THE UNIVERSITY OF NEWCASTLE AUSTRALIA

FACULTY OF ENGINEERING AND BUILT ENVIRONMENT

www.newcastle.edu.au

SEQ2130 – Week 1 Introduction

Dr. Joe Ryan

SENG2130 – Systems Analysis and Design

University of Newcastle

UML Diagram

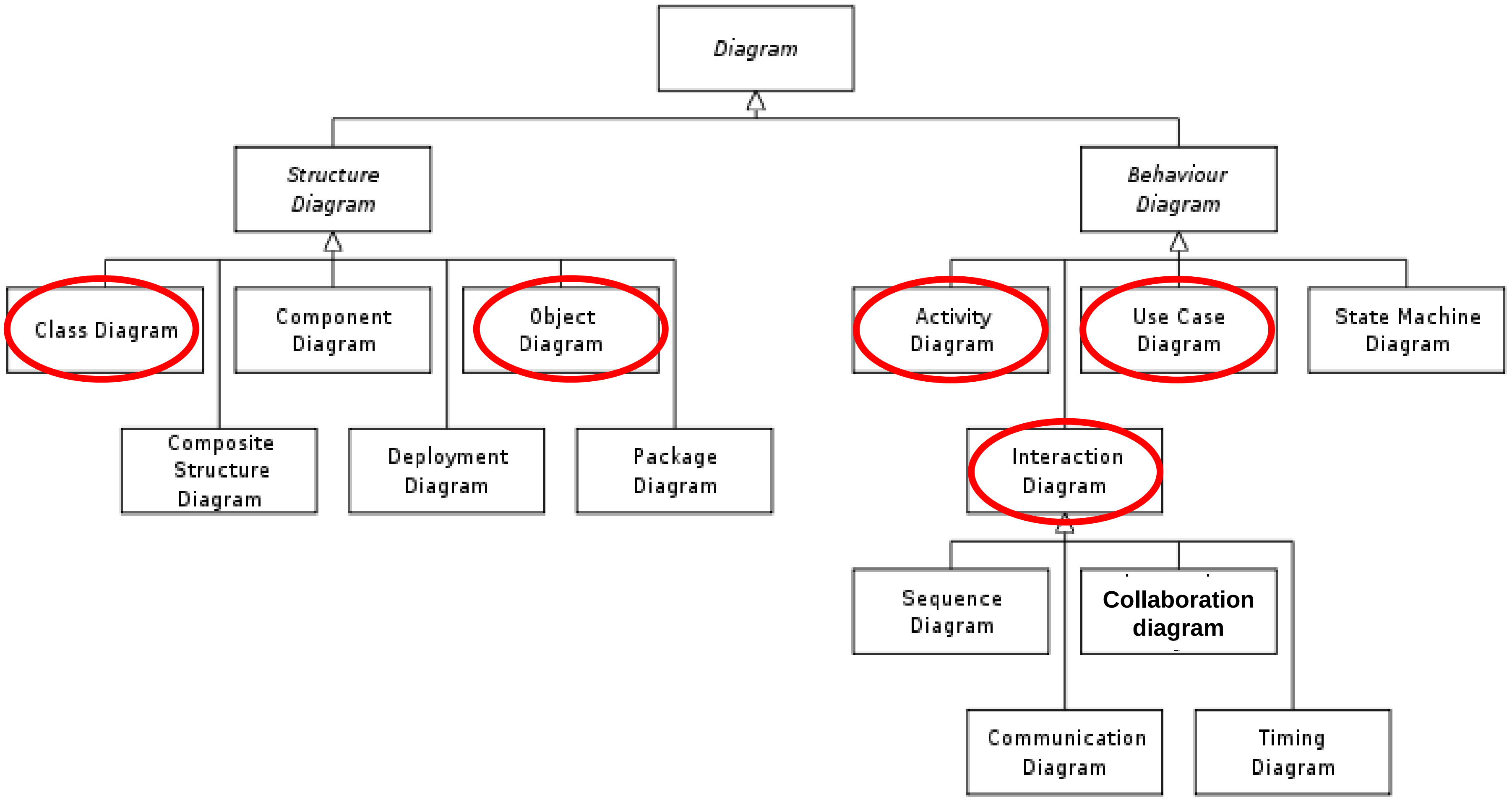


Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Collaboration diagram
Communication Diagram
Timing Diagram

This Week

1. Sequence Diagrams
2. Advanced modelling

Sequence diagrams

- Sequence diagram used to model the dynamic aspects of systems.

- A sequence diagram is an interaction diagram that emphasizes the **time ordering** of the **messages.**

- An object interacts with another object by sending **messages.**

Sequence Diagrams

Sequence Diagram

Overview

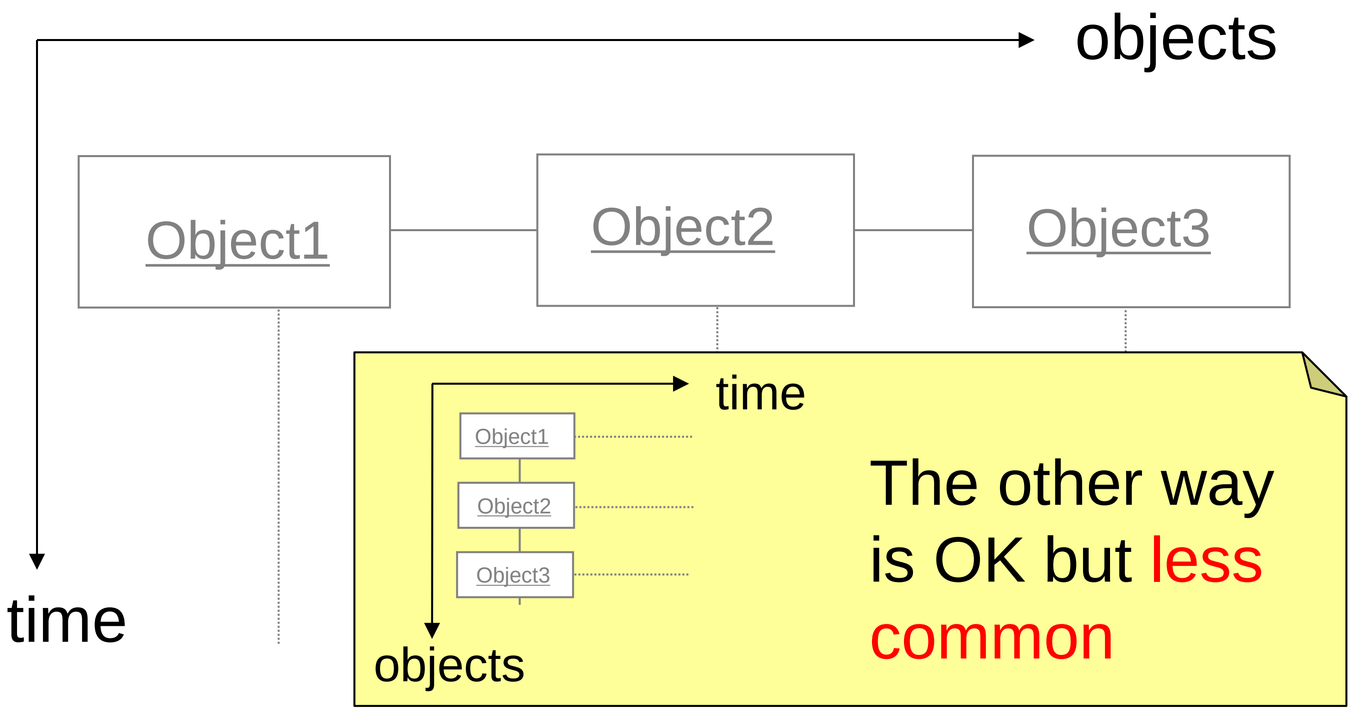
Sequence diagrams document the interactions between objects.

Sequence Diagrams

Sequence Diagram

Overview

The Sequence diagram lists objects horizontally, and time vertically, and models these messages over time.



The other way is OK but less common

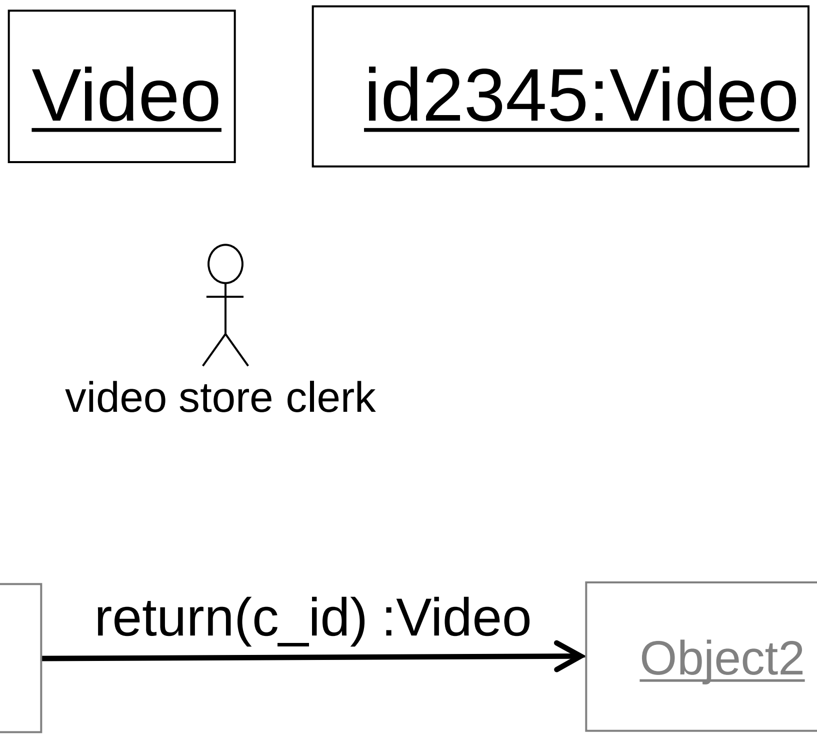
Sequence Diagrams

Sequence Diagram

Notation

- Object
- Actor
- Messages
- Lifeline
- Activation

Object, Actors & Messages



Sequence Diagrams

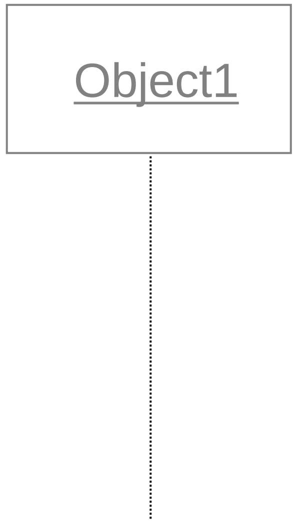
Sequence Diagram

Notation

- Object
- Actor
- Messages
- Lifeline
- Activation

Lifeline

Identifies the existence of the object over time. The notation is a vertical dotted line extending from an object.



Sequence Diagrams

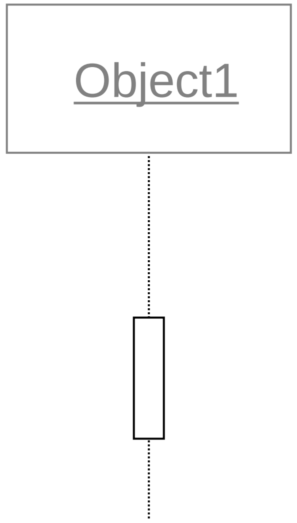
Sequence Diagram

Notation

- Object
- Actor
- Messages
- Lifeline
- Activation

Activation

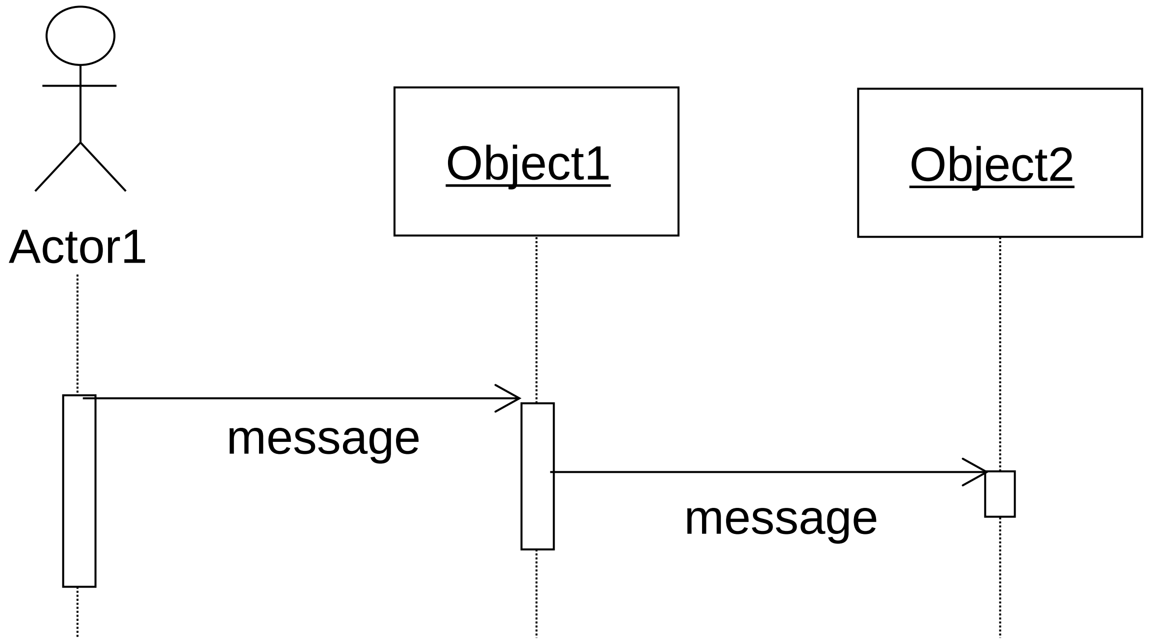
Indicates when the object is performing an action. Modelled as rectangular boxes on the lifeline



Sequence Diagrams

Sequence Diagram

Example



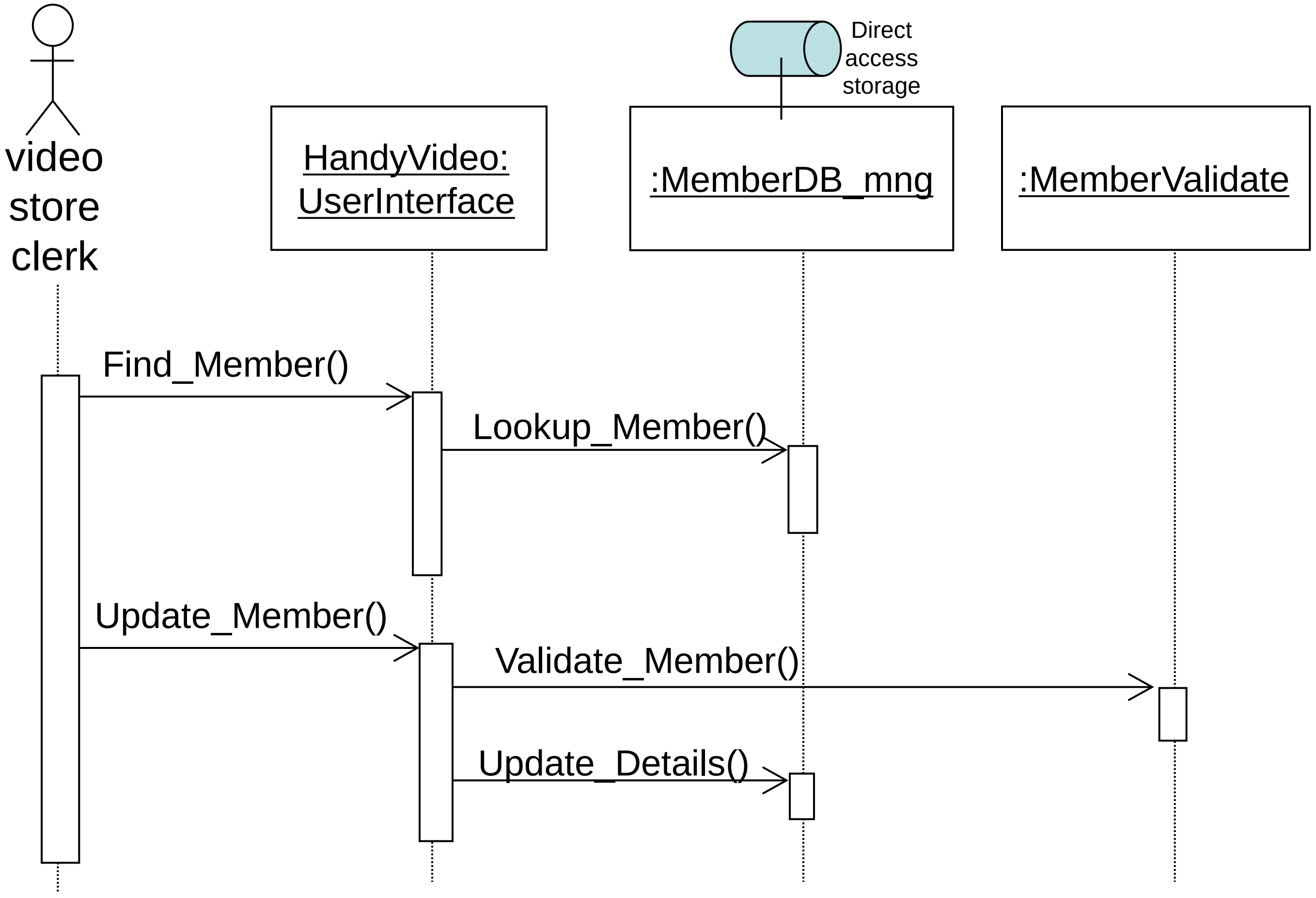
Sequence Diagrams

Sequence Diagram

Example

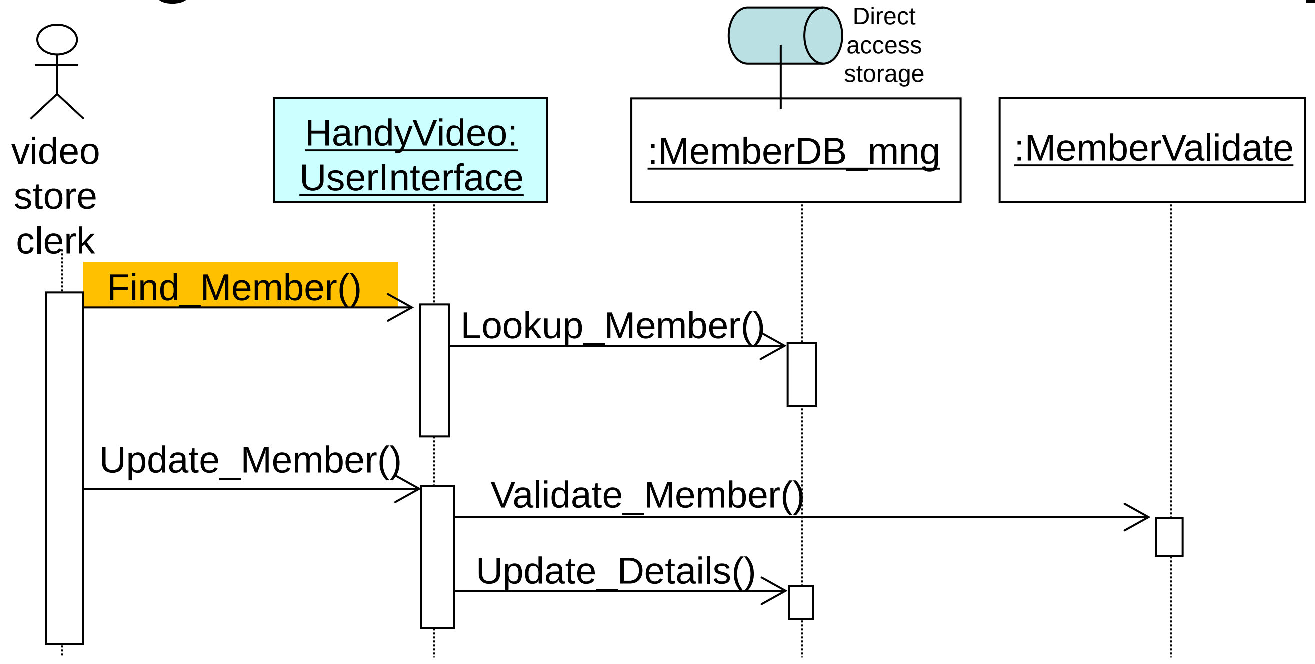
1.0 Find_Member
  1.1 Lookup_Member

2.0 Update_Member
  2.1 Validate_Member
  2.2 Update_Details



Sequence Diagrams

Sequence Diagram



Messages

1.0 Find_Member
- 1.1 Lookup_Member

2.0 Update_Member
- 2.1 Validate_Member
- 2.2 Update_Details

Classes

UserInterface	MemberDB_mng	MemberValidate

Find_Member()
Update_Member()
	Lookup_Member()
	Update_Details()
		Validate_Member()

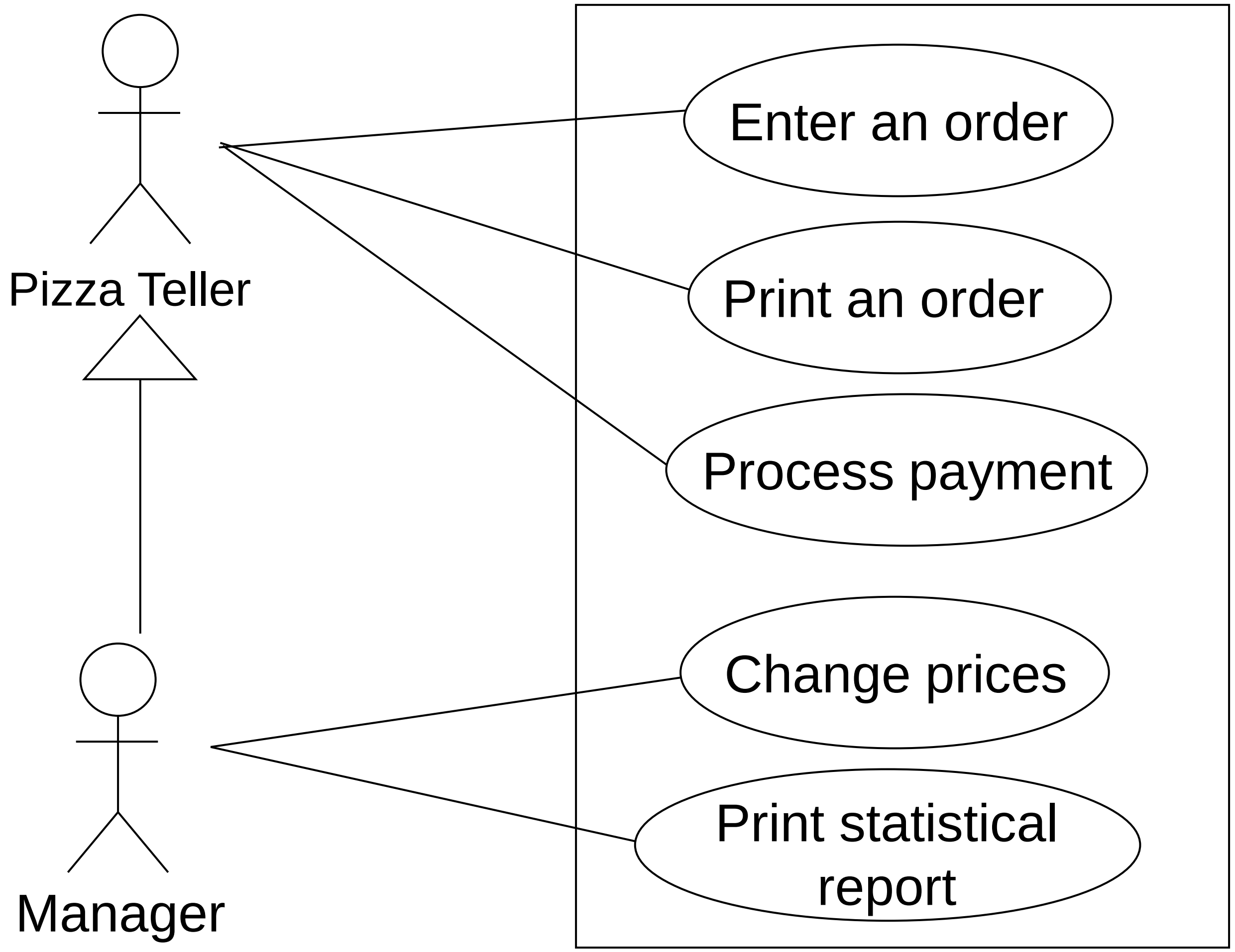
THE UNIVERSITY OF NEWCASTLE AUSTRALIA

The Pizza Shop

- Consider a pizza shop. This store caters for many activities, the major one being “pizza eating”. The choice of pizzas is simple: Meat Lovers, Vegetarian or Supreme. The pizzas come in large, medium and small. The base is thick or thin. Customers come to the desk and talk to the Pizza Tellers to make a selection and the order is taken. This includes the customer's name so they can be identified with the order. The customer goes to a table and waits for the order to be made up. The customer is called back to the desk (by name) when the order is ready. The customer pays the bill. Every so often the manager of the store changes the prices for each type of pizza.

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

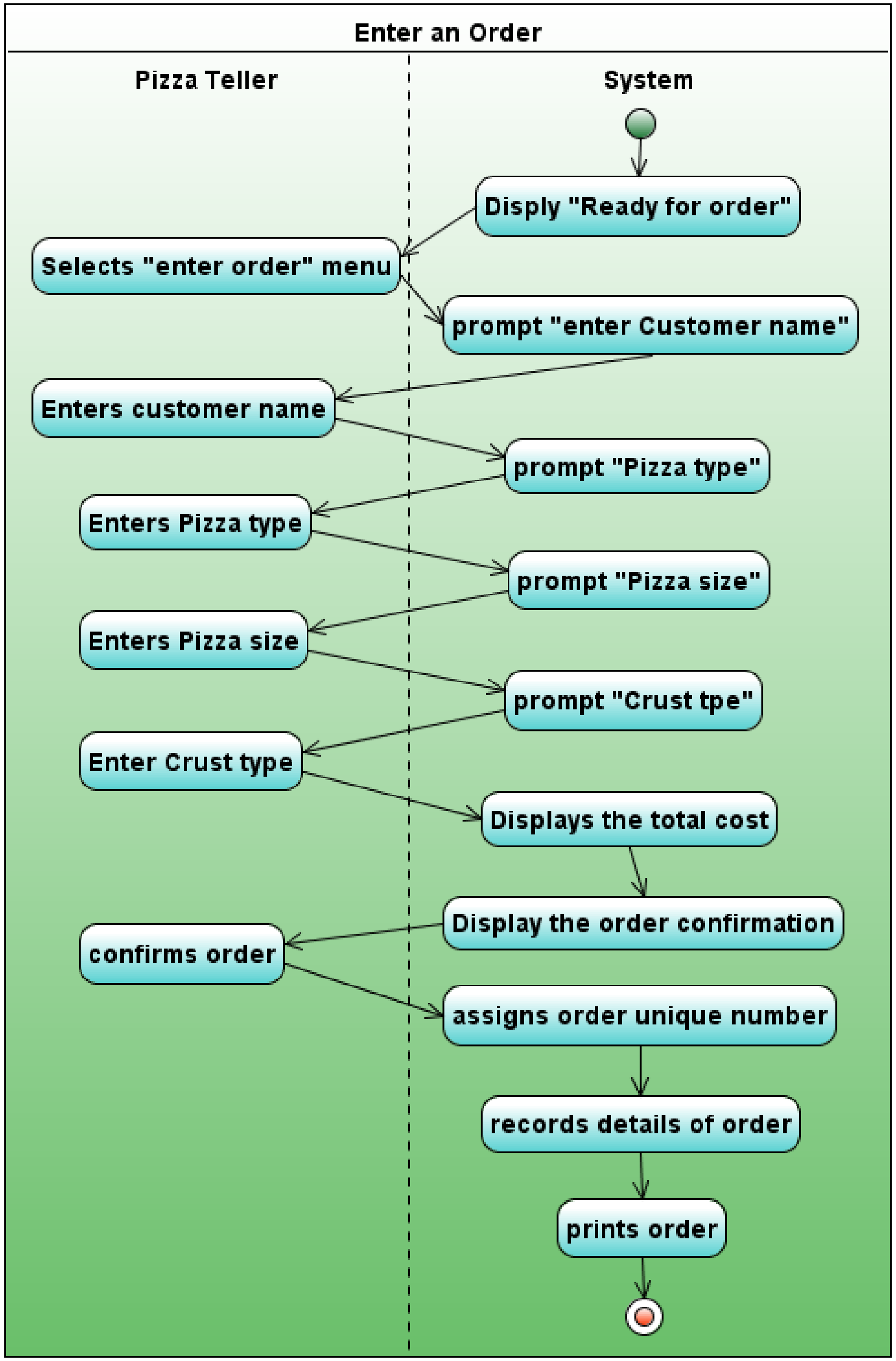
The Pizza Shop use case diagram



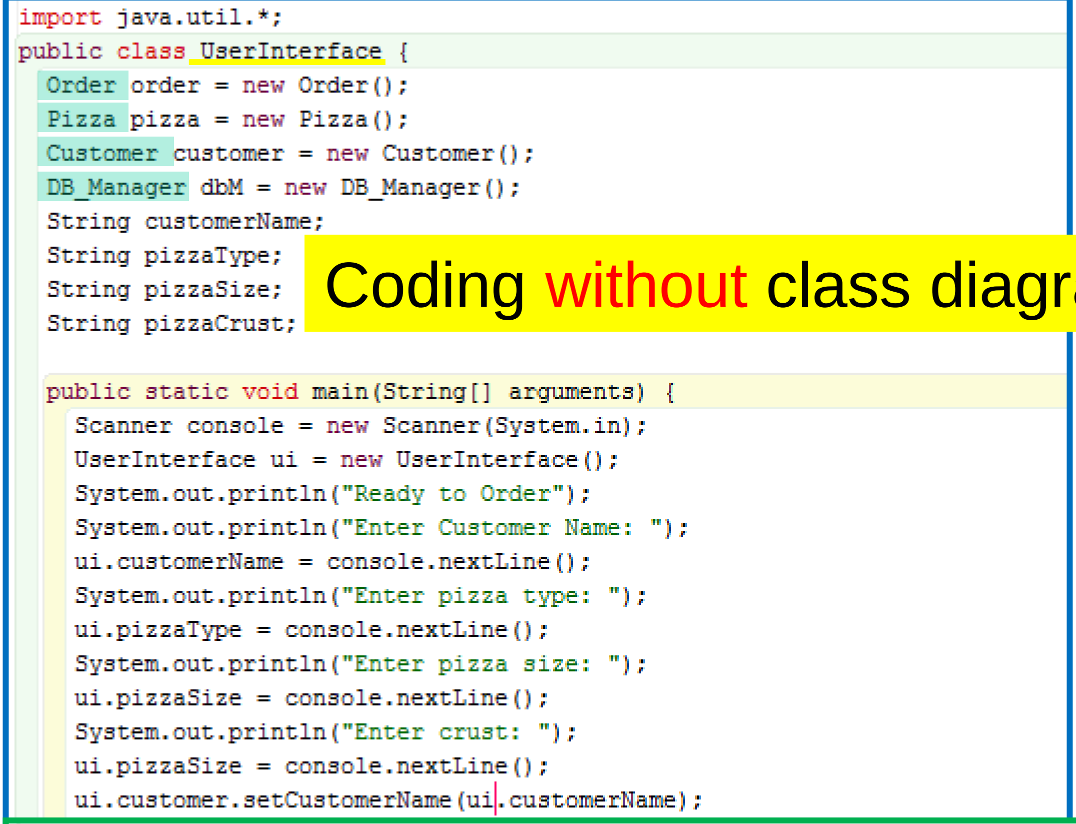
The Pizza Shop use case – Enter an Order

Use Case Name	Enter an Order	
Brief Description	Pizza Teller enters new order into the system	
Actors	Pizza Teller (or Manager)	
Related Use cases		
Entry condition	The system is waiting for user input.	
Exit condition	The system completes the order.	
Flow of Events	Actors	System
	1. Customer comes to pizza shop to get a pizza. Pizza Teller selects enter order Screen. 2. Pizza Teller enters Customer name 3. Pizza Teller enters Pizza type 4. Pizza Teller enters Pizza size 5. Pizza Teller enters Crust type 6. Pizza Teller confirms order	1.1 prompt “worker” to enter details of Customer name 2.1 prompts for Pizza type 3.1 prompts for Pizza size 4.1 prompts for Crust type 5.1 displays order for confirmation with pricing 6.1 assigns order unique number 6.2 records details of order 6.3 prints order
Exception condition		

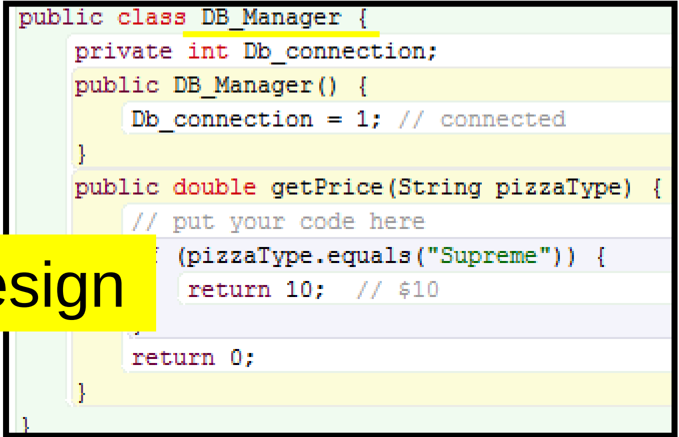
The Pizza an Order



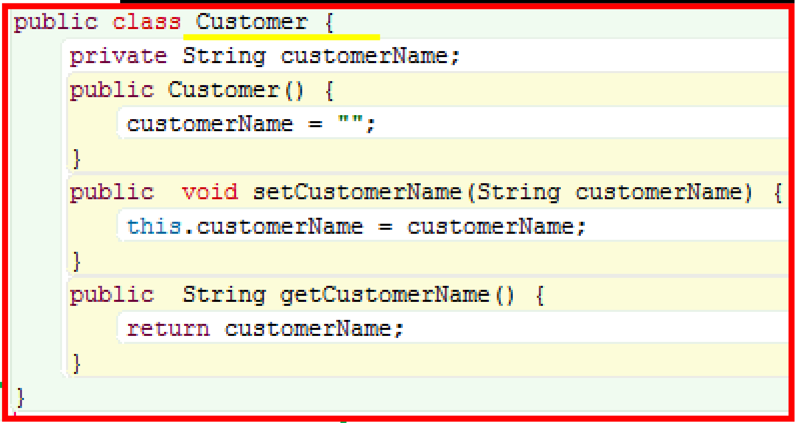
Coding without class diagram design



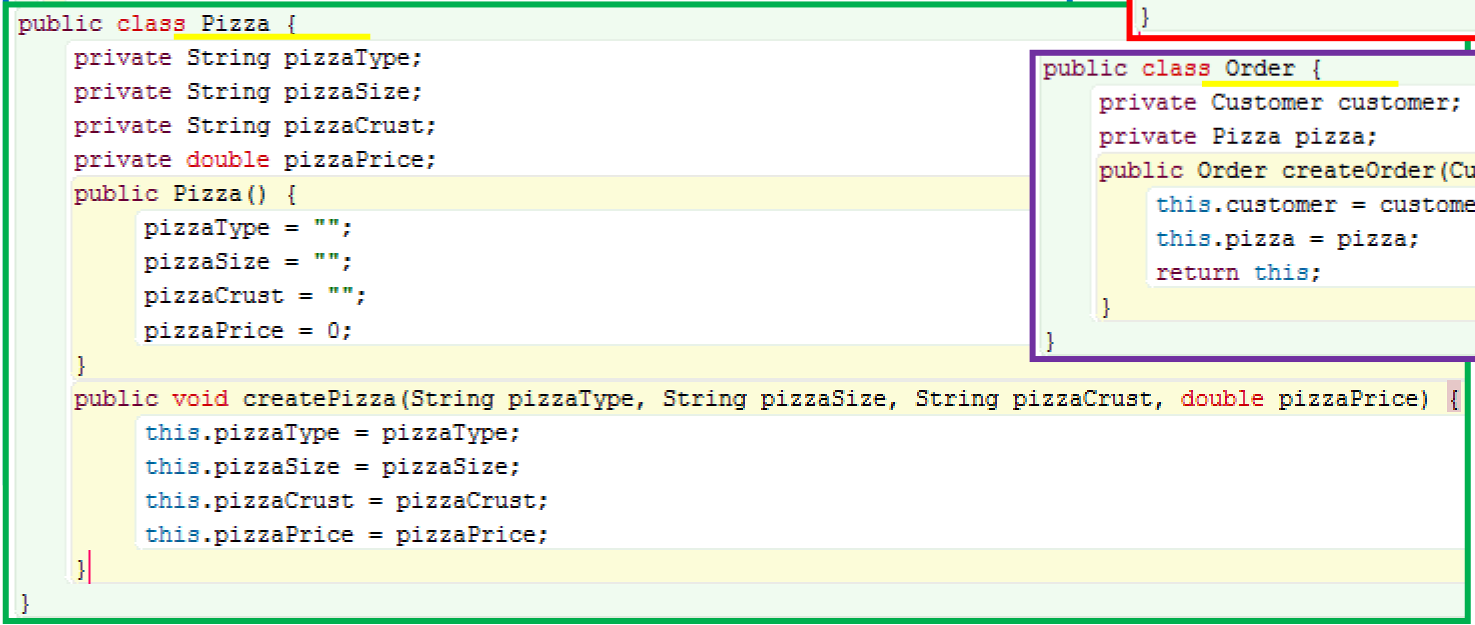
```
import java.util.*;
public class UserInterface {
    Order order = new Order();
    Pizza pizza = new Pizza();
    Customer customer = new Customer();
    DB_Manager dbM = new DB_Manager();
    String customerName;
    String pizzaType;
    String pizzaSize;
    String pizzaCrust;

    public static void main(String[] arguments) {
        Scanner console = new Scanner(System.in);
        UserInterface ui = new UserInterface();
        System.out.println("Ready to Order");
        System.out.println("Enter Customer Name: ");
        ui.customerName = console.nextLine();
        System.out.println("Enter pizza type: ");
        ui.pizzaType = console.nextLine();
        System.out.println("Enter pizza size: ");
        ui.pizzaSize = console.nextLine();
        System.out.println("Enter crust: ");
        ui.pizzaSize = console.nextLine();
        ui.customer.setCustomerName(ui.customerName);
```



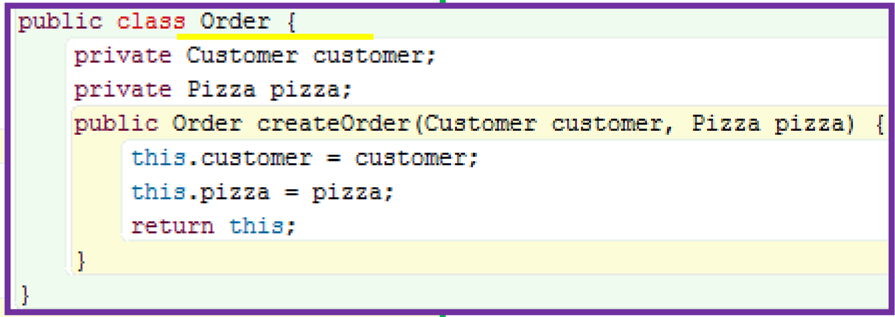
```
public class DB_Manager {
    private int Db_connection;
    public DB_Manager() {
        Db_connection = 1; // connected
    }
    public double getPrice(String pizzaType) {
        // put your code here
          (pizzaType.equals("Supreme")) {
            return 10;   // $10

        return 0;
    }
}
```



```
public class Customer {
    private String customerName;
    public Customer() {
        customerName = "";
    }
    public  void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
    public  String getCustomerName() {
        return customerName;
    }
}
```

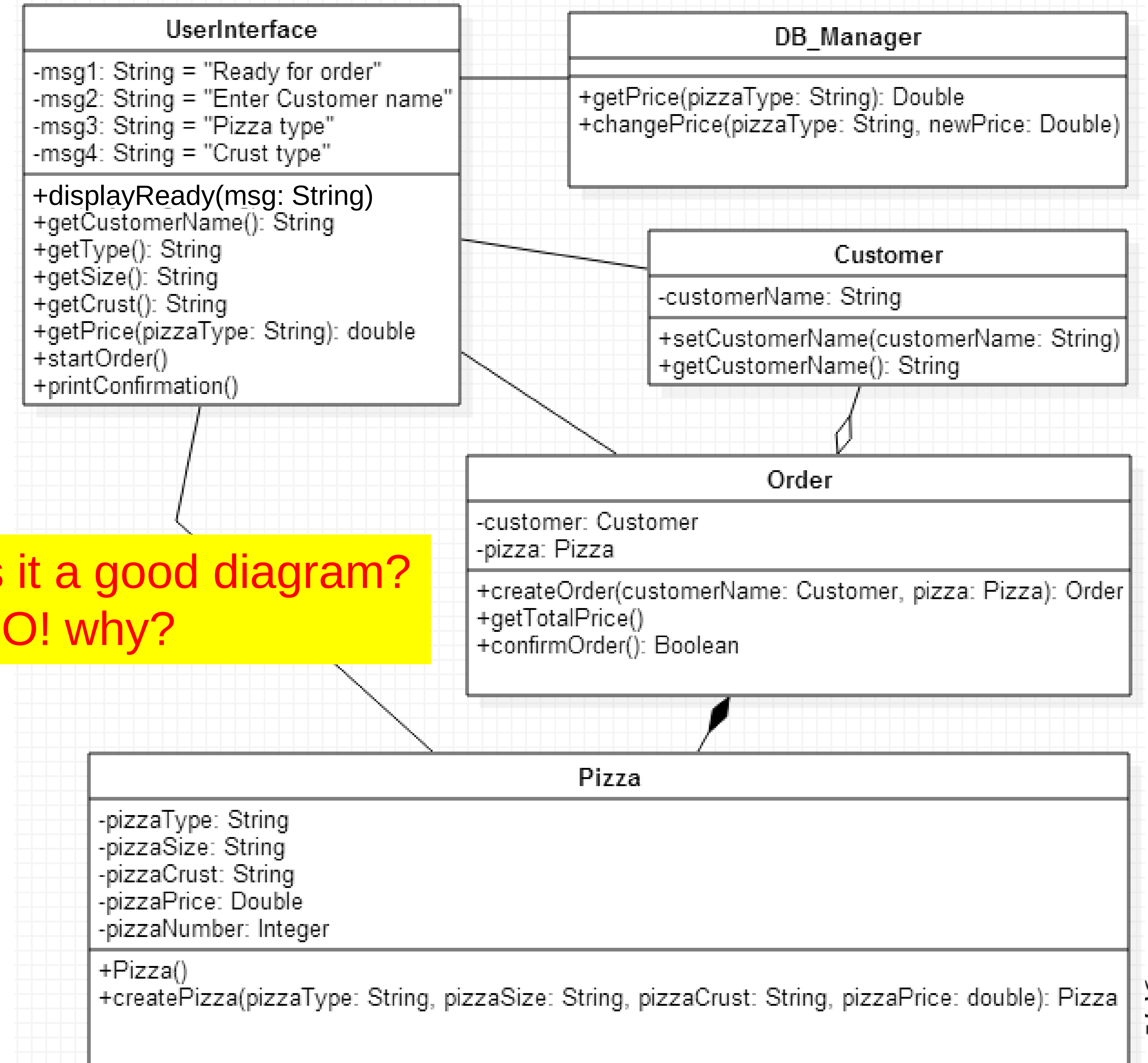


```
public class Pizza {
    private String pizzaType;
    private String pizzaSize;
    private String pizzaCrust;
    private double pizzaPrice;
    public Pizza() {
        pizzaType = "";
        pizzaSize = "";
        pizzaCrust = "";
        pizzaPrice = 0;
    }
    public void createPizza(String pizzaType, String pizzaSize, String pizzaCrust, double pizzaPrice) {
        this.pizzaType = pizzaType;
        this.pizzaSize = pizzaSize;
        this.pizzaCrust = pizzaCrust;
        this.pizzaPrice = pizzaPrice;
    }
}
```

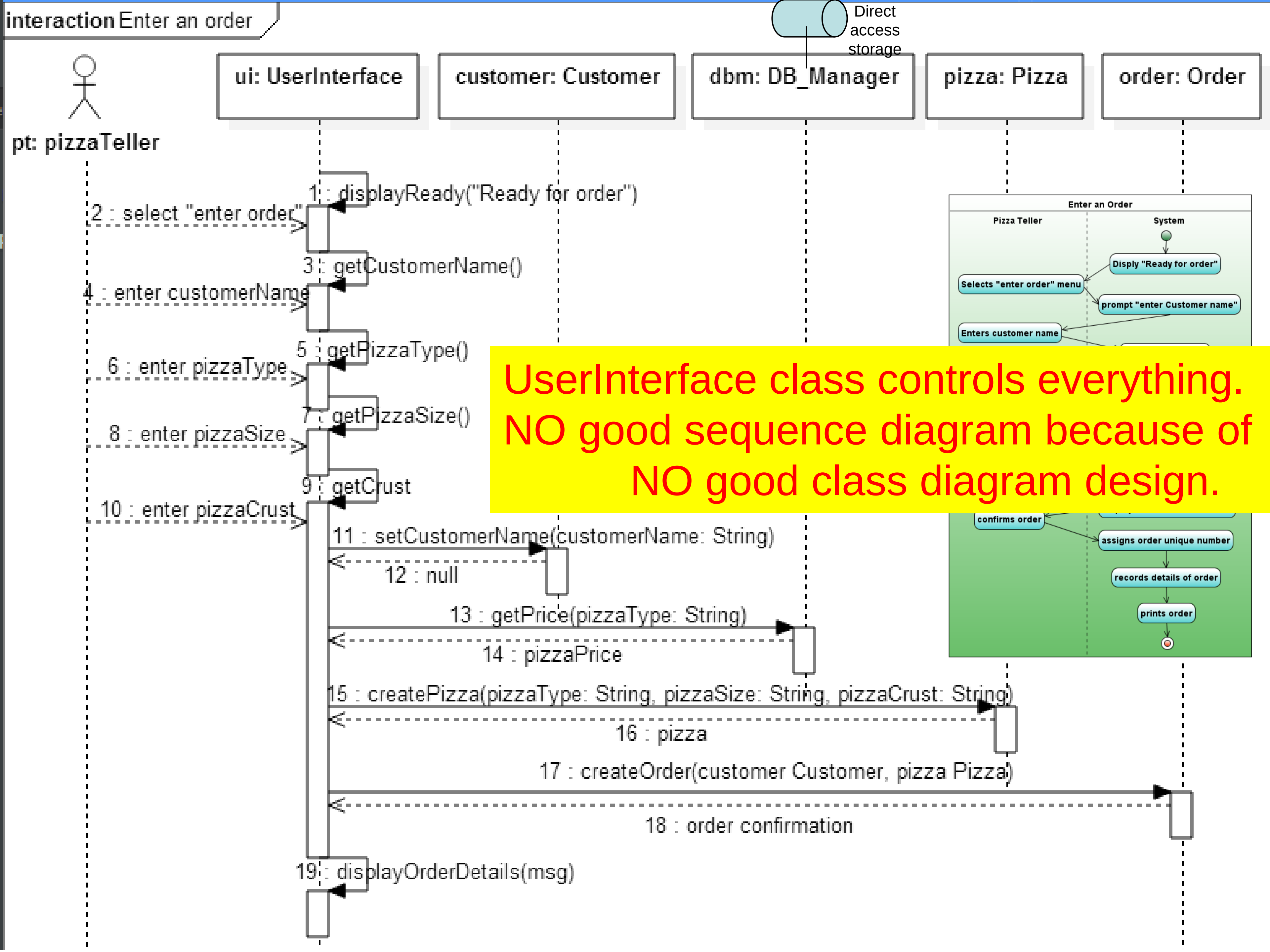


```
public class Order {
    private Customer customer;
    private Pizza pizza;
    public Order createOrder(Customer customer, Pizza pizza) {
        this.customer = customer;
        this.pizza = pizza;
        return this;
    }
}
```

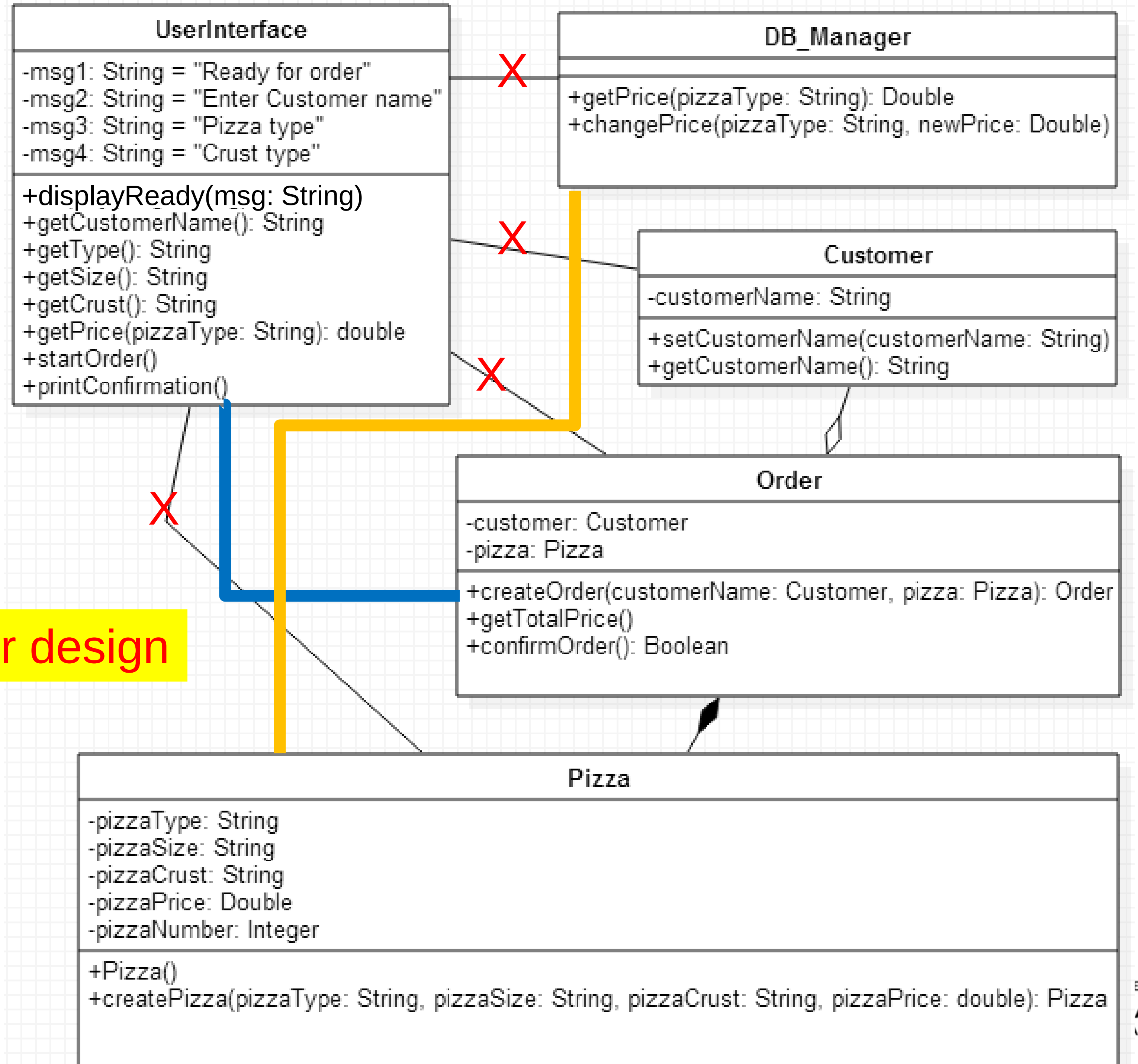
THE UNIVERSITY OF NEWCASTLE AUSTRALIA


UserInterface
-msg1: String = "Ready for order"
-msg2: String = "Enter Customer name"
-msg3: String = "Pizza type"
-msg4: String = "Crust type"
+displayReady(msg: String)
+getCustomerName(): String
+getType(): String
+getSize(): String
+getCrust(): String
+getPrice(pizzaType: String): double
+startOrder()
+printConfirmation()
DB_Manager
+getPrice(pizzaType: String): Double
+changePrice(pizzaType: String, newPrice: Double)
Customer
-customerName: String
+setCustomerName(customerName: String)
+getCustomerName(): String
Order
-customer: Customer
-pizza: Pizza
+createOrder(customerName: Customer, pizza: Pizza): Order
+getTotalPrice()
+confirmOrder(): Boolean
Pizza
-pizzaType: String
-pizzaSize: String
-pizzaCrust: String
-pizzaPrice: Double
-pizzaNumber: Integer
+Pizza()
+createPizza(pizzaType: String, pizzaSize: String, pizzaCrust: String, pizzaPrice: double): Pizza

Is it a good diagram?
NO! why?

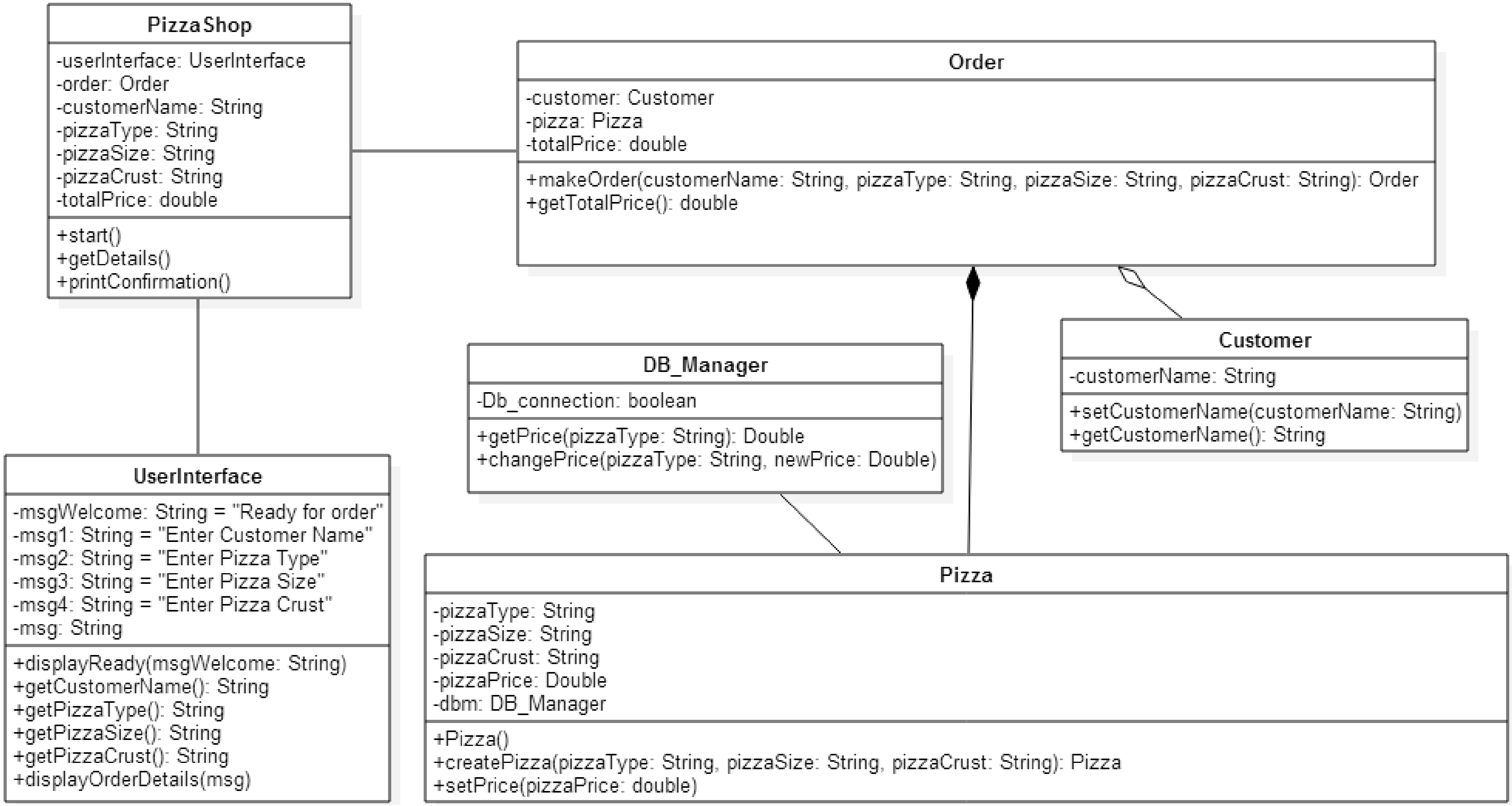


UserInterface class controls everything.
NO good sequence diagram because of
NO good class diagram design.

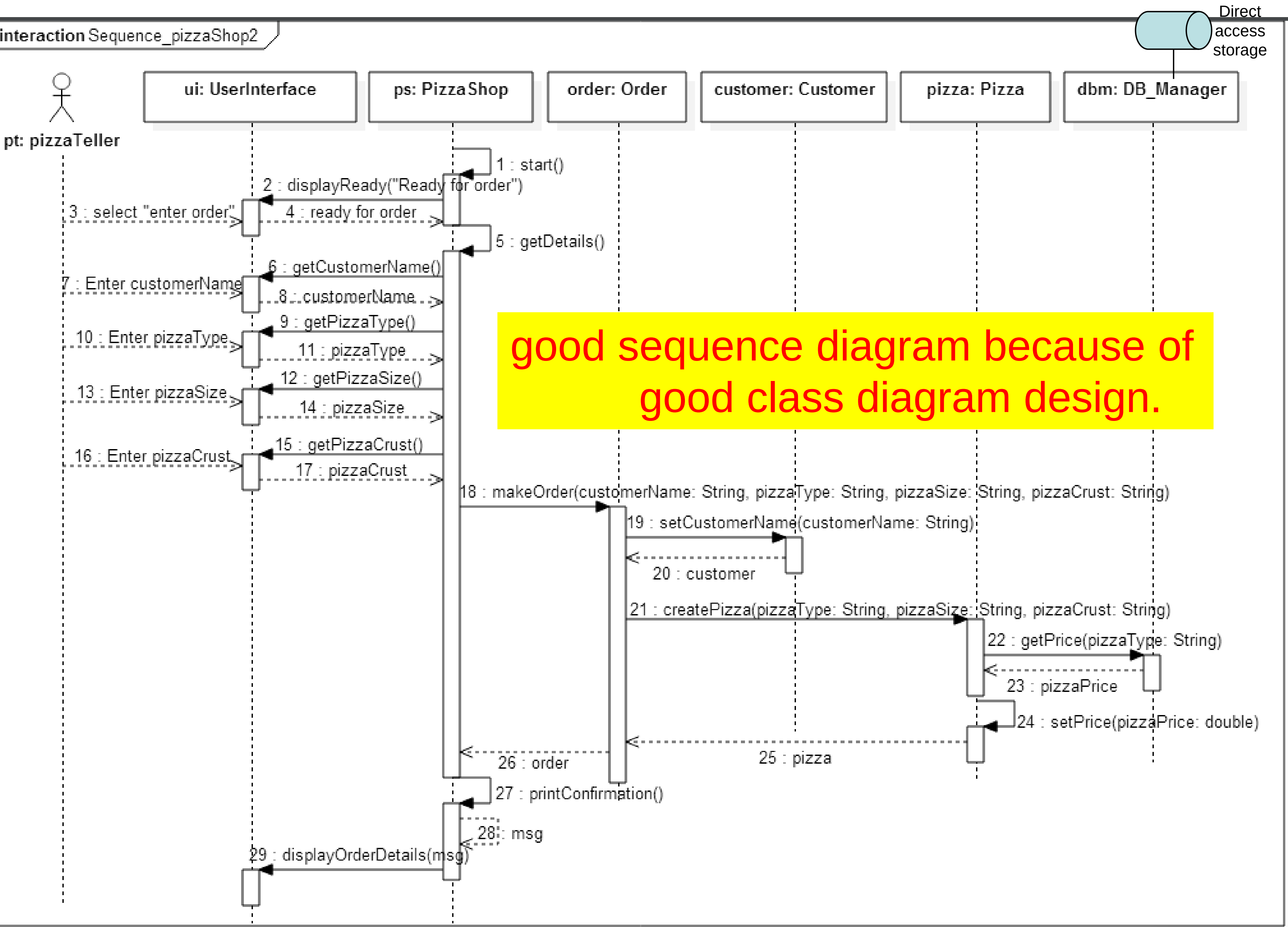

UserInterface
-msg1: String = "Ready for order"
-msg2: String = "Enter Customer name"
-msg3: String = "Pizza type"
-msg4: String = "Crust type"
+displayReady(msg: String)
+getCustomerName(): String
+getType(): String
+getSize(): String
+getCrust(): String
+getPrice(pizzaType: String): double
+startOrder()
+printConfirmation()
DB_Manager
+getPrice(pizzaType: String): Double
+changePrice(pizzaType: String, newPrice: Double)
Customer
-customerName: String
+setCustomerName(customerName: String)
+getCustomerName(): String
Order
-customer: Customer
-pizza: Pizza
+createOrder(customerName: Customer, pizza: Pizza): Order
+getTotalPrice()
+confirmOrder(): Boolean
Pizza
-pizzaType: String
-pizzaSize: String
-pizzaCrust: String
-pizzaPrice: Double
-pizzaNumber: Integer
+Pizza()
+createPizza(pizzaType: String, pizzaSize: String, pizzaCrust: String, pizzaPrice: double): Pizza
X
X
X
X

Better design

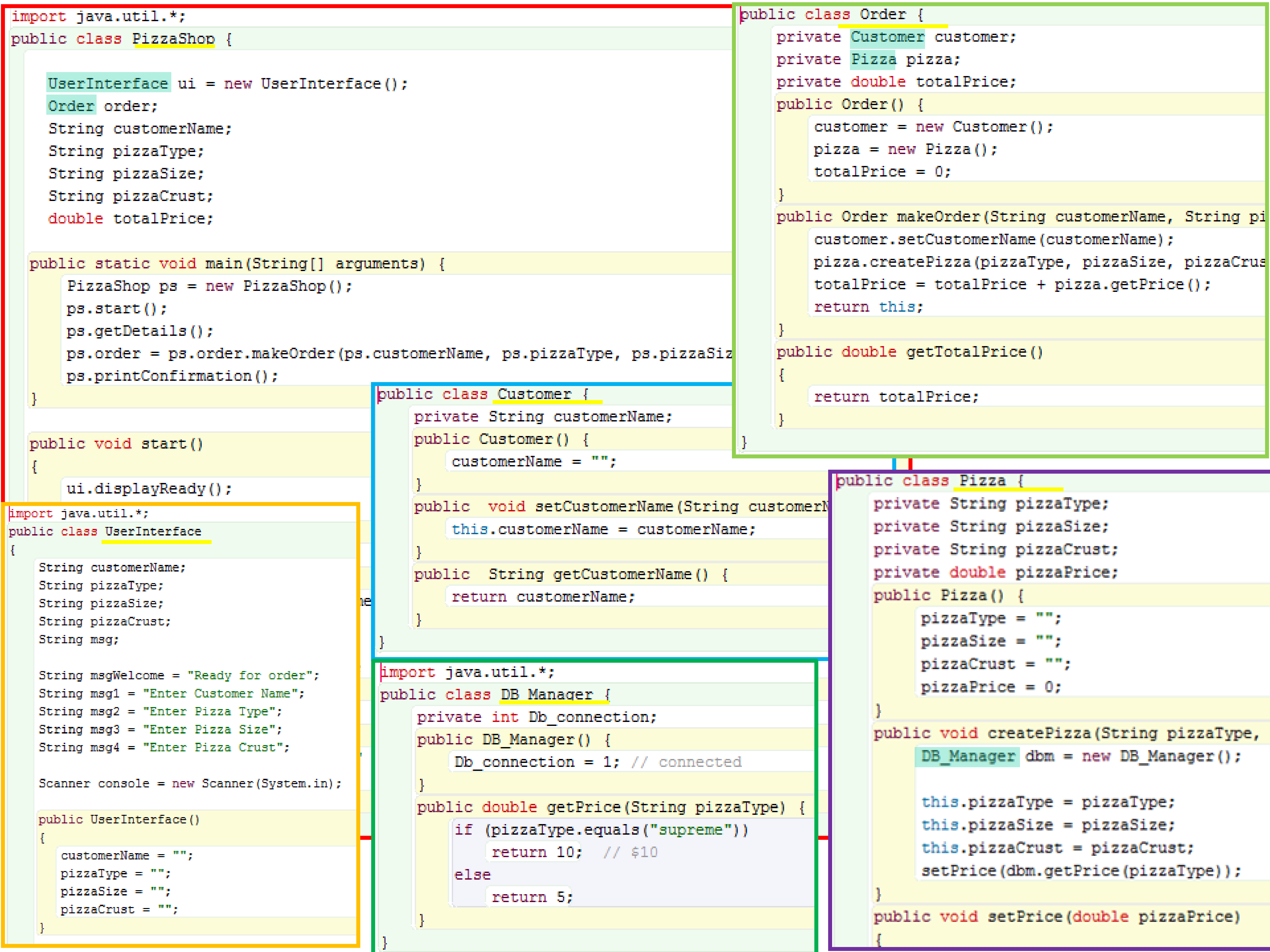
Pizza Shop Class Diagram


PizzaShop
-userInterface: UserInterface
-order: Order
-customerName: String
-pizzaType: String
-pizzaSize: String
-pizzaCrust: String
-totalPrice: double
+start()
+getDetails()
+printConfirmation()
Order
-customer: Customer
-pizza: Pizza
-totalPrice: double
+makeOrder(customerName: String, pizzaType: String, pizzaSize: String, pizzaCrust: String): Order
+getTotalPrice(): double
Customer
-customerName: String
+setCustomerName(customerName: String)
+getCustomerName(): String
DB_Manager
-Db_connection: boolean
+getPrice(pizzaType: String): Double
+changePrice(pizzaType: String, newPrice: Double)
UserInterface
-msgWelcome: String = "Ready for order"
-msg1: String = "Enter Customer Name"
-msg2: String = "Enter Pizza Type"
-msg3: String = "Enter Pizza Size"
-msg4: String = "Enter Pizza Crust"
-msg: String
+displayReady(msgWelcome: String)
+getCustomerName(): String
+getPizzaType(): String
+getPizzaSize(): String
+getPizzaCrust(): String
+displayOrderDetails(msg)
Pizza
-pizzaType: String
-pizzaSize: String
-pizzaCrust: String
-pizzaPrice: Double
-dbm: DB_Manager
+Pizza()
+createPizza(pizzaType: String, pizzaSize: String, pizzaCrust: String): Pizza
+setPrice(pizzaPrice: double)

THE UNIVERSITY OF NEWCASTLE AUSTRALIA



good sequence diagram because of good class diagram design.



```java
import java.util.*;
public class PizzaShop {

    UserInterface ui = new UserInterface();
    Order order;
    String customerName;
    String pizzaType;
    String pizzaSize;
    String pizzaCrust;
    double totalPrice;

  public static void main(String[] arguments) {
      PizzaShop ps = new PizzaShop();
      ps.start();
      ps.getDetails();
      ps.order = ps.order.makeOrder(ps.customerName, ps.pizzaType, ps.pizzaSiz
      ps.printConfirmation();
  }

  public void start()
  {
      ui.displayReady();
```

```java
public class Order {
    private Customer customer;
    private Pizza pizza;
    private double totalPrice;
    public Order() {
        customer = new Customer();
        pizza = new Pizza();
        totalPrice = 0;
    }
    public Order makeOrder(String customerName, String pi
        customer.setCustomerName(customerName);
        pizza.createPizza(pizzaType, pizzaSize, pizzaCrus
        totalPrice = totalPrice + pizza.getPrice();
        return this;
    }
    public double getTotalPrice()
    {
        return totalPrice;
    }
}
```

```java
public class Customer {
    private String customerName;
    public Customer() {
        customerName = "";
    }
    public  void setCustomerName(String customerN
        this.customerName = customerName;
    }
    public  String getCustomerName() {
        return customerName;
    }
}
```

```java
import java.util.*;
public class UserInterface
{
    String customerName;
    String pizzaType;
    String pizzaSize;
    String pizzaCrust;
    String msg;

    String msgWelcome = "Ready for order";
    String msg1 = "Enter Customer Name";
    String msg2 = "Enter Pizza Type";
    String msg3 = "Enter Pizza Size";
    String msg4 = "Enter Pizza Crust";

    Scanner console = new Scanner(System.in);

    public UserInterface()
    {
      customerName = "";
      pizzaType = "";
      pizzaSize = "";
      pizzaCrust = "";
    }
```

```java
import java.util.*;
public class DB_Manager {
    private int Db_connection;
    public DB_Manager() {
        Db_connection = 1; // connected
    }
    public double getPrice(String pizzaType) {
        if (pizzaType.equals("supreme"))
            return 10;   // $10
        else
            return 5;
    }
}
```

```java
public class Pizza {
    private String pizzaType;
    private String pizzaSize;
    private String pizzaCrust;
    private double pizzaPrice;
    public Pizza() {
         pizzaType = "";
         pizzaSize = "";
         pizzaCrust = "";
         pizzaPrice = 0;
    }
    public void createPizza(String pizzaType,
         DB_Manager dbm = new DB_Manager();

         this.pizzaType = pizzaType;
         this.pizzaSize = pizzaSize;
         this.pizzaCrust = pizzaCrust;
         setPrice(dbm.getPrice(pizzaType));
    }
    public void setPrice(double pizzaPrice)
    {
```

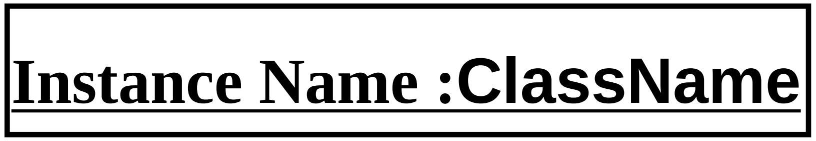
This Week

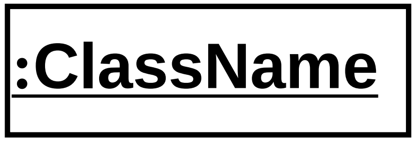
1. Sequence Diagrams
2. Advanced modelling
   - More details

Sequence Diagram

- The interactions between objects in the sequential order
- It shows how the objects interact with others in particular scenario of a use case
  - the flow of events in the use case description

Sequence Diagram: Notation

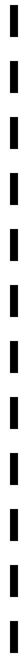




- **Object:** Objects are instances of classes. Object is represented as a rectangle which contains the name of the object underlined.

- Because the system is instantiated, it is shown as an object.

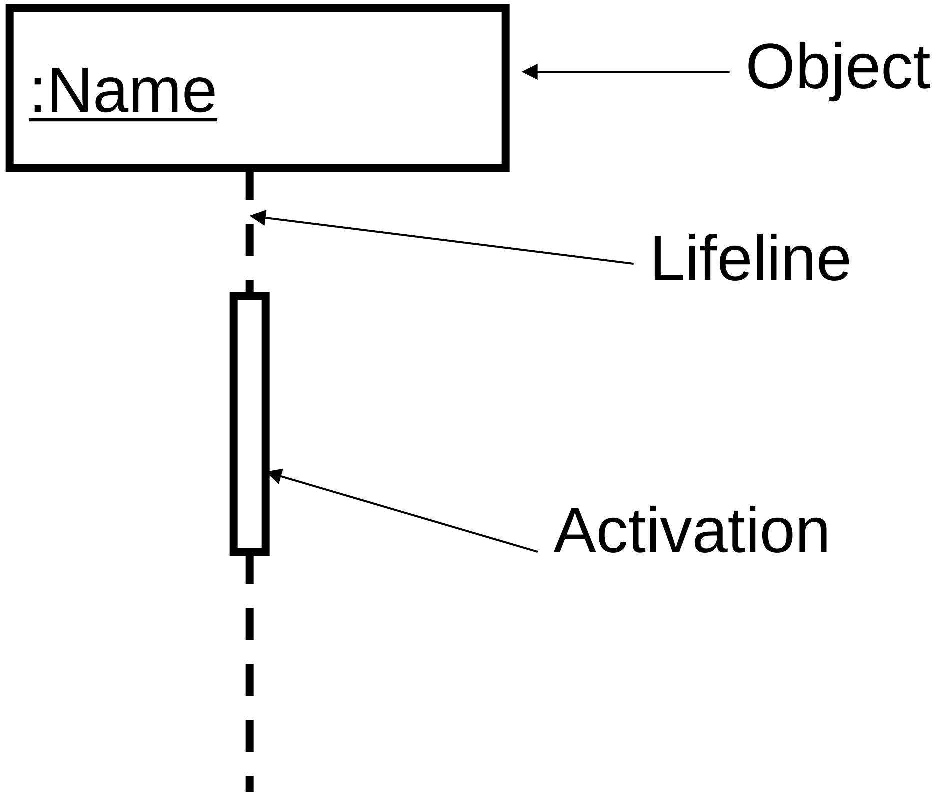
Sequence Diagram: Notation



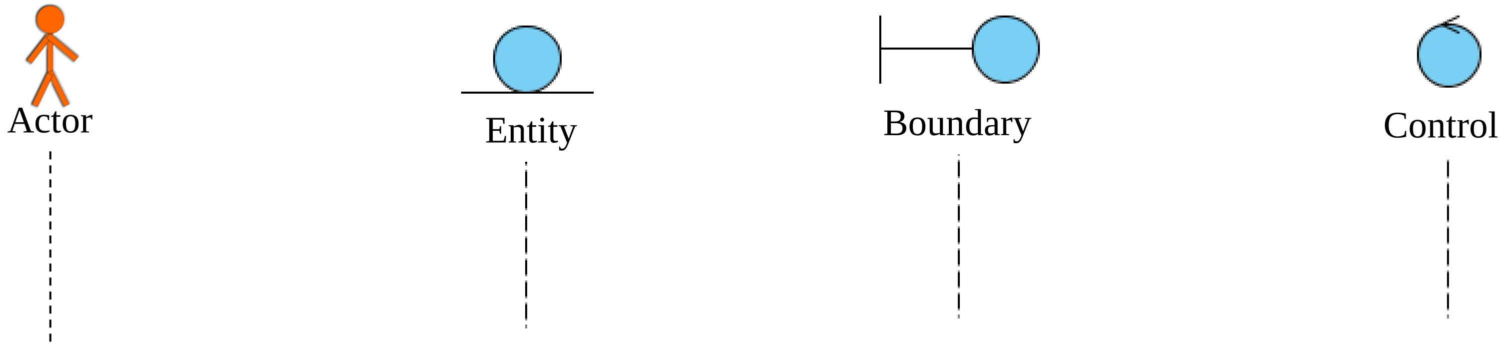
- **Lifeline**
  - The Lifeline identifies the existence of the object over time.
  - The notation for a Lifeline is a vertical dotted line extending from an object

Sequence Diagram - Objects

- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated



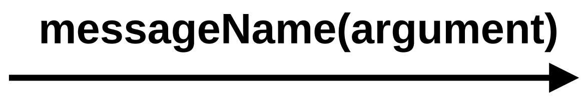
Sequence Diagram: Notation



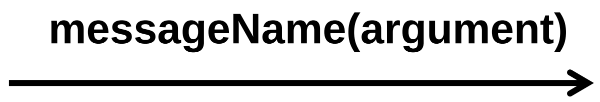
- LifeLine
  - A lifeline represents an individual participant in the interaction.
  - This will usually be the case if the sequence diagram is owned by a use case.
  - Actors may represent roles played by human users, external hardware, or other subjects.
  - Entity, Boundary, and Control elements can also own lifelines.

Sequence Diagram: Notation

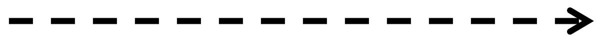
- **Message type:** Messages, modeled as horizontal arrows between Activations, indicate the communications between objects.



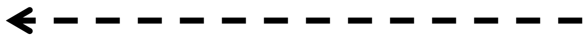
Synchronous: indicates wait semantics, the sender waits for the message to be handled before it continues.



Asynchronous: indicates no-wait semantics, the sender does not wait for the message before it continues. With an asynchronous flow of control, there is no explicit return message to the caller.



Create: results in the creation of a new object. The message could call a constructor (Java).



Reply: This shows the return message from another message.

Sequence Diagram: Notation

- Message type: Messages, modelled as horizontal arrows between Activations, indicate the communications between objects.

- Complete message notation

*[true/false condition] return-value := message-name (parameter-list)

*[another item] description, price, extendedPrice := addItem (itemID, quantity)

Loop

True/False Condition

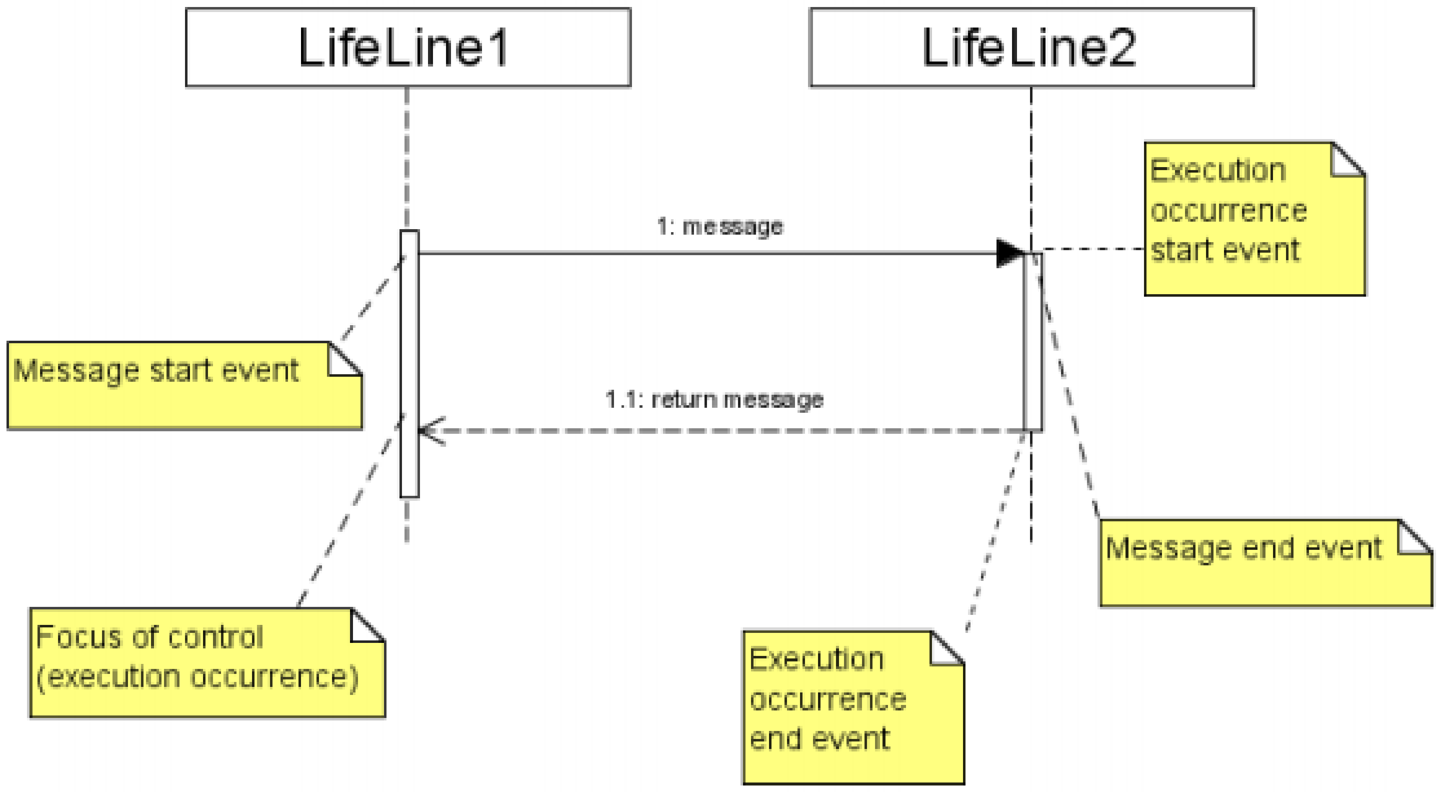
Expected output

Message-name

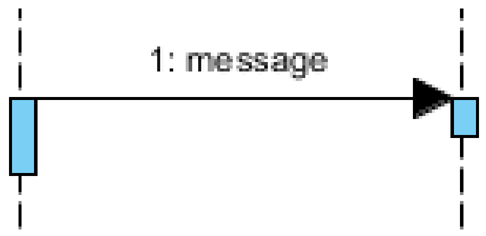
Input

Sequence Diagram: Notation

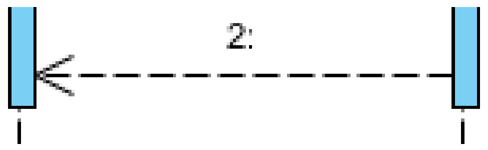
- Messages


LifeLine1
LifeLine2
1: message
1.1: return message
Execution occurrence start event
Message start event
Focus of control (execution occurrence)
Execution occurrence end event
Message end event

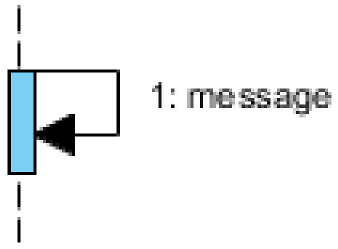
Sequence Diagram: Notation



- Send message
  - A message defines a particular communication between Lifelines of an interaction
  - Call message is a kind of message that represents an invocation of operation of target lifeline.

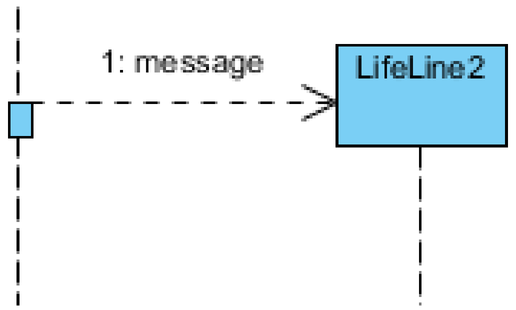


- Return message
  - Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message

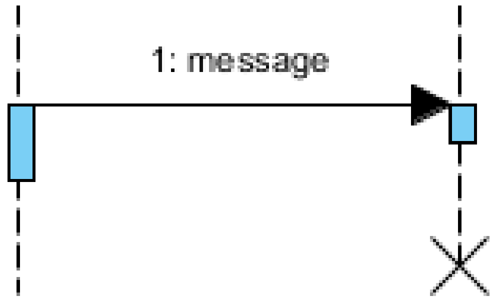


- Self Message
  - Is a kind of message that represents the invocation of message on the same lifeline.

Sequence Diagram: Notation

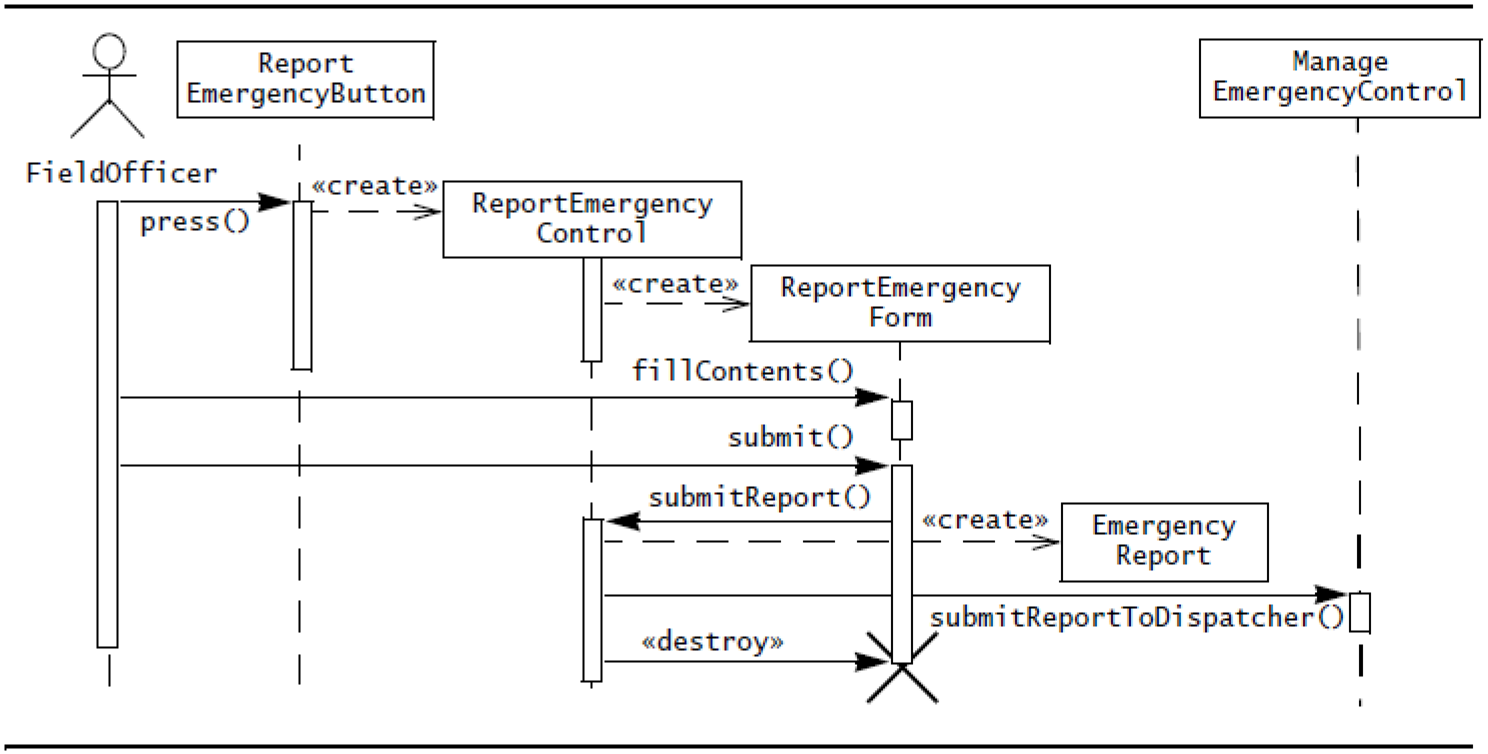


- Create Object
  - An object created after the start of the sequence appears lower than the others

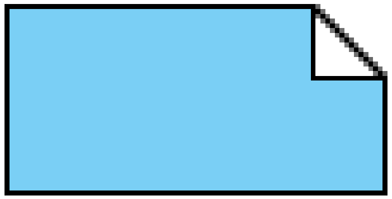


- Destroy Object
  - Destroy object is a kind of message that represents the request of destroying the lifecycle of target lifeline.

Sequence Diagram: Notation

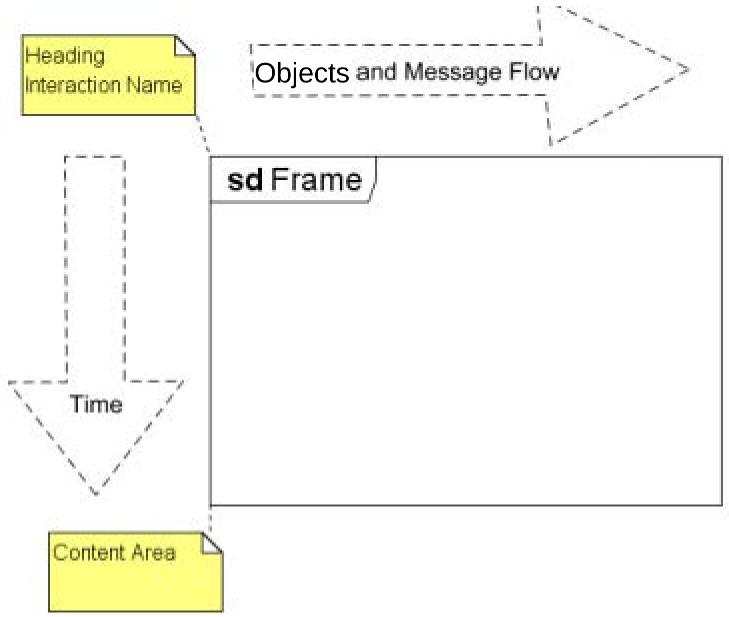


Sequence Diagram: Notation



- Note
  - A note (comment) gives the ability to attach various remarks to elements.
  - A comment carries no semantic force, but may contain information that is useful a modeller

Sequence Diagram: Notation



- Frame
  - Represents an interaction, which is a unit of behaviour that focuses on the observable exchange of information between Connectable Elements
  - **sd** for sequence diagram

Sequence Diagram: Combined Fragment

Combined Fragment

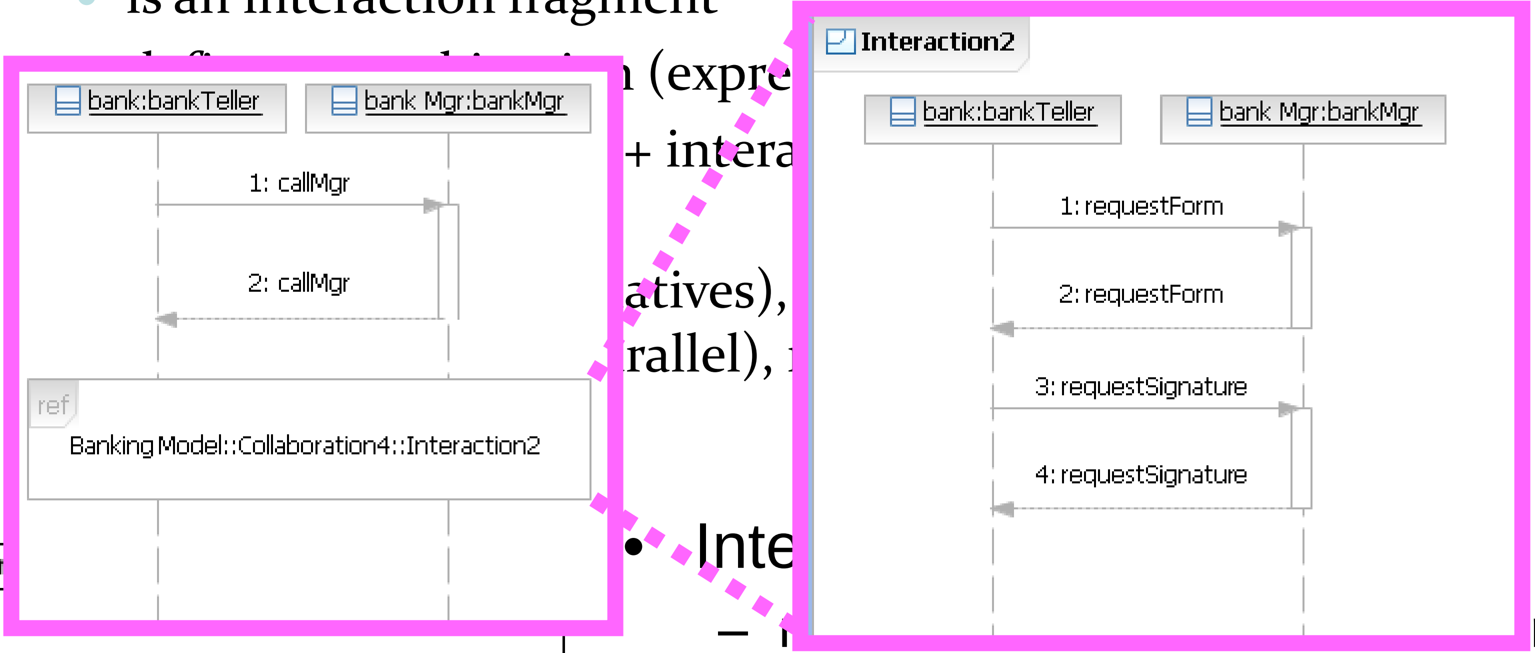
- is an interaction fragment

n (expre

+ intera

atives),

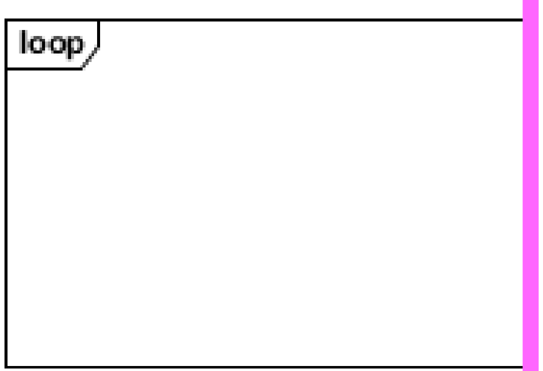
rallel),

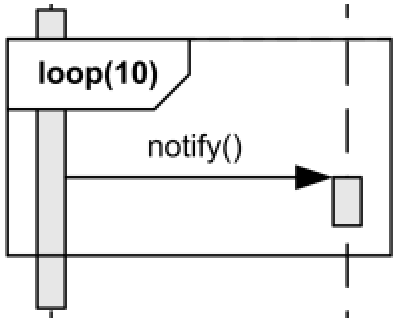


- Inte

– raction use is a shorthand for copying the contents of the referred interaction where the InteractionUse is.

Sequence Diagram: Combined Fragment

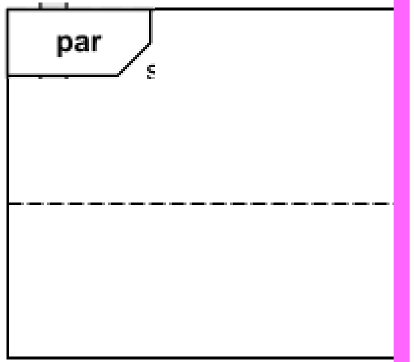


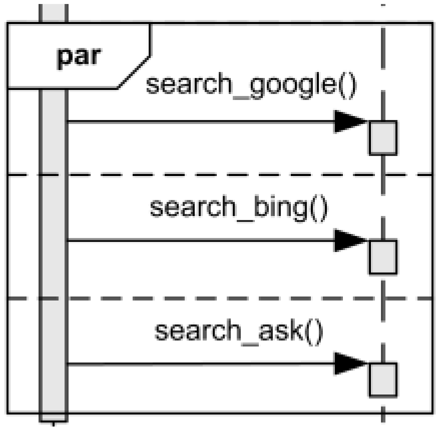


Loop to execute exactly 10 times.

ined Fragment

expression of interaction
. The loop operand will be
a number of times.



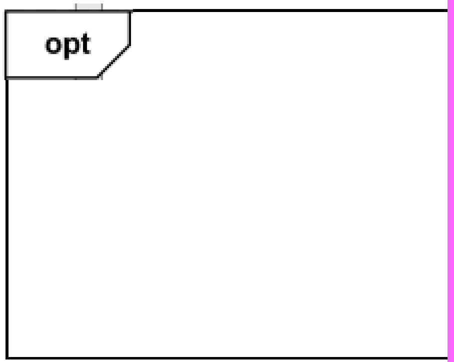


Search Google, Bing and Ask in any order, possibly parallel.

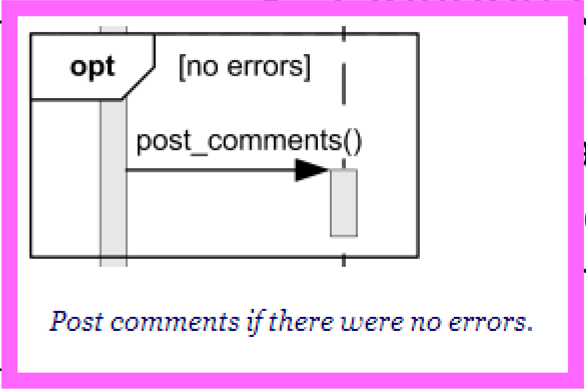
ed Fragment

ally parallel execution
f the operands.

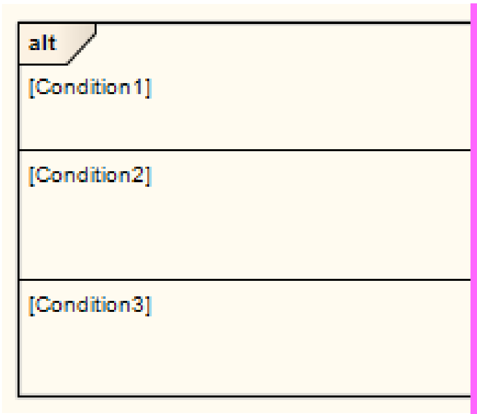
Sequence Diagram: Combined Fragment

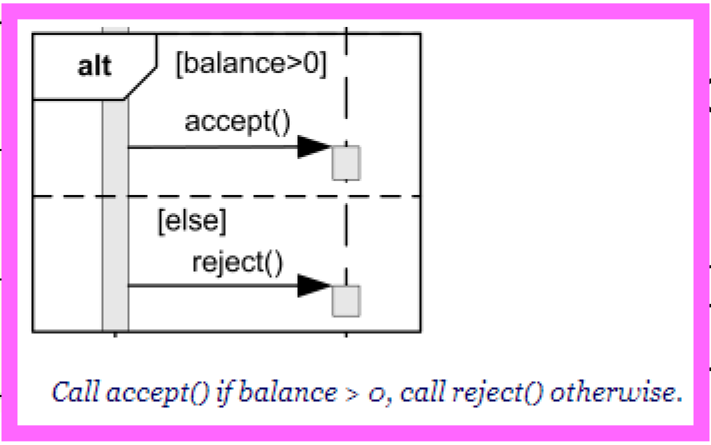


- Options Combined Fragment
n fragment represents a choice
aviour where either the (sole)
d happens or nothing happens
n" logic).



Post comments if there were no errors.



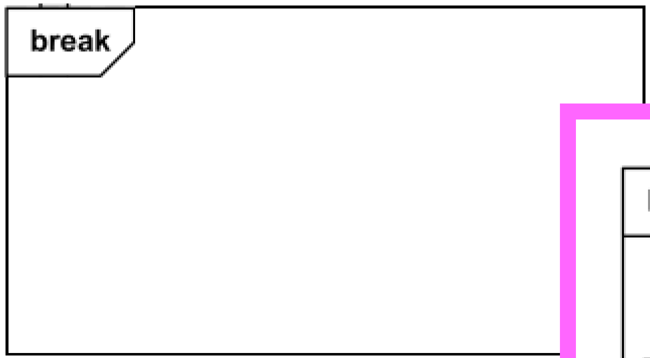


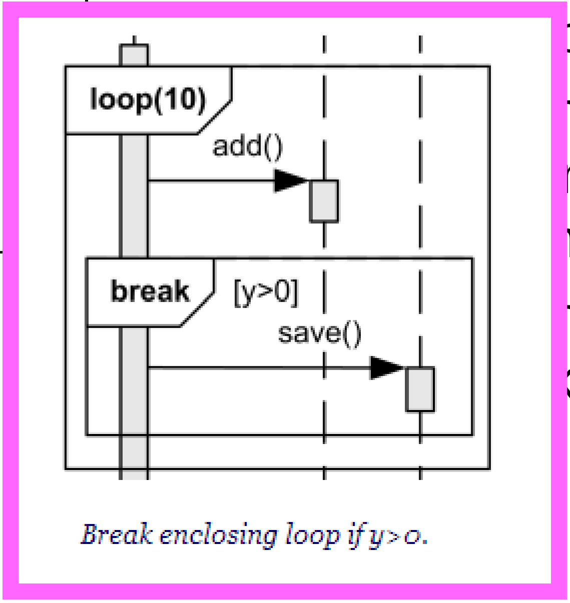
Call accept() if balance > 0, call reject() otherwise.

ombined Combined

. choice of behaviour. At
the operands will be
hen else" logic).

Sequence Diagram: Combined Fragment

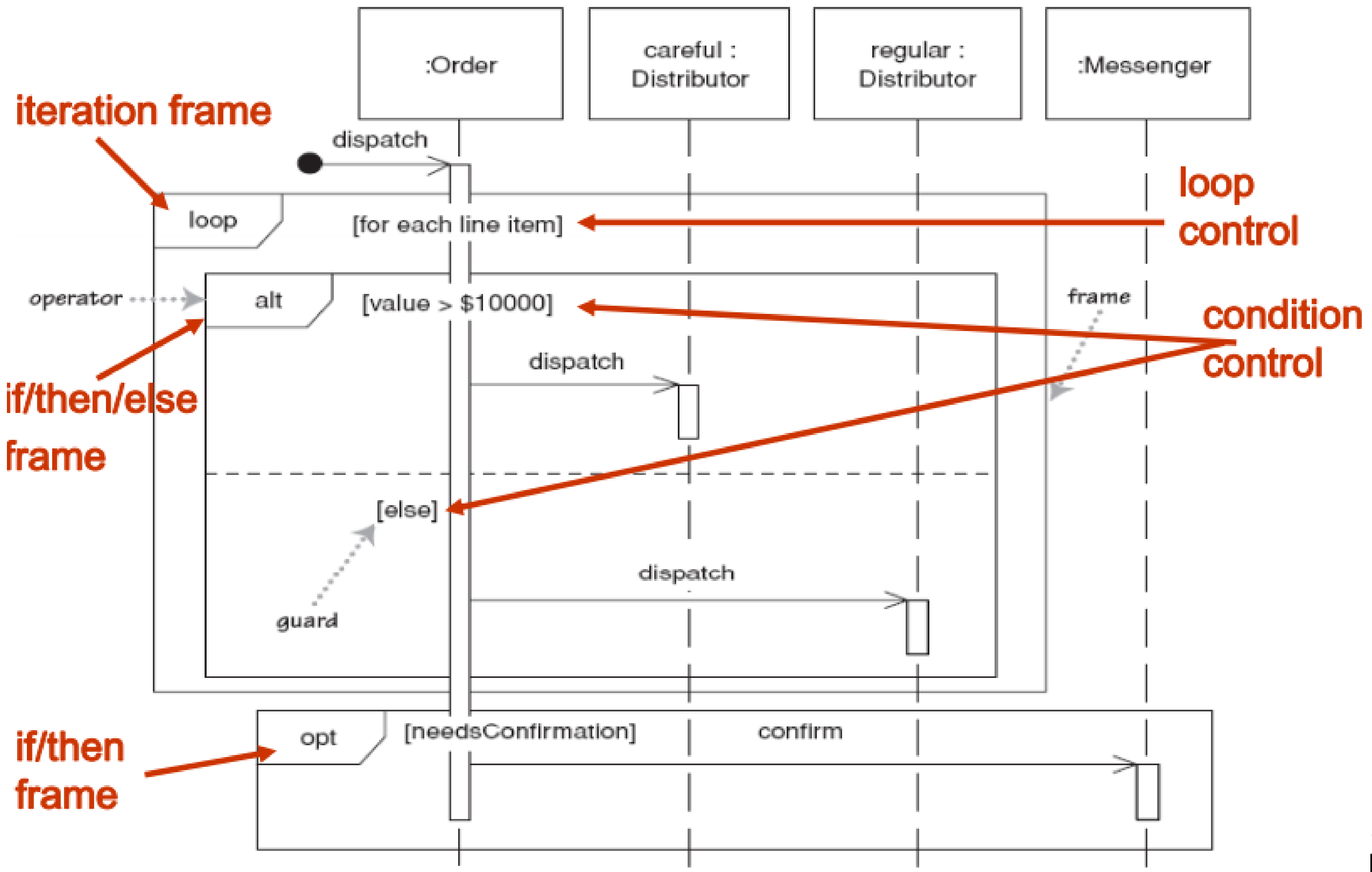




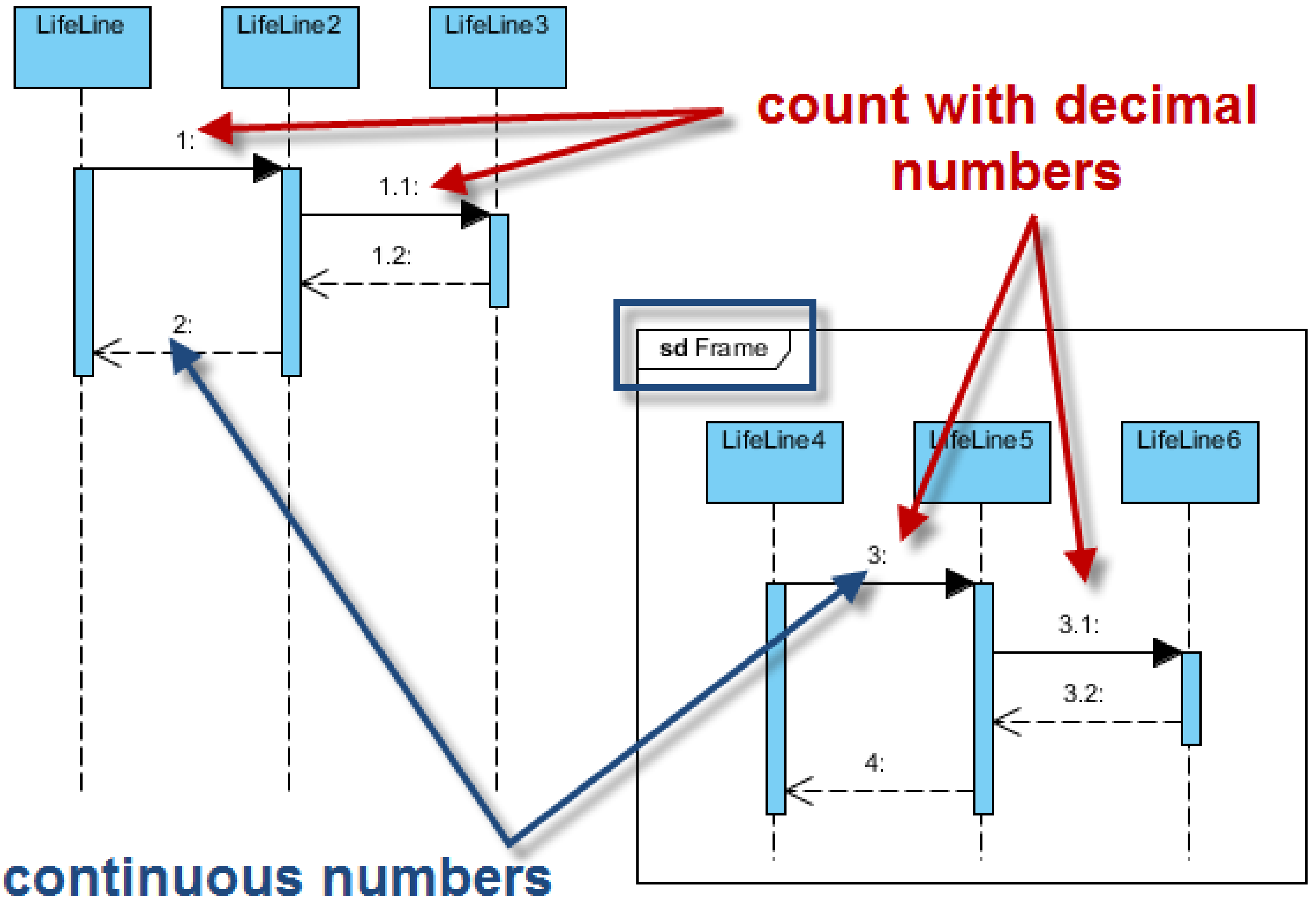
Break enclosing loop if y>0.

- Break Combined Fragment
  - represents a breaking or ptional scenario
  - n the condition is false, nclosing interaction nent proceeds.
  - n the condition is true, a k operator is performed.

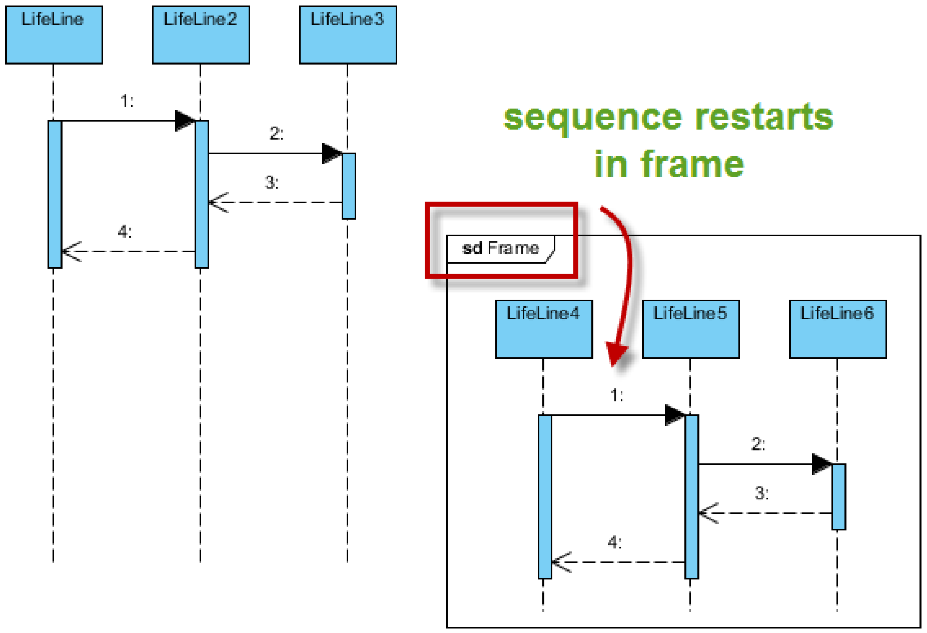
Sequence Diagram: Combined Fragment

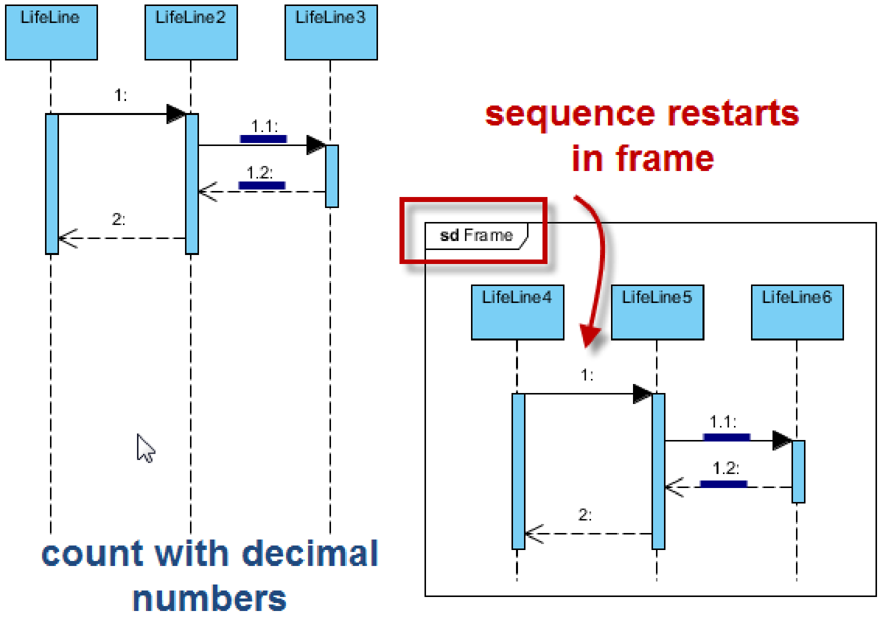


Sequence Diagram: Diagram-based Numbering



Sequence Diagram: Frame-based Numbering

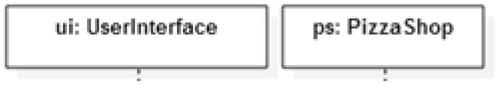




Sequence Diagram: Build

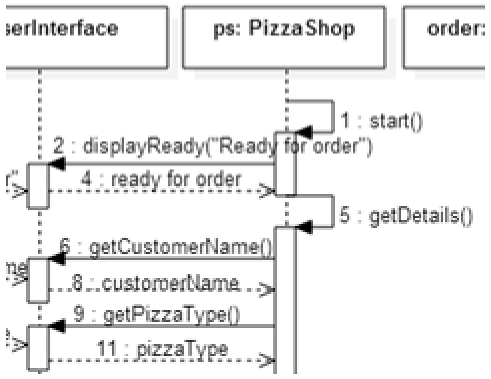
1. Finding objects by examining Use Case Scenarios

2. Add objects to the sequence diagram



3. Draw message lines between objects

4. Complete the sequence diagram
    - 4.1 Add activation bars
    - 4.2 Add a method to a message line



Sequence Diagram: Build

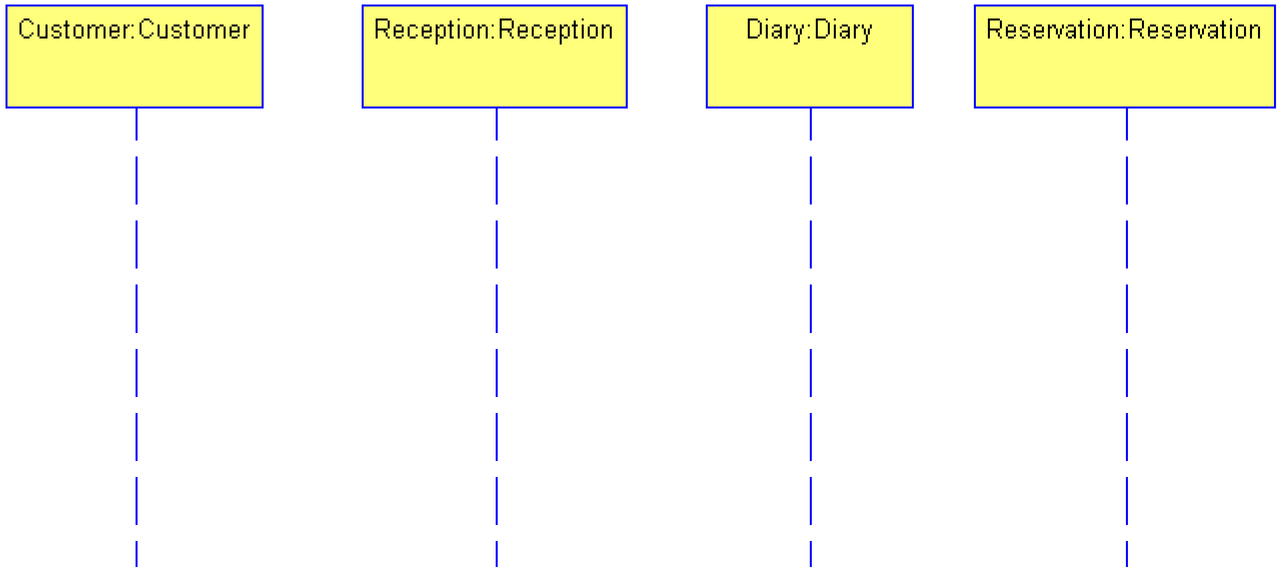
- Scenarios: Create a sequence diagram of the interaction between a receptionist and a customer in a hotel business
    1. Customer Queries for Available Rooms
    2. Store Customer Details
    3. Check Diary for Room Availability
    4. Room is Available
    5. Advise Customer of Availability
    6. Customer Requests Reservation
    7. Provisionally Book Room
    8. Figure Out Price, Advise Customer
    9. Customer Accepts Terms
    10. Check Customer Credit
    11. Customer Credit is OK
    12. Reserve Room

Sequence Diagram: Build

- Scenarios: Create a sequence diagram of the interaction between a receptionist and a customer in a hotel business
  1. Customer Queries for Available Rooms
  2. Store Customer Details
  3. Check Diary for Room Availability
  4. Room is Available
  5. Advise Customer of Availability
  6. Customer Requests Reservation
  7. Provisionally Book Room
  8. Figure Out Price, Advise Customer
  9. Customer Accepts Terms
  10. Check Customer Credit
  11. Customer Credit is OK
  12. Reserve Room

Sequence Diagram: Build

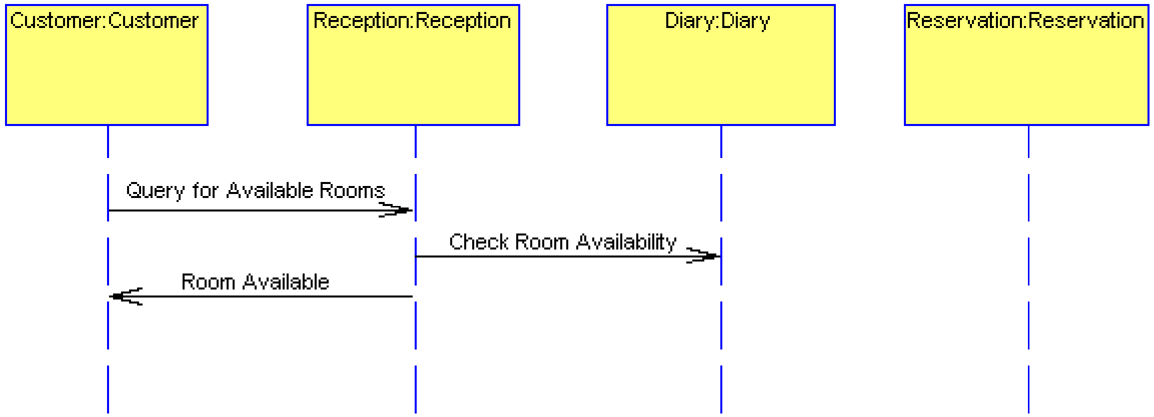
1. Finding objects by examining Use Case Scenarios
   - Use case diagram, Class diagram
2. Add objects to the sequence diagram
   - Add objects: Customer, Reception, Reservation, Diary and Room



Sequence Diagram: Build

3. Draw Message lines between objects
   - Message lines are drawn between objects to show how and when they communicate.
   - The message line represents a message sent from one object to another.
   - The 'from' object is requesting that an operation be performed by the 'to' object.
   - The 'to' object performs the operation using a method that its class

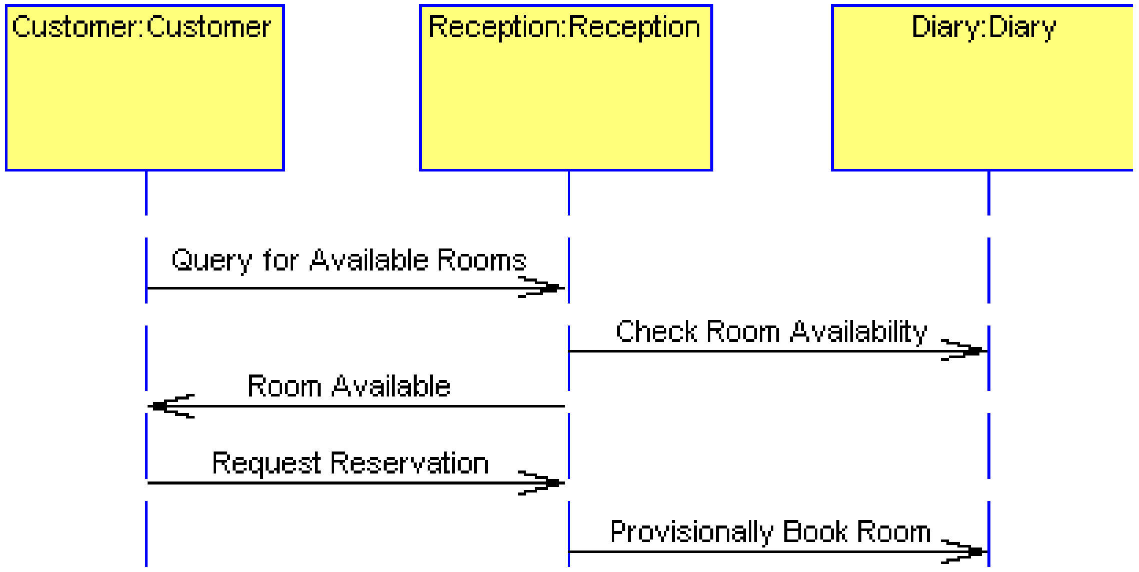
1. Customer Queries for Available Rooms
2. Store Customer Details
3. Check Diary for Room Availabilit
4. Room is Available
5. Advise Customer of Availability



Sequence Diagram: Build

3. Draw Message lines between objects

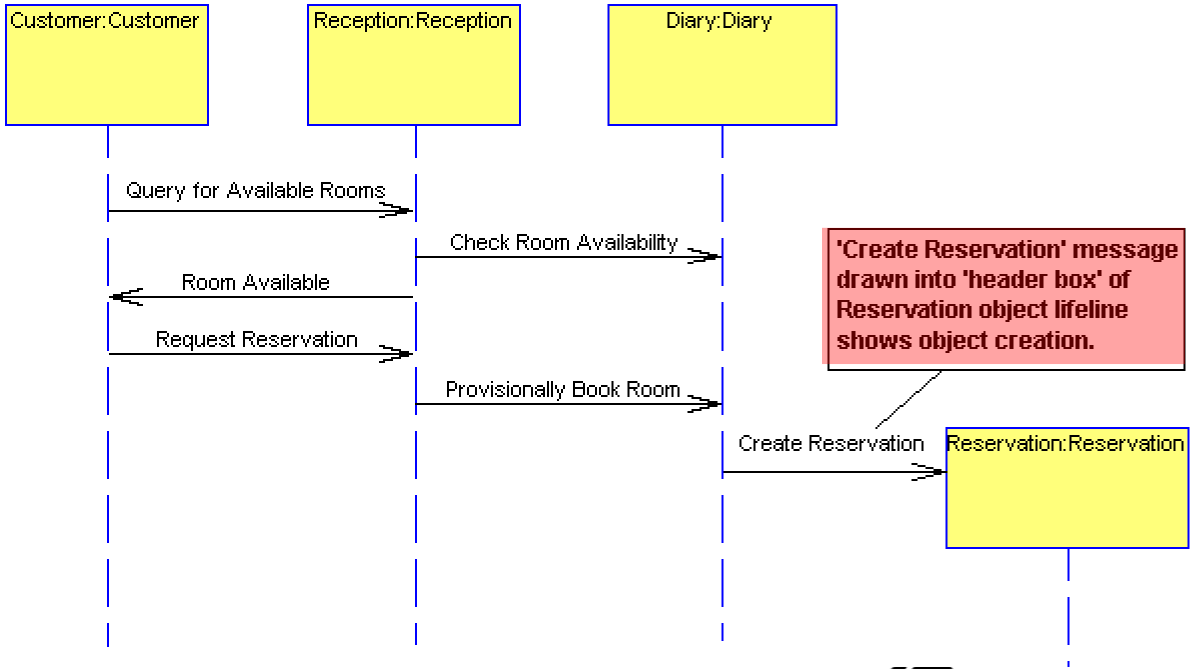
1. Customer Queries for Available Rooms
2. Store Customer Details
3. Check Diary for Room Availability
4. Room is Available
5. Advise Customer of Availability
6. Customer Requests Reservation
7. Provisionally Book Room
8. Figure Out Price, Advise Customer
9. Customer Accepts Terms
10. Provisionally Book Room
11. Check Customer Credit



Sequence Diagram: Build

- Show object creation
  - To show that Reservation is being created in this scenario, move Reservation object downward, so a line is drawn into its header box.

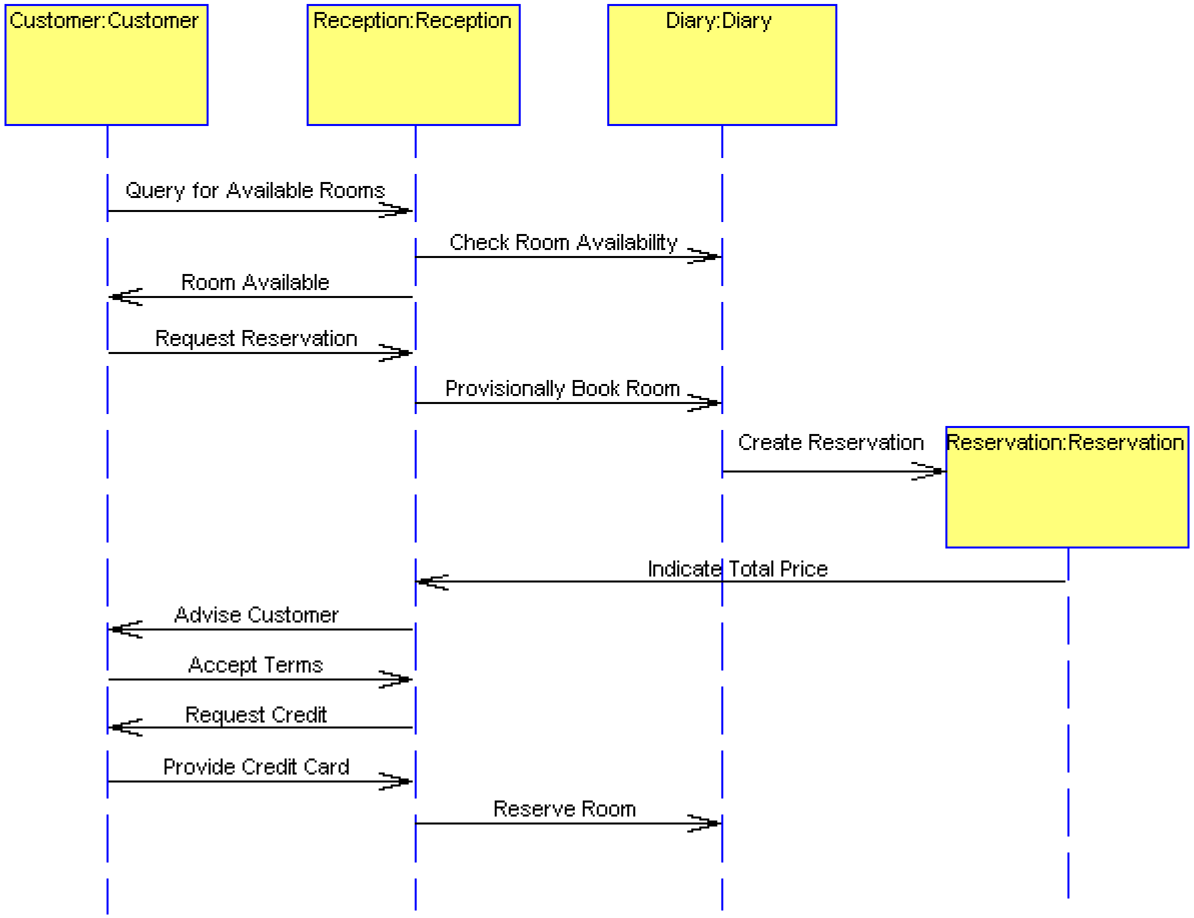
6. Customer Requests Reservation
7. Provisionally Book Room
8. Figure Out Price, Advise Customer
9. Customer Accepts Terms
10. Provisionally Book Room
11. Check Customer Credit
12. Customer Credit is OK
13. Reserve Room



Sequence Diagram: Build

4. Complete the sequence diagram

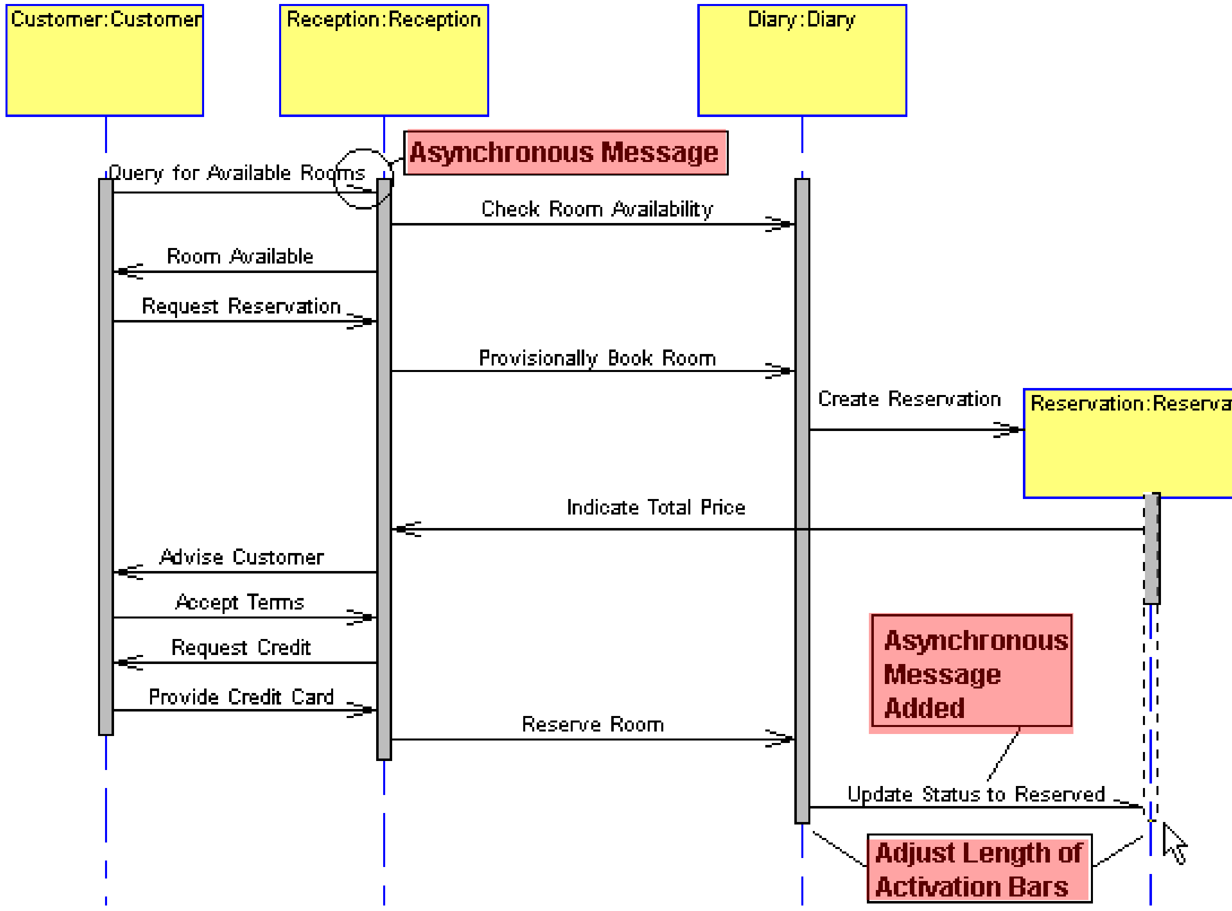
6. Customer Requests Reservation
7. Provisionally Book Room
8. Figure Out Price, Advise Customer
9. Customer Accepts Terms
10. Provisionally Book Room
11. Check Customer Credit
12. Customer Credit is OK
13. Reserve Room



Sequence Diagram: Build

4.1 Add activation bars

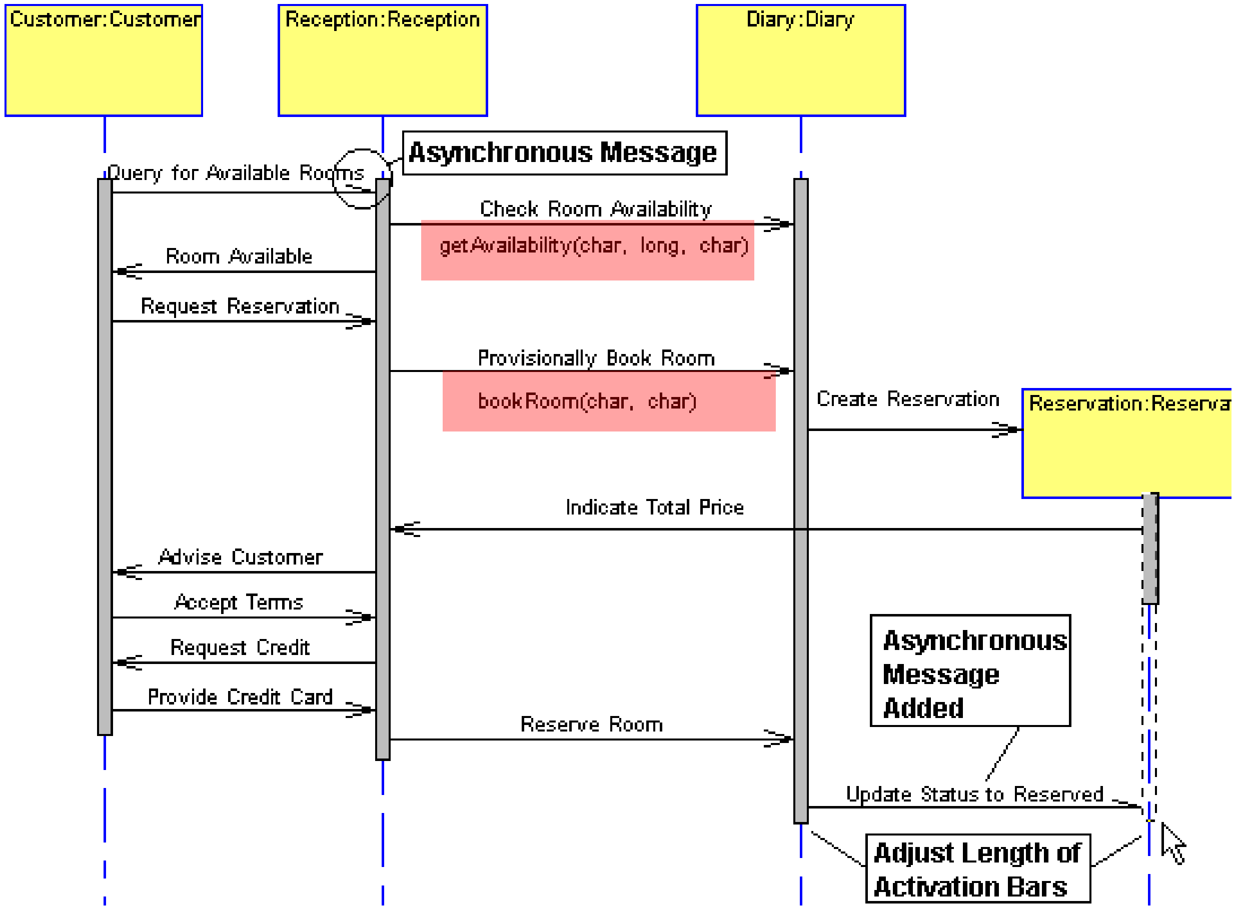
1. Customer Queries for Available Rooms
2. Store Customer Details
3. Check Diary for Room Availability
4. Room is Available
5. Advise Customer of Availability
6. Customer Requests Reservation
7. Provisionally Book Room
8. Figure Out Price, Advise Customer
9. Customer Accepts Terms
10. Provisionally Book Room
11. Check Customer Credit
12. Customer Credit is OK
13. Reserve Room



Sequence Diagram: Build

4.2 Add a method to a message line

- The object receiving the message must be able to perform this task and return an answer to the sending object.
- The Sequence diagram specifies what method is 'invoked' by the sending of a message from one object to another.
- The method invoked belongs to the class of the receiving object.



NEWCASTLE AUSTRALIA

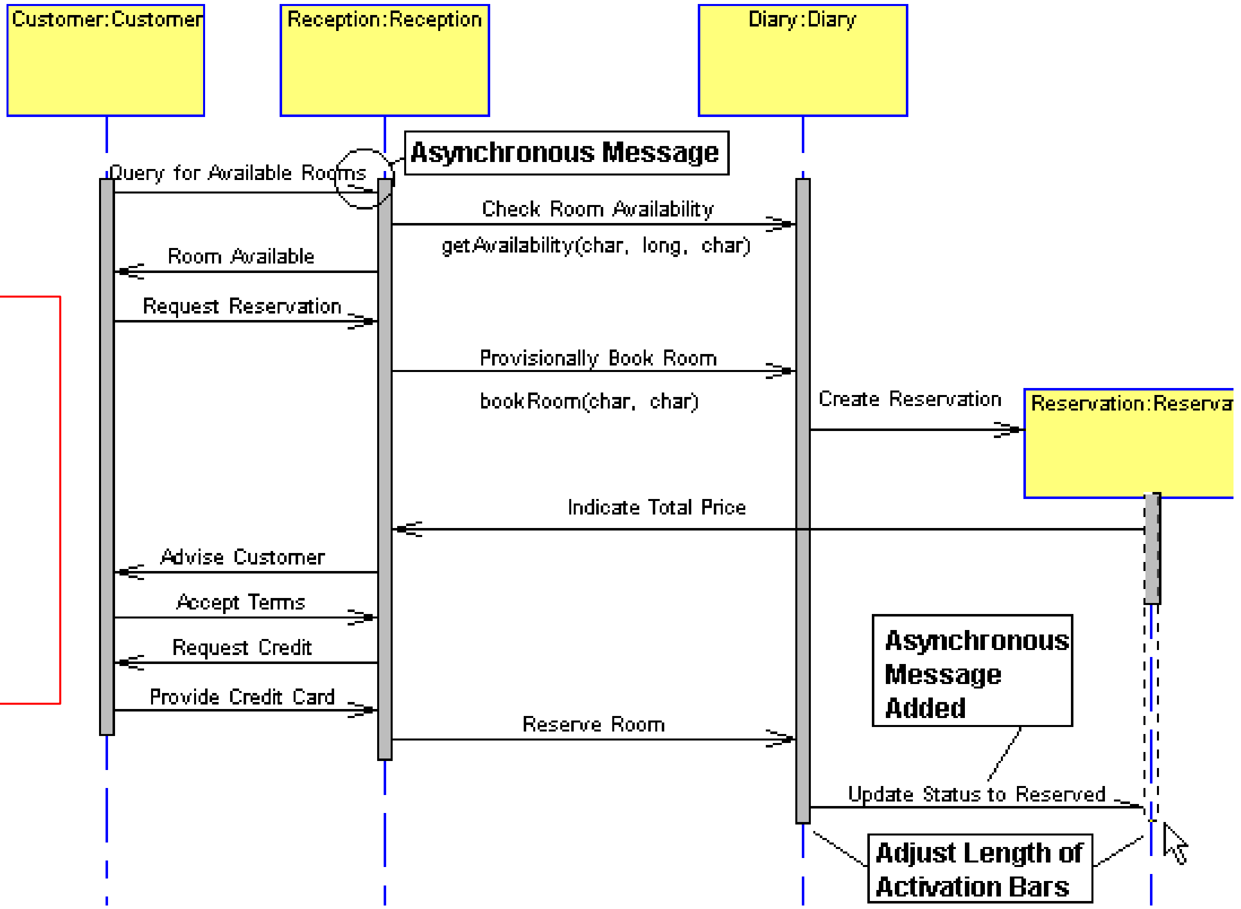
Sequence Diagram: Build

4.2 Add a method to a message line

- The object receiving the message must be able to perform this task and return an answer to the sending object.

MISSING??

The method invoked belongs to the class of the receiving object.

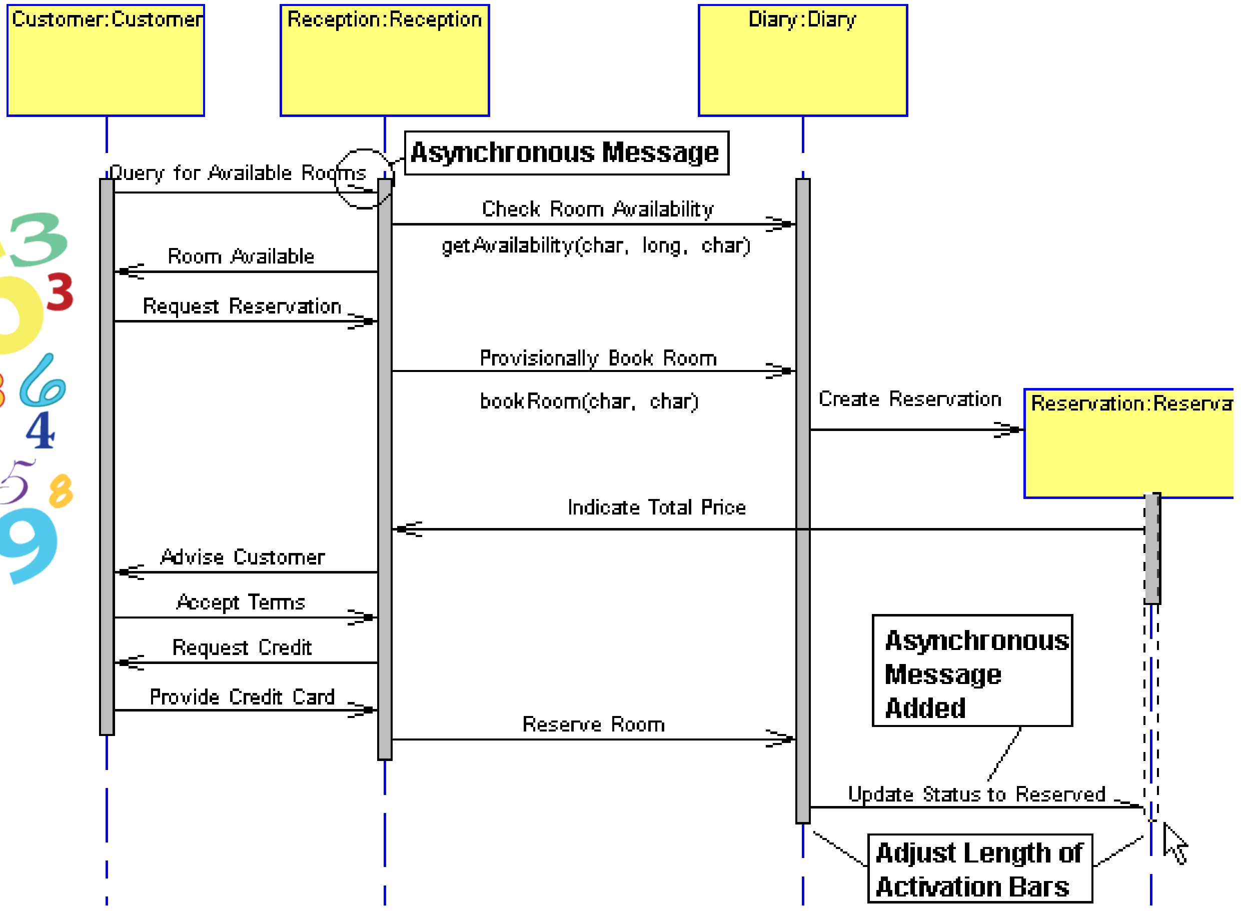


Sequence Diagram: Build

4.2 Add a method to a message line

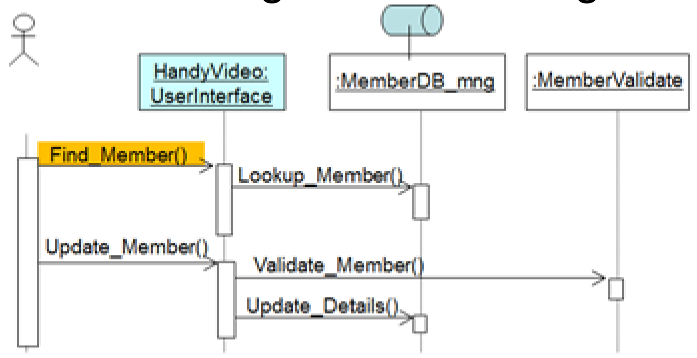
- The object receiving the message must be able to perform this task and return an answer to the sending

- The method invoked belongs to the class of the receiving object.



Summary

- Sequence Diagrams
  - Sequence diagram used to model the dynamic aspects systems
  - A sequence diagram is an interaction diagram that emphasizes the time ordering of the messages



- Combined Fragment
  - Ref, loop, par, opt, alt, break

Summary

- Sequence Diagram: build
  1. Finding objects by examining Use Case Scenarios
  2. Add objects to the sequence diagram
  3. Draw message lines between objects
  4. Complete the sequence diagram
     4.1 Add activation bars
     4.2 Add a method to a message line

Next week

- Analysis