returns proddesc.

Singleton in Sequence diagram:  


Code:

public class MySingleton {

private static MySingleton *instance*;

private MySingleton() {

}

public static MySingleton getInstance() {

if (*instance* == null) {

*instance* = new MySingleton();

}

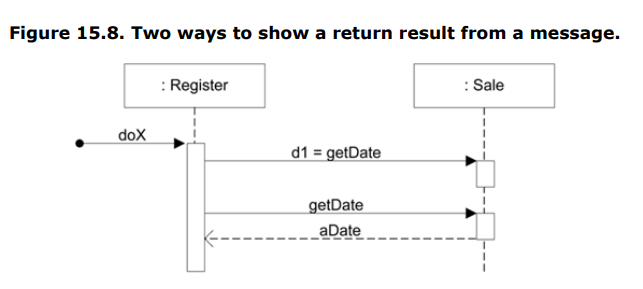
return *instance*;

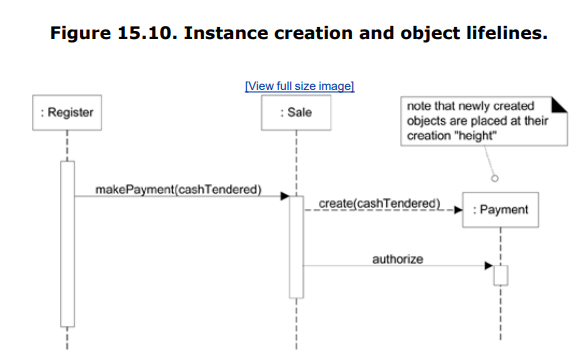
}

}

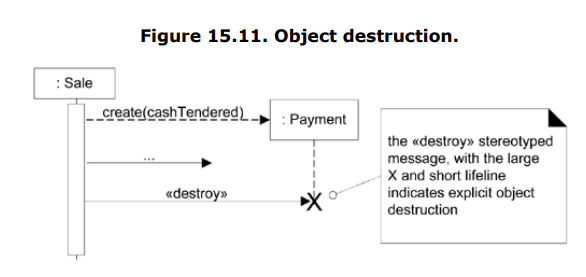
}

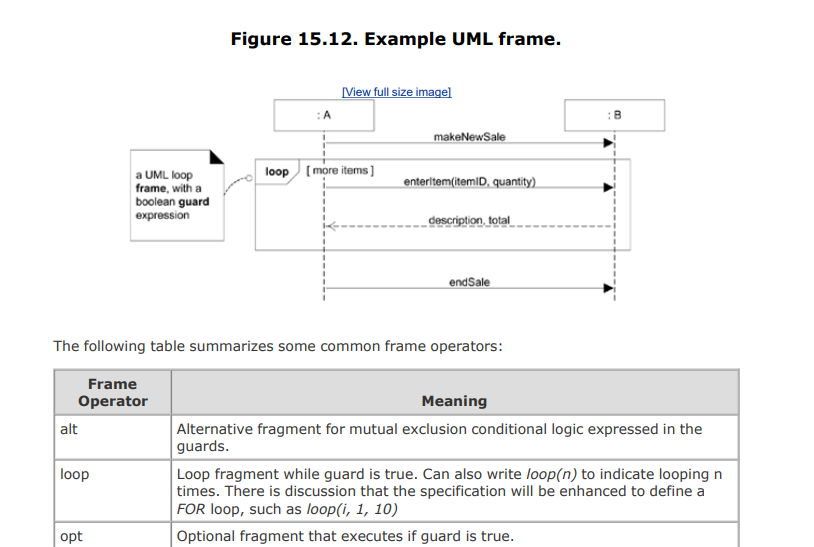
Return type:



Object creation:  


If new object created, it will be at same height of its creation time.

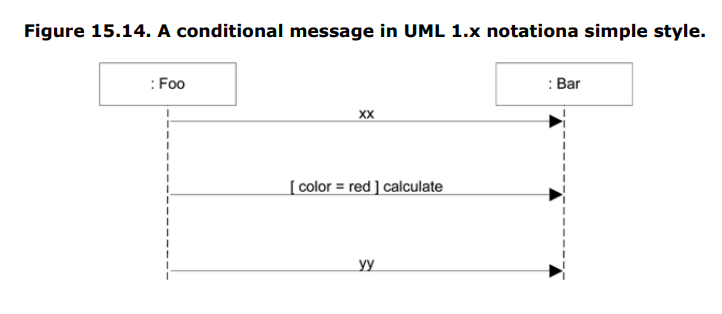




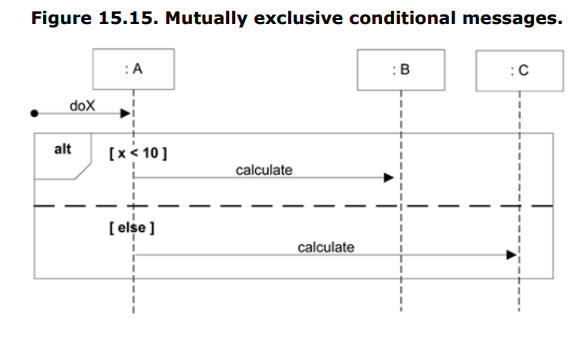
Alt= if,elseif,else

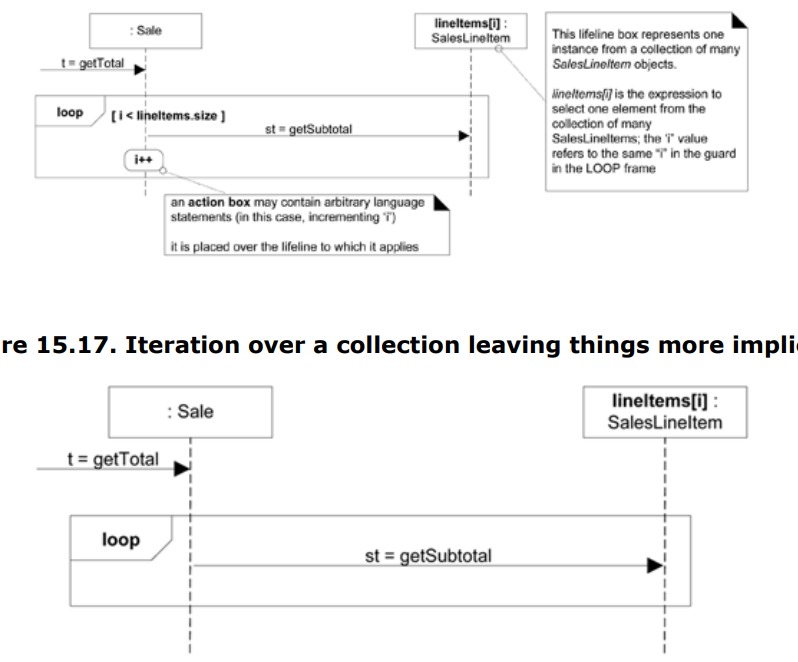
Loop

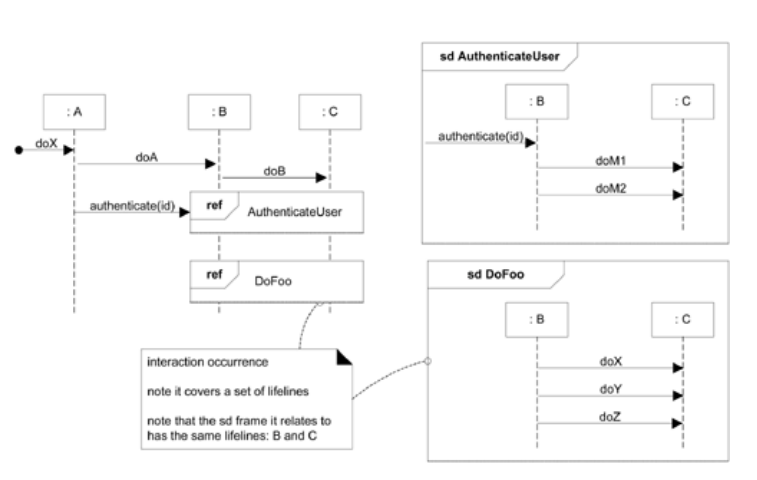
Opt= only if.

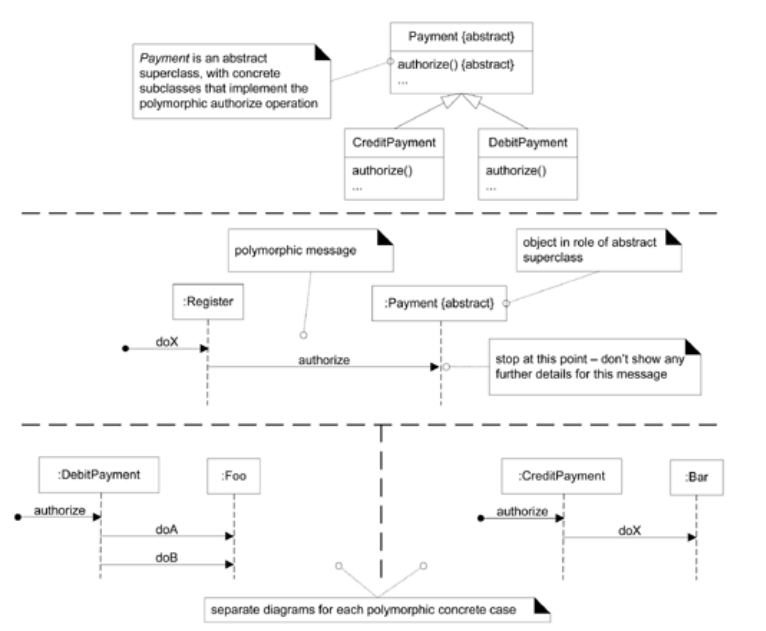


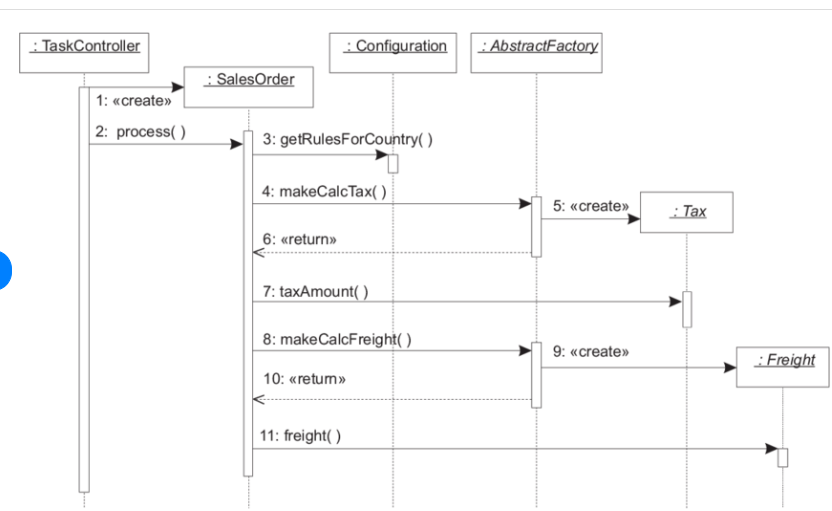
Above is another way to show opt.



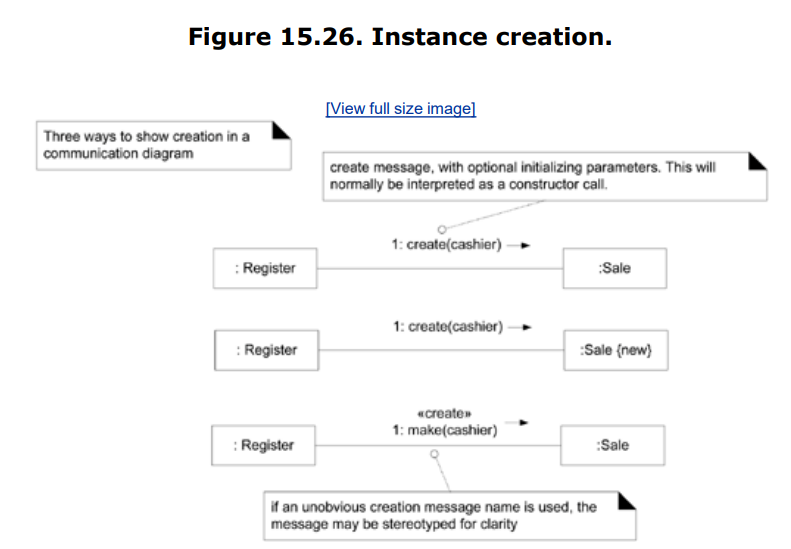
Implicit and explicit looping:  


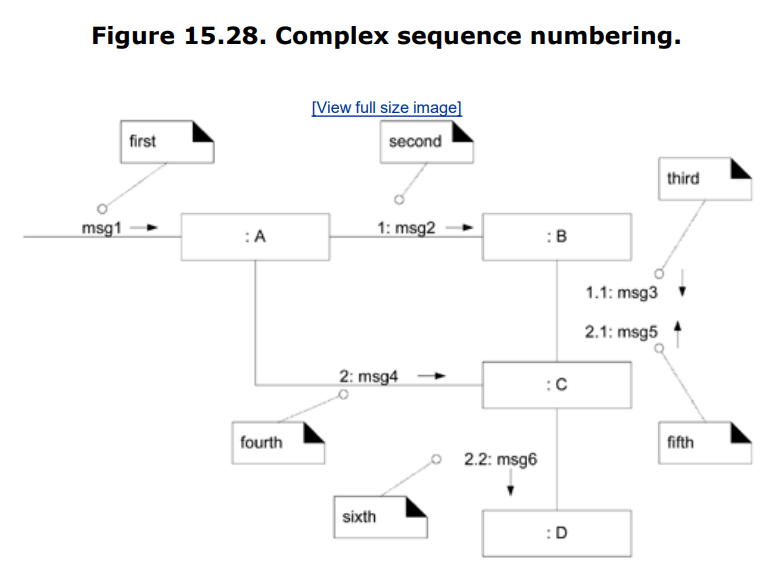
Dividing code into smaller parts:  


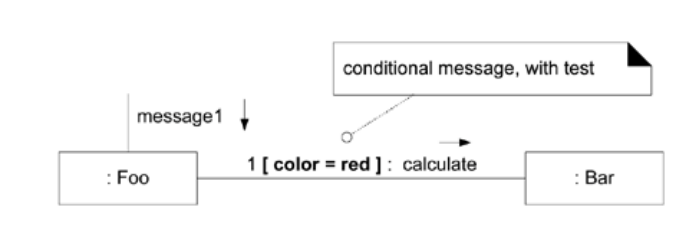
Polymorphism:  


Abstract factory:  


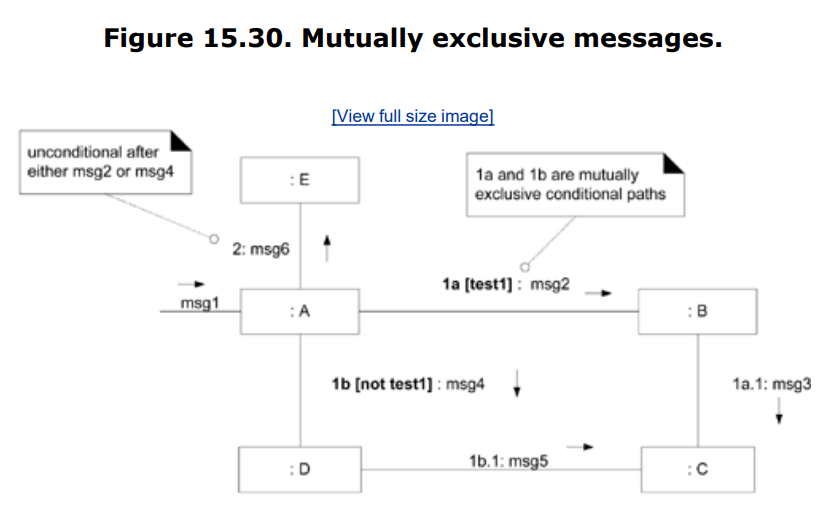
**Communication Diagram**

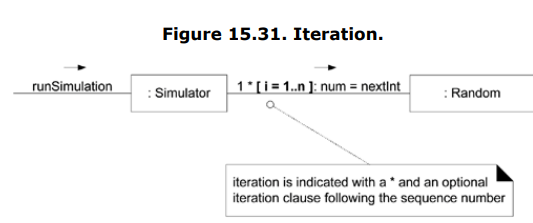




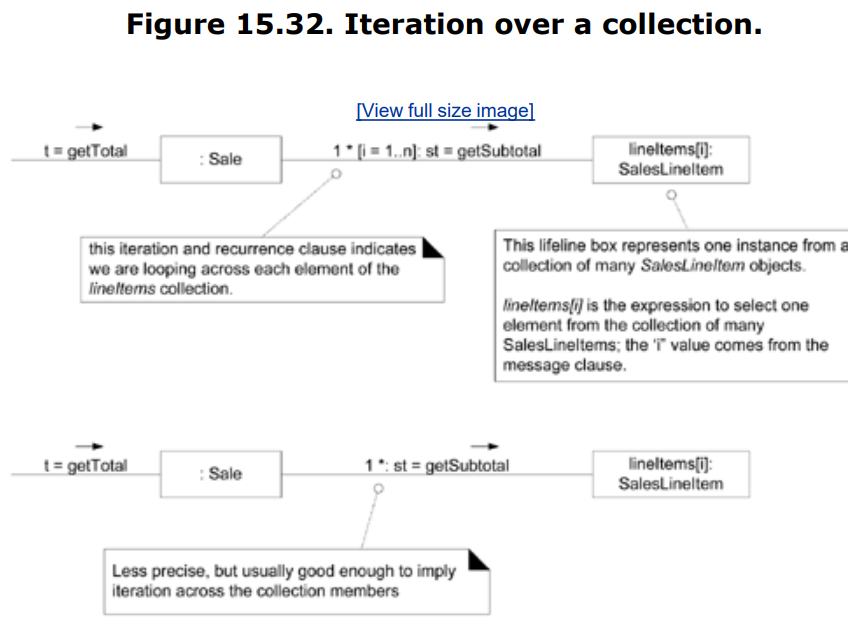


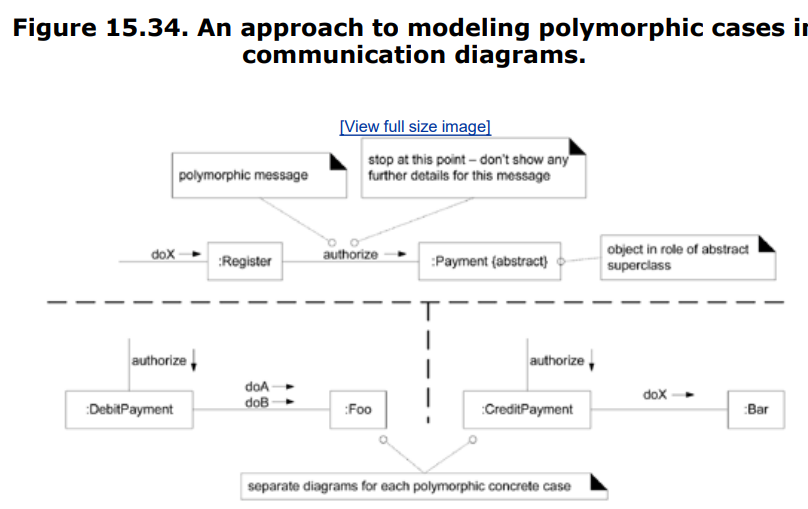
If color == red



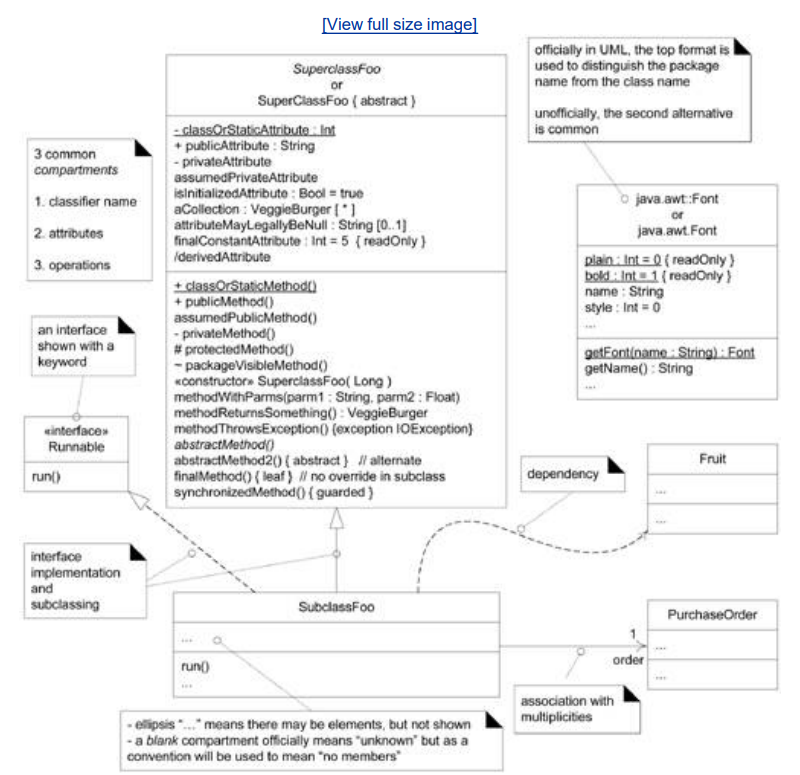
Loops in diagrams:  


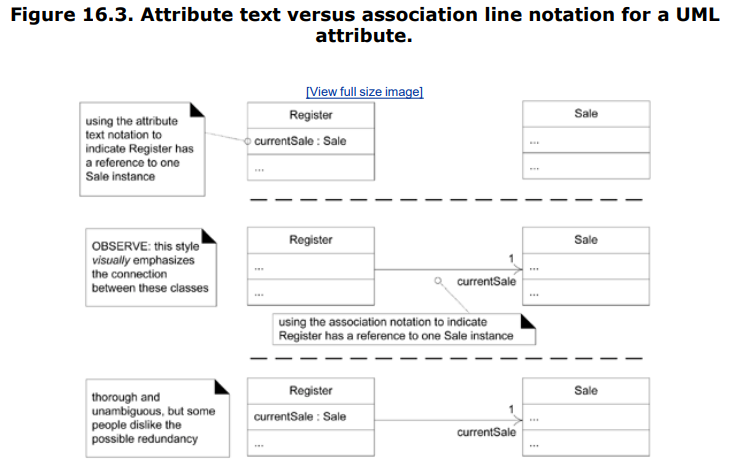
Iterations over a collection:

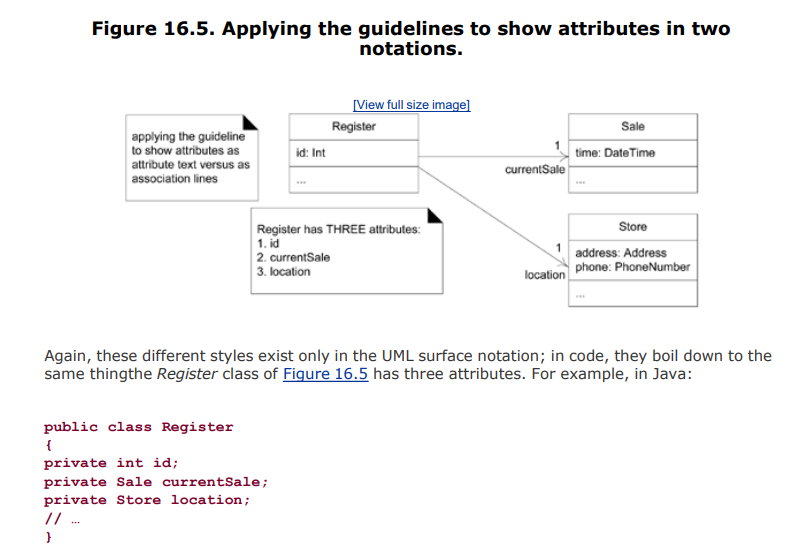


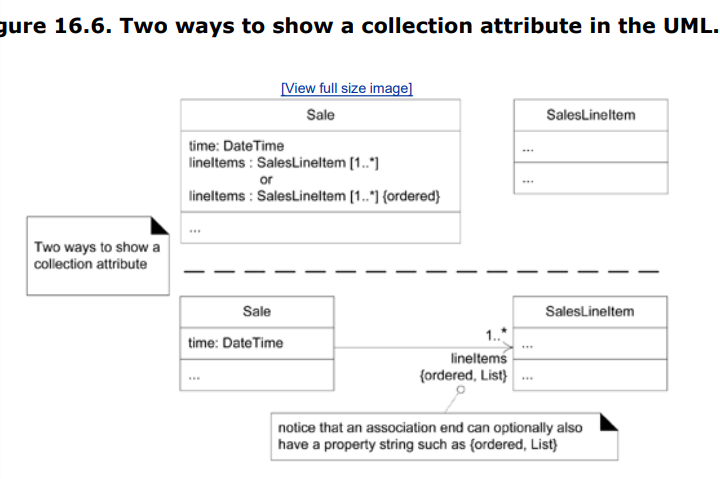
Polymorphism:  


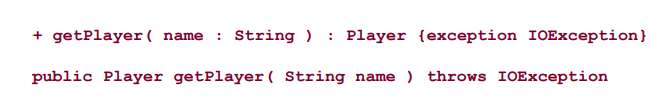
**Class diagram**

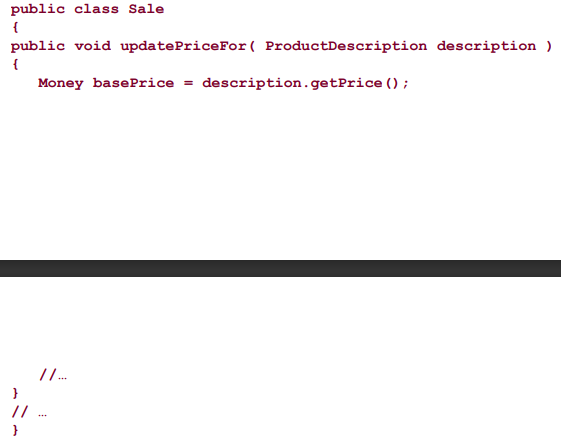


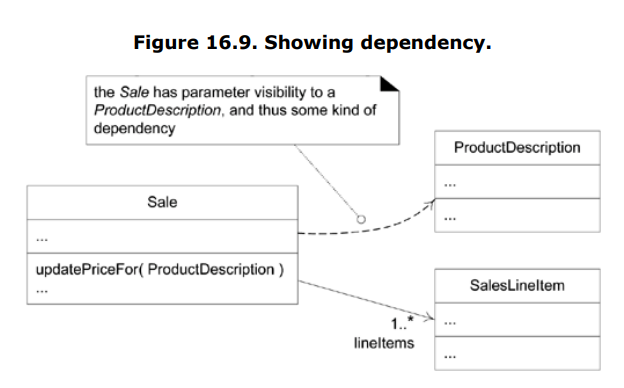


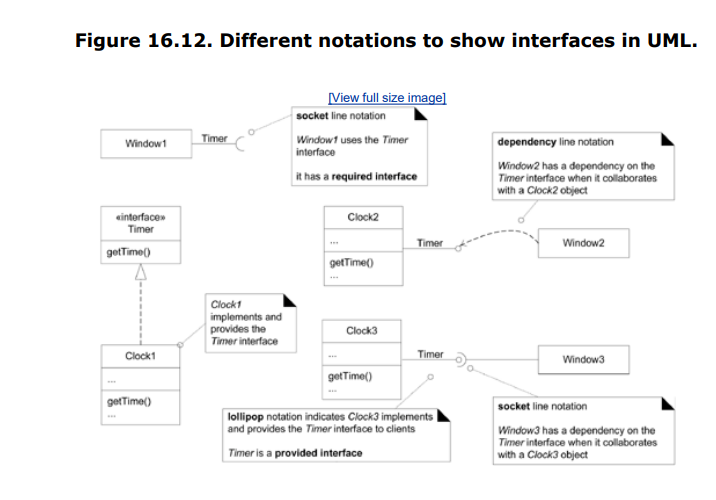
Different ways of showing creation of object:  


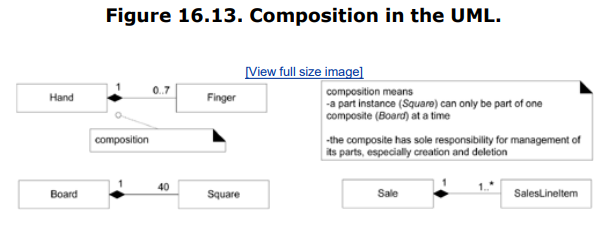
Showing array in class diagram:  


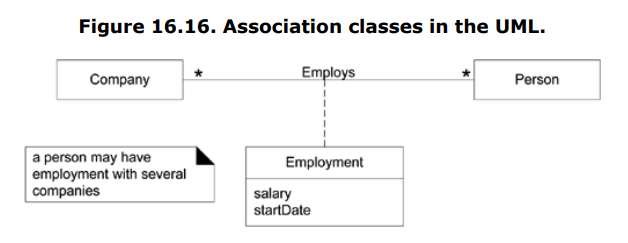


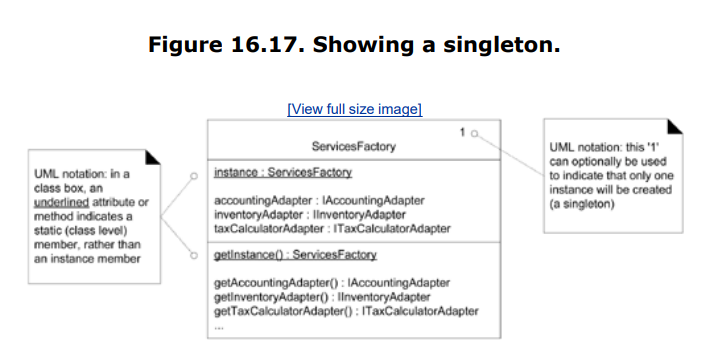
Showing dependency:  




Interfaces:  








State design pattern/diagram:

public class EmailValidator {

enum *State* {

***START***,

***ALPHANUM\_BEFORE\_AT***,

***AT***,

***ALPHANUM\_AFTER\_AT***,

***DOT***,

***ALPHABET\_AFTER\_DOT***,

***INVALID***

}

public static boolean isValidEmail(String email) {

*State* state = *State*.***START***;

for (char character : email.toCharArray()) {

switch (state) {

case ***START***:

if (Character.*isLetter*(character)) {

state = *State*.***ALPHANUM\_BEFORE\_AT***;

} else {

state = *State*.***INVALID***;

}

break;

case ***ALPHANUM\_BEFORE\_AT***:

if (Character.*isLetterOrDigit*(character)) {

// Continue in the same state

} else if (character == '@') {

state = *State*.***AT***;

} else {

state = *State*.***INVALID***;

}

break;

case ***AT***:

if (Character.*isLetterOrDigit*(character)) {

state = *State*.***ALPHANUM\_AFTER\_AT***;

} else {

state = *State*.***INVALID***;

}

break;

case ***ALPHANUM\_AFTER\_AT***:

if (Character.*isLetterOrDigit*(character)) {

// Continue in the same state

} else if (character == '.') {

state = *State*.***DOT***;

} else {

state = *State*.***INVALID***;

}

break;

case ***DOT***:

if (Character.*isLetter*(character)) {

state = *State*.***ALPHABET\_AFTER\_DOT***;

} else {

state = *State*.***INVALID***;

}

break;

case ***ALPHABET\_AFTER\_DOT***:

if (Character.*isLetter*(character)) {

// Continue in the same state

} else {

state = *State*.***INVALID***;

}

break;

case ***INVALID***:

// Stay in the INVALID state

break;

}

}

return state == *State*.***ALPHABET\_AFTER\_DOT***;

}

public static void main(String[] args) {

String validEmail = "example@example.com";

String invalidEmail = "invalid.email";

System.***out***.println("Is Valid Email: " + *isValidEmail*(validEmail)); // Should print true

System.***out***.println("Is Valid Email: " + *isValidEmail*(invalidEmail)); // Should print false

}

}

Component diagram:

CODES ON ECLIPSE.

REFACTORING AND CODE SMELLS.

**Long method (Function too long)**

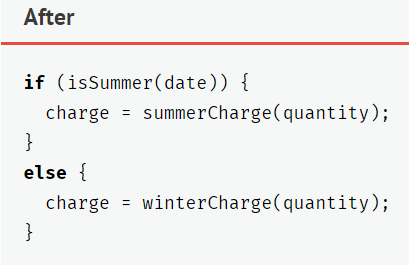
Extract method e.g:

TRY EXTRACTING LOOPS

TO NEW FUNCTIONS.

AGAR IF ELSE HO TOU WO CONDITION

AIK FUNCTION ME RKH KR RETURN

MARLO E.G.

**Primitive obsession:** int to INTEGER.

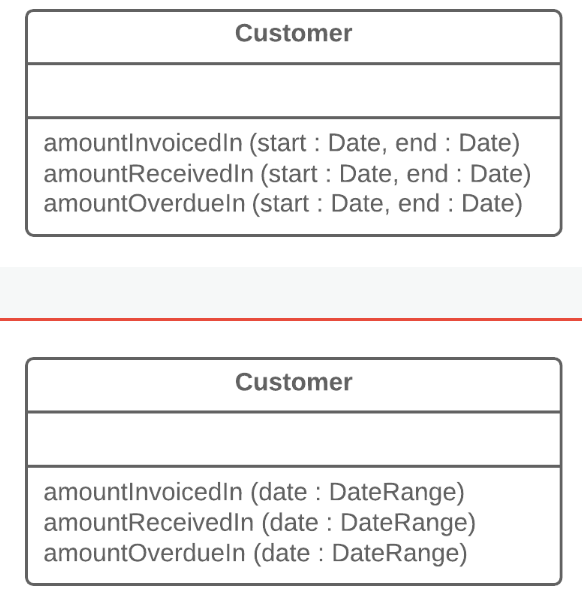
**Data clumps:** Extract class. When variables are used again and again e.g. db connect variables, make a class for them. E.g. jdbc singleton.

**Large class:** Extract class. Classes choti krni.

**Long parameter list:**

****

Parameter obj:



**REFACTORING**

1. Change Function Declaration (Rename Function):

- Problem: The name of a function doesn't accurately reflect what it does.

- Solution: Rename the function to better describe its purpose.

2. Combine Functions Into Class:

- Problem: Related functions are scattered and not organized.

- Solution: Group related functions into a class to improve cohesion.

3. Consolidate Conditional Expressions:

- Problem: Multiple conditionals that can be combined.

- Solution: Combine the conditions into a single conditional.

4. Decompose Conditional:

- Problem: A complex conditional that can be broken into simpler parts.

- Solution: Break down the conditional into separate functions or methods.

5. Encapsulate Variable:

- Problem: Direct access to a class field from outside the class.

- Solution: Make the field private and provide access through getter and setter methods.

6. Extract Function:

- Problem: A block of code within a function can be extracted for better readability.

- Solution: Extract the block of code into a new function.

7. Extract Variable:

- Problem: Complex expressions that can be clarified by introducing a variable.

- Solution: Introduce a variable to store the result of the complex expression.

8. Introduce Assertion:

- Problem: Lack of explicit checks for expected conditions.

- Solution: Introduce assertions to explicitly state expected conditions.

9. Introduce Null Objects:

- Problem: Excessive null checks in the code.

- Solution: Introduce a null object to represent the absence of an object.

10. Introduce Parameter Object:

- Problem: Too many parameters in a method.

- Solution: Group related parameters into a single object.

11. Move Field:

- Problem: A field is better suited in another class.

- Solution: Move the field to the appropriate class.

12. Move Function:

- Problem: A function is better suited in another class.

- Solution: Move the function to the appropriate class.

13. Remove Dead Code:

- Problem: Unused or redundant code.

- Solution: Delete the unnecessary code.

14. Rename Field:

- Problem: The name of a class field doesn't accurately reflect its purpose.

- Solution: Rename the field to better describe its purpose.

15. Rename Variable:

- Problem: The name of a variable doesn't accurately reflect its purpose.

- Solution: Rename the variable to better describe its purpose.

16. Replace Conditional With Polymorphism:

- Problem: Excessive use of conditional statements.

- Solution: Replace conditionals with polymorphic behavior.

17. Replace Control Flag With Break:

- Problem: Use of control flags to exit loops.

- Solution: Replace control flags with a `break` statement.

18. Replace Magical Literal:

- Problem: Use of hard-coded constants without explanation.

- Solution: Replace magic numbers with named constants.

19. Replace Nested Conditional With Guard Clause:

- Problem: Deeply nested conditionals.

- Solution: Use guard clauses to handle special cases at the beginning of a function.

20. Split Loop:

- Problem: A loop that is doing multiple things.

- Solution: Split the loop into multiple loops, each handling a specific task.

21. Split Variable:

- Problem: A variable that is used for multiple purposes.

- Solution: Split the variable into multiple variables, each serving a specific purpose.

EXAMPLE CODE MESSY:  
public class MessyCode {

private int a;

private int b;

private int c;

public MessyCode(int a, int b, int c) {

this.a = a;

this.b = b;

this.c = c;

}

public void doSomething() {

if (a > 0 && b > 0) {

if (c == 10) {

System.out.println("Condition met!");

process();

}

}

int result = a + b - c \* 2;

System.out.println("Result: " + result);

if (result > 0) {

System.out.println("Result is positive!");

}

// ... more messy code ...

int value = 42;

if (value == 42) {

System.out.println("The meaning of life!");

}

}

private void process() {

// ... process something ...

}

}  
  
EXAMPLE CODE FIXED:  
public class CleanCode {

private int firstNumber;

private int secondNumber;

private int thirdNumber;

public CleanCode(int firstNumber, int secondNumber, int thirdNumber) {

this.firstNumber = firstNumber;

this.secondNumber = secondNumber;

this.thirdNumber = thirdNumber;

}

public void performOperation() {

if (areNumbersPositive()) {

if (isThirdNumberEqualTo(10)) {

System.out.println("Condition met!");

processOperation();

}

}

int result = calculateResult();

System.out.println("Result: " + result);

printPositiveResult(result);

// ... more clean code ...

printMeaningOfLife();

}

private boolean areNumbersPositive() {

return firstNumber > 0 && secondNumber > 0;

}

private boolean isThirdNumberEqualTo(int value) {

return thirdNumber == value;

}

private int calculateResult() {

return firstNumber + secondNumber - thirdNumber \* 2;

}

private void printPositiveResult(int result) {

if (result > 0) {

System.out.println("Result is positive!");

}

}

private void processOperation() {

// ... process something ...

}

private void printMeaningOfLife() {

int meaning = 42;

if (meaning == 42) {

System.out.println("The meaning of life!");

}

}

}

**CODE SMELLS**

Conditional Complexity (Switch Statement):

Description: Excessive use of switch statements or deeply nested if-else conditions.

Refactoring: Replace switch statements with polymorphism (Replace Conditional With Polymorphism) (Jitne cases utni classes bana kr unme working kro sari aur aik interface se sari implement hui hon.)

1. **Data Clump:**
   * **Description:** Multiple variables that often appear together in various parts of the code.
   * **Refactoring:** Group related variables into a class (Extract Class).
2. **Dead Code:**
   * **Description:** Code that is never executed.
   * **Refactoring:** Remove the unused code (Remove Dead Code).
3. **Divergent Change:**
   * **Description:** A class is frequently modified for different reasons.
   * **Refactoring:** Separate different concerns into distinct classes (Extract Class, Move Function).
4. **Duplicated Code:**
   * **Description:** Repeated code in different parts of the application.
   * **Refactoring:** Extract duplicated code into methods or functions (Extract Method).
5. **Fallacious Comment:**
   * **Description:** Comments that are no longer accurate or misleading.
   * **Refactoring:** Update or remove incorrect comments (Remove Comment).
6. **Feature Envy:**
   * **Description:** A class is more interested in the data of another class than its own.
   * **Refactoring:** Move methods to the class that holds the data (Move Method).
7. **Flag Argument:**
   * **Description:** A boolean parameter that indicates different behavior.
   * **Refactoring:** Split the function into separate functions with clear purposes (Split Function).
8. **Global Data:**
   * **Description:** Excessive use of global variables.
   * **Refactoring:** Limit the scope of variables by encapsulating them (Encapsulate Variable).
9. **Insider Trading (Inappropriate Intimacy):**

* **Description:** Excessive reliance on another class's internal details.
* **Refactoring:** Move methods to the appropriate class (Move Method).

1. **Large Class:**

* **Description:** A class that performs too many responsibilities.
* **Refactoring:** Extract smaller, more focused classes (Extract Class).

1. **Lazy Element/Class:**

* **Description:** Elements or classes that are not used.
* **Refactoring:** Remove the unused elements or classes (Remove Dead Code).

1. **Long Method:**

* **Description:** A method that is excessively long.
* **Refactoring:** Break down the method into smaller, more manageable methods (Extract Method).

1. **Long Parameter List:**

* **Description:** A method with a large number of parameters.
* **Refactoring:** Group related parameters into a single object (Introduce Parameter Object).

1. **Magic Number:**

* **Description:** Use of hard-coded constants without explanation.
* **Refactoring:** Replace magic numbers with named constants (Replace Magic Literal).

1. **Null Check:**

* **Description:** Frequent checks for null values.
* **Refactoring:** Introduce Null Objects or use Optional (Introduce Null Objects).

1. **Parallel Inheritance Hierarchies:**

* **Description:** Corresponding class hierarchies that evolve independently.
* **Refactoring:** Combine hierarchies or use interfaces to decouple them (Move Method, Extract Interface).

1. **Primitive Obsession:**

* **Description:** Excessive use of primitive data types instead of custom objects.
* **Refactoring:** Replace primitives with custom objects (Replace Data Value with Object).

1. **Refused Bequest:**

* **Description:** Subclasses don't use the methods or data inherited from a superclass.
* **Refactoring:** Adjust the inheritance hierarchy or remove unused methods (Push Down Method, Remove Method).

1. **Shotgun Surgery:**

* **Description:** A single change requires modifications in many classes.
* **Refactoring:** Consolidate related functionality into a single class (Move Method).

1. **Speculative Generality:**

* **Description:** Overly complex designs or structures that aren't currently needed.
* **Refactoring:** Remove unused abstractions or simplify the design (Remove Class, Remove Method).

1. **Temporary Field:**

* **Description:** A field that is set but never used.
* **Refactoring:** Remove the unused field (Remove Unused Field).

1. **What Comment:**

* **Description:** Comments that don't provide useful information.
* **Refactoring:** Remove unnecessary comments (Remove Comment).