

## **Principios del Diseño de Software:**

**Suficiencia:** El sistema maneja los requerimientos solicitados.

**Entendimiento:** El sistema puede ser entendible o se comprende su uso para la audiencia que lo solicitó

**Modularidad:** El sistema debe de estar dividido en partes bien definidas, pueden considerarse objetos o incluso funciones.

**Cohesión:** El sistema se organiza de tal manera que elementos con las mismas características se agrupan unos con otros.

**Acoplamiento:** El sistema organiza sus partes de tal manera que se minimiza la dependencia entre elementos.

**Robustez:** El sistema puede manejar diversos tipos de entrada.

**Flexibilidad:** En caso que se den cambio en los requerimientos el sistema puede ser modificado fácilmente.

**Reusabilidad:** Partes del sistema pueden ser usadas en otras aplicaciones. Esto abarca tanto al diseño como a la implementación.

**Ocultamiento de la información:** La implementación de los módulos debe de mantenerse oculta de personas ajenas al sistema.

**Eficiencia:** Su ejecución de da en un tiempo aceptable y utilizando ciertos límites en espacio.

**Eficacia (Reliability):** El sistema se ejecuta con una tasa aceptable de fallas.

- Muchos de estos principios se observan también como tácticas dentro de patrones de Arquitectura de Software.

## **Modelo Estructural**

### **1. Diagrama de Clases:**

- En esta parte se recordará algunos conceptos previos adquiridos y relacionados a Programación Orientada a Objetos. Estos principios son utilizados también en el Análisis Orientado a Objetos.
- Considere que este diagramado es una **fase inicial**, esto debido a que pueden existir diversos agujeros en nuestro modelo del sistema. Por ejemplo, aún puede ser difícil establecer el comportamiento o los métodos que permitirán la funcionalidad de los objetos del sistema.

### **1.2 Identificación de Componentes**

a) Se podría hacer un análisis textual de los casos de uso. Por ejemplo se tienen los siguientes lineamientos:

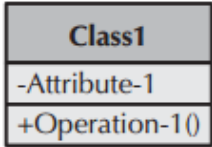
- A common or improper noun implies a class of objects.
- A proper noun or direct reference implies an instance of a class.
- A collective noun implies a class of objects made up of groups of instances of another class.
- An adjective implies an attribute of an object.
- A doing verb implies an operation.
- A being verb implies a classification relationship between an object and its class.
- A having verb implies an aggregation or association relationship.
- A transitive verb implies an operation.
- An intransitive verb implies an exception.
- A predicate or descriptive verb phrase implies an operation.
- An adverb implies an attribute of a relationship or an operation.

Adapted from: These guidelines are based on Russell J. Abbott, "Program Design by Informal English Descriptions," *Communications of the ACM* 26, no. 11 (1983): 882–894; Peter P-S Chen, "English Sentence Structure and Entity-Relationship Diagrams," *Information Sciences: An International Journal* 29, no. 2–3 (1983): 127–149; Ian Graham, *Migrating to Object Technology* (Reading, MA: Addison Wesley Longman, 1995).

b) Tormenta de Ideas: Consiste en discutir sobre las probables clases que formaran parte de un sistema. Por ejemplo, en el caso del sistema de citas un facilitador pregunta a los demás integrantes cuales fueron sus experiencias al separar una cita médica.

c) Patrones: Permite la integración de ciertas prácticas o modelos que han sido exitosos en la mayoría de casos. Se ven en detalle en cursos relacionados a Patrones de Diseño.

### 1.3 Nomenclatura para Diagramas de Clases

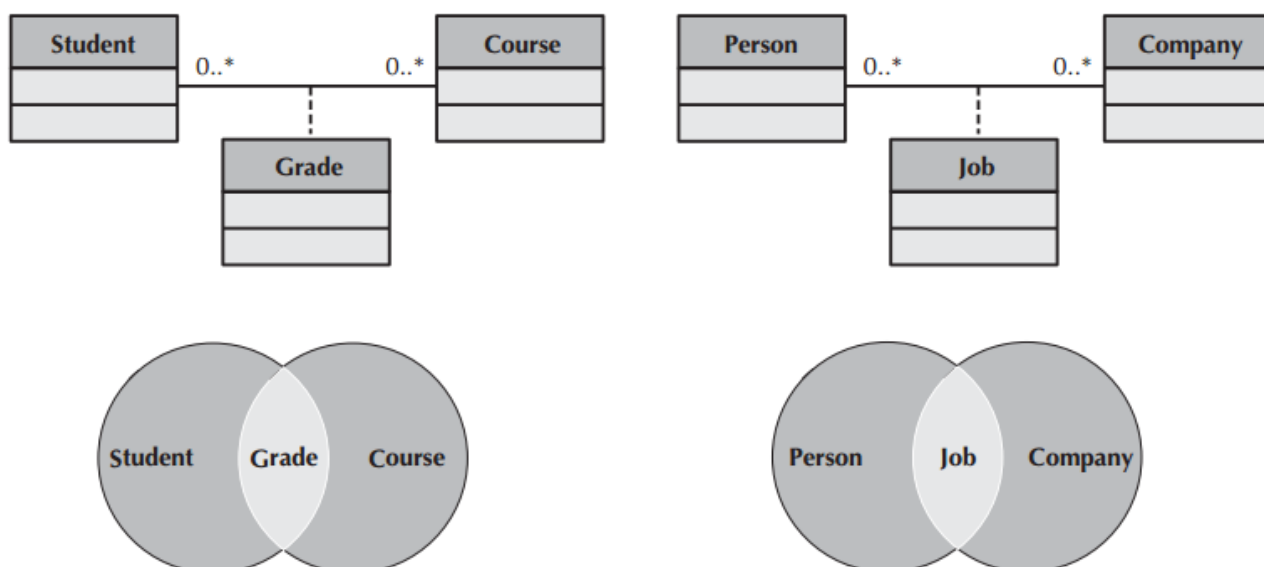
<p><b>A class:</b></p> <ul style="list-style-type: none"> <li>• Represents a kind of person, place, or thing about which the system will need to capture and store information.</li> <li>• Has a name typed in bold and centered in its top compartment.</li> <li>• Has a list of attributes in its middle compartment.</li> <li>• Has a list of operations in its bottom compartment.</li> <li>• Does not explicitly show operations that are available to all classes.</li> </ul>	 <pre> classDiagram     class Class1 {         -Attribute-1         +Operation-1()     } </pre>
<p><b>An attribute:</b></p> <ul style="list-style-type: none"> <li>• Represents properties that describe the state of an object.</li> <li>• Can be derived from other attributes, shown by placing a slash before the attribute's name.</li> </ul>	<p>attribute name /derived attribute name</p>
<p><b>An operation:</b></p> <ul style="list-style-type: none"> <li>• Represents the actions or functions that a class can perform.</li> <li>• Can be classified as a constructor, query, or update operation.</li> <li>• Includes parentheses that may contain parameters or information needed to perform the operation.</li> </ul>	<p>operation name ()</p>

<b>An association:</b> <ul style="list-style-type: none"> <li>Represents a relationship between multiple classes or a class and itself.</li> <li>Is labeled using a verb phrase or a role name, whichever better represents the relationship.</li> <li>Can exist between one or more classes.</li> <li>Contains multiplicity symbols, which represent the minimum and maximum times a class instance can be associated with the related class instance.</li> </ul>	<div style="text-align: center;"> <u>AssociatedWith</u>  0..* ————— 1 </div>
<b>A generalization:</b> <ul style="list-style-type: none"> <li>Represents a kind-of relationship between multiple classes.</li> </ul>	<div style="text-align: center;"> —————&gt; </div>
<b>An aggregation:</b> <ul style="list-style-type: none"> <li>Represents a logical a-part-of relationship between multiple classes or a class and itself.</li> <li>Is a special form of an association.</li> </ul>	<div style="text-align: center;"> 0..* — IsPartOf ▶ 1 ◊ </div>
<b>A composition:</b> <ul style="list-style-type: none"> <li>Represents a physical a-part-of relationship between multiple classes or a class and itself.</li> <li>Is a special form of an association.</li> </ul>	<div style="text-align: center;"> 1..* — IsPartOf ▶ 1 ◆ </div>

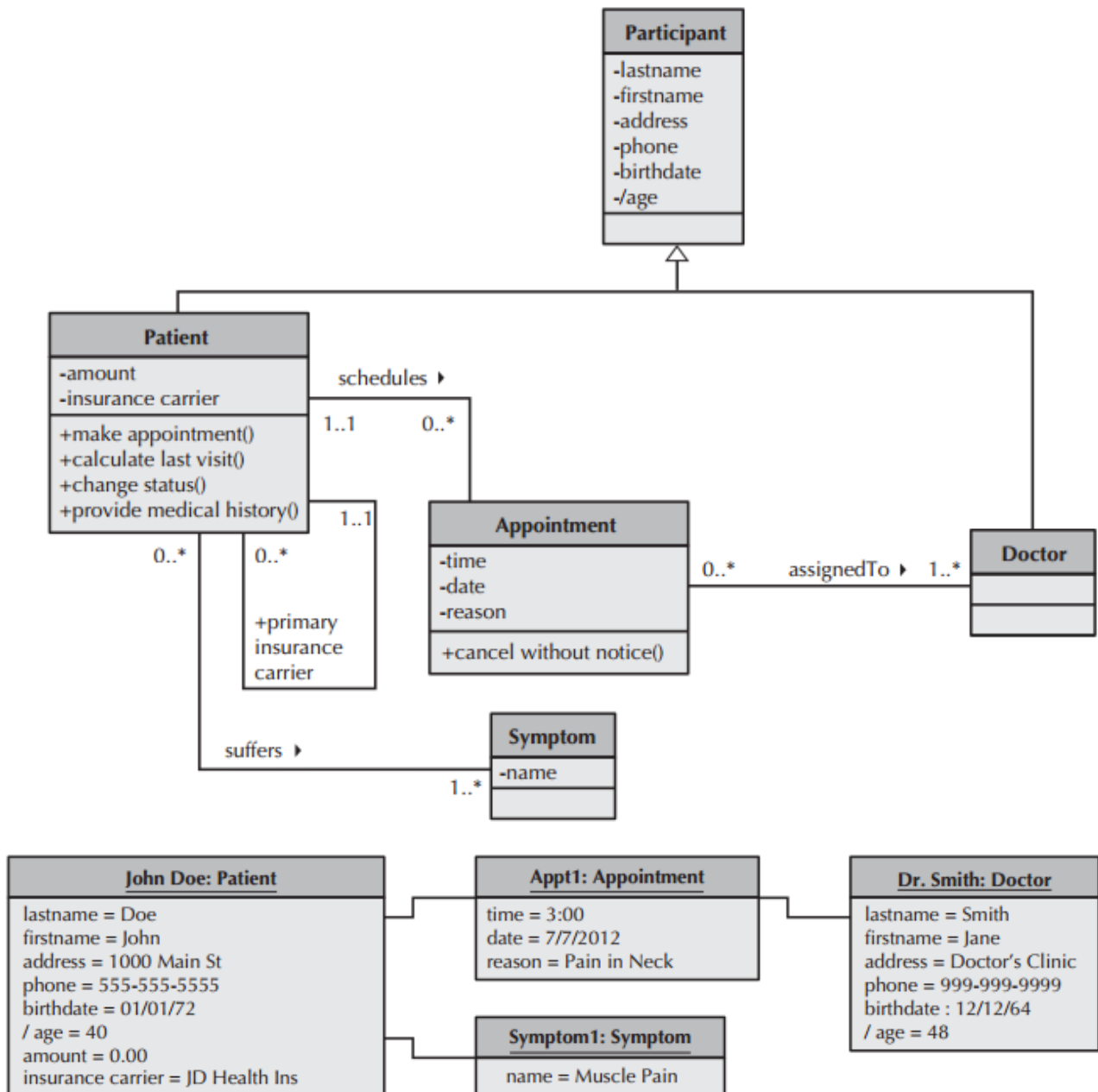
Ejemplo: Ver caso de citas médicas (revisar adjunto)

### 1.3 Casos Especiales

- Relaciones de muchos a muchos, parecido al caso que se ve en el modelo Relacional. Sin embargo, se puede programar en cualquier lenguaje orientado a objetos:



- Diagrama de clases con objetos instanciados



#### 1.4 Tarjetas CRC

- Sus siglas corresponden a Class-Responsability-Collaboration, los cuales sirven para identificar componentes y ver asuntos relacionados al diseño en las etapas iniciales de construcción de un sistema. Por ejemplo si estuviéramos modelando un sistema para reservas de libros su esquema sería el siguiente:

<b>Class</b> Reservations	<b>Collaborators</b> <ul style="list-style-type: none"> <li>• Catalog</li> <li>• User session</li> </ul>
<b>Responsibility</b> <ul style="list-style-type: none"> <li>• Keep list of reserved titles</li> <li>• Handle reservations</li> </ul>	

Para nuestro sistema de pacientes que se vino examinando el esquema de una tarjeta CRC sería el siguiente:

<b>Front:</b>	
<b>Class Name:</b> Old Patient	<b>ID:</b> 3
<b>Type:</b> Concrete, Domain	
<b>Description:</b> An individual who needs to receive or has received medical attention	
<b>Associated Use Cases:</b> 2	
<b>Responsibilities</b> Make appointment Calculate last visit Change status Provide medical history    	<b>Collaborators</b> Appointment   Medical history    
<b>Back:</b>	
<b>Attributes:</b>	
Amount (double)	
Insurance carrier (text)	
<b>Relationships:</b>	
<b>Generalization (a-kind-of):</b>	Person
<b>Aggregation (has-parts):</b>	Medical History
<b>Other Associations:</b>	Appointment

## 2. Robustness Diagram

Es un tipo de diagrama que sirve para agregar funcionalidad a las clases, se podría decir que como enlace entre los casos de uso y los diagramas de secuencia.

Por ejemplo: Caso modelo Iconix:

Step 1: Identify your real-world domain objects (**domain modeling**).

Step 2: Define the behavioral requirements (**use cases**).

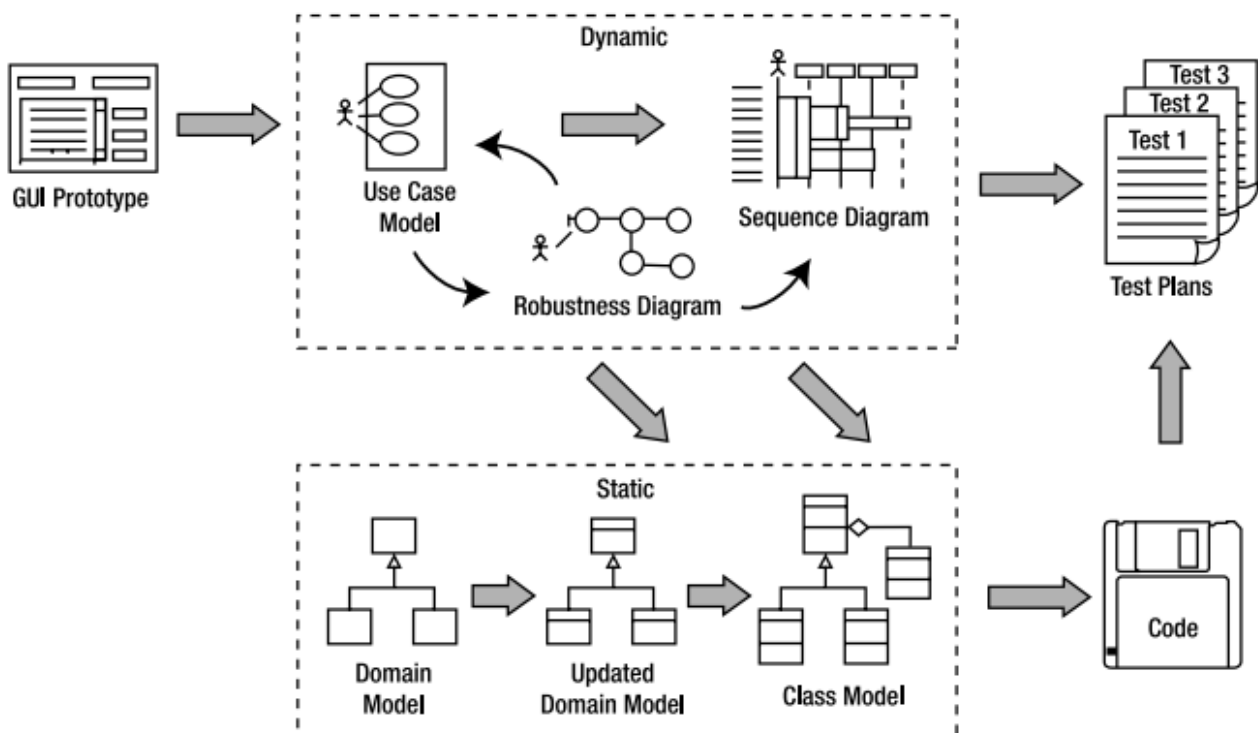
Step 3: Perform **robustness analysis** to disambiguate the use cases and identify gaps in the domain model.

Step 4: Allocate behavior to your objects (**sequence diagrams**).

Step 5: Finish the static model (**class diagram**).

Step 6: Write/generate the code (**source code**).

Step 7: Perform system and user-acceptance testing.

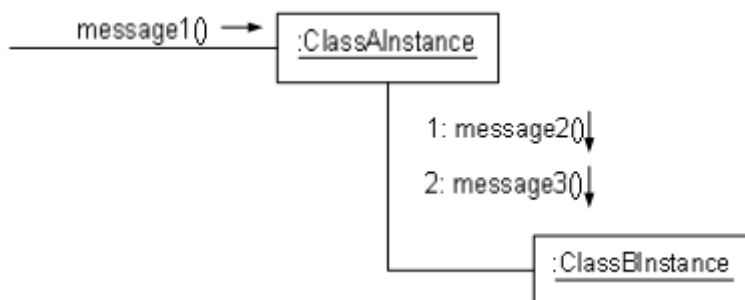


Se ampliará esta sección.

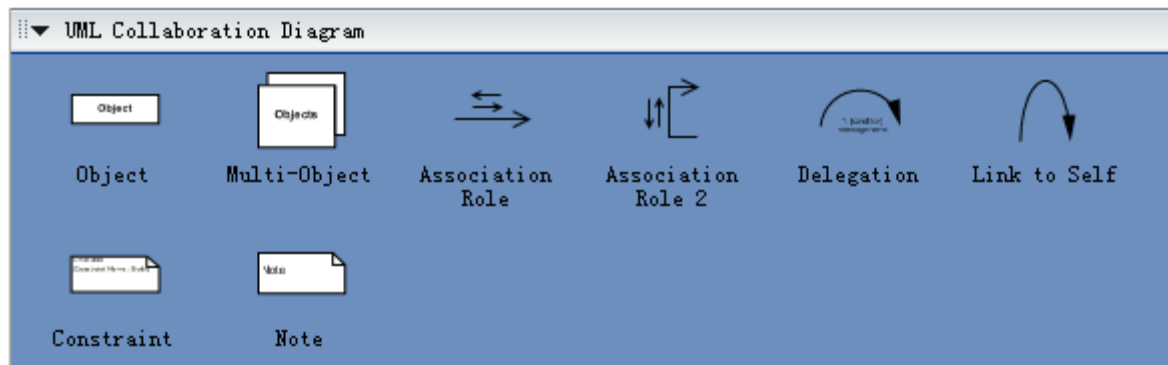
## Modelo de Comportamiento

### 1. Diagrama de Colaboración

- Sirven para representar interacciones entre los distintos objetos de un sistema. Por ejemplo:



- Nomenclatura:



Ejemplo:

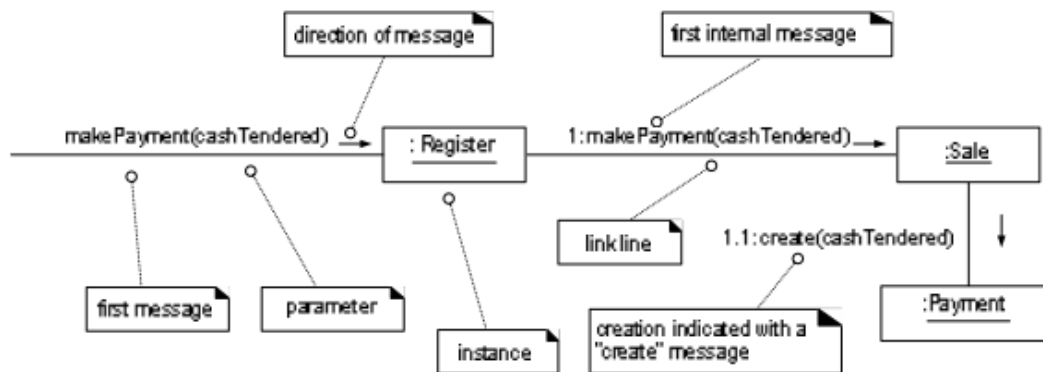
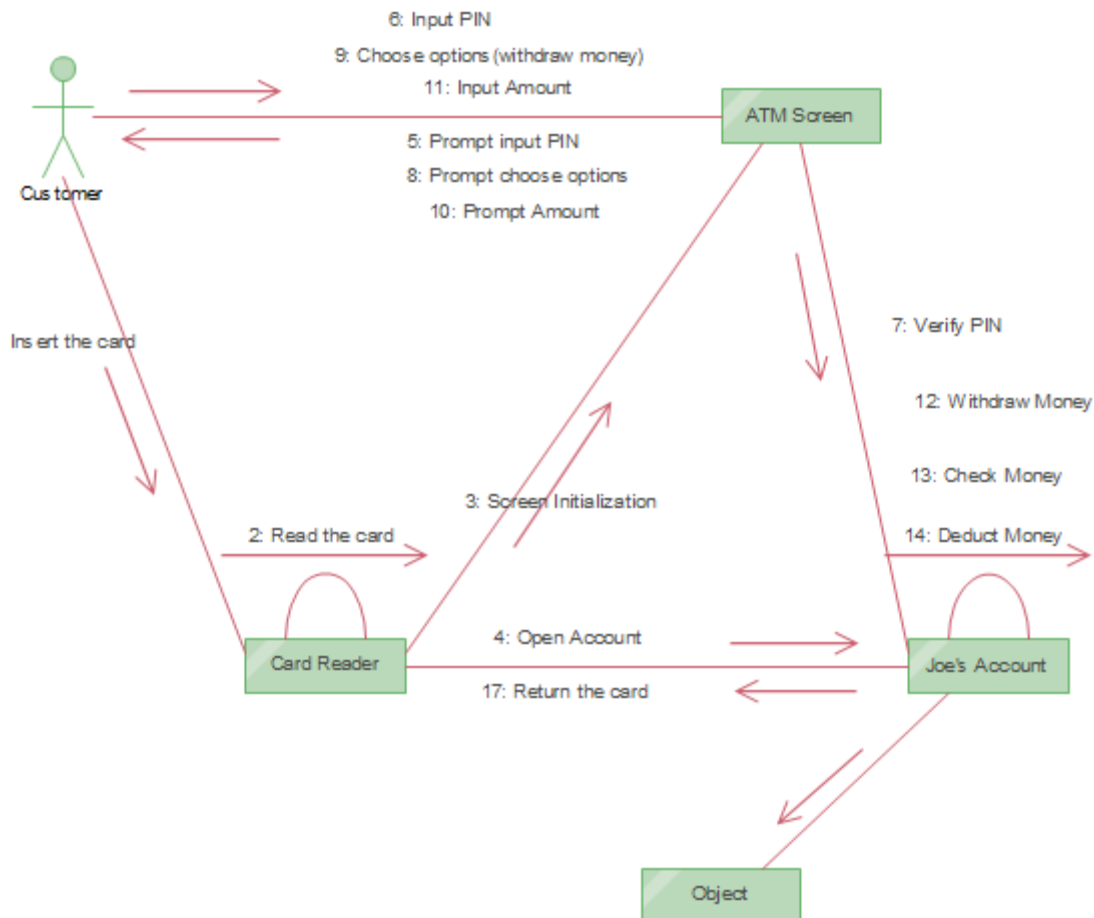


Figure 15.3 Collaboration diagram.

The collaboration diagram shown in Figure 15.3 is read as follows:

1. The message *makePayment* is sent to an instance of a *Register*. The sender is not identified.
2. The *Register* instance sends the *makePayment* message to a *Sale* instance.
3. The *Sale* instance creates an instance of a *Payment*.

Ejemplo: Diagrama de Colaboración para un sistema de ATM





Algunas anotaciones:

- Las anotaciones encima de las flechas son mensajes.
- Se pueden dar casos de loops en las clases.
- La forma representada puede incluir actores como en el caso anterior.

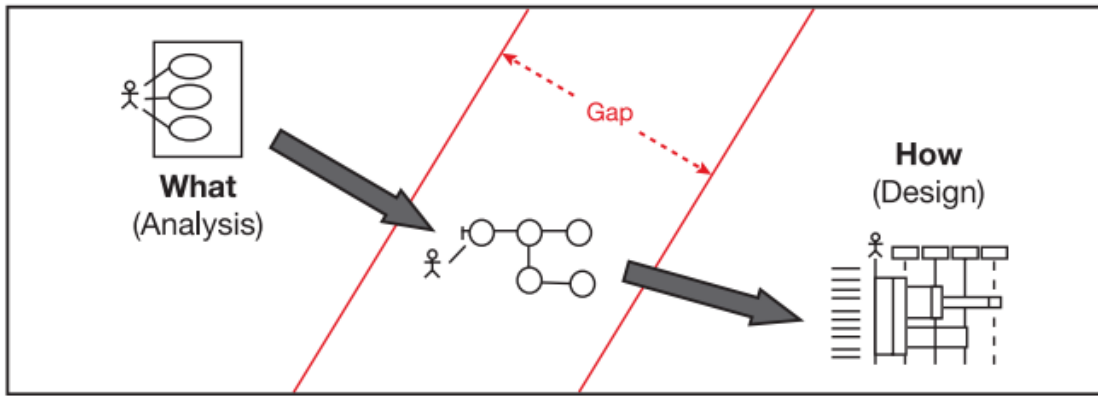
## 2. Diagrama de Robustez

Sería conveniente empezar con la siguiente diferencia:

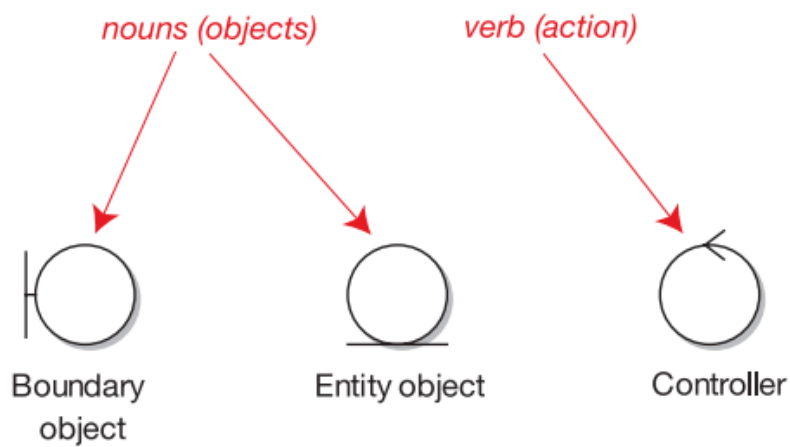
 Use Case	 Algorithm
Dialogue between user and system	"Atomic" computation
Event/response sequence	Series of steps
Basic/alternate courses	One step of a use case
Multiple participating objects	Operation on a class
User and System	All System

Es un tipo de diagrama que nos permite cerrar la brecha entre lo que se desea de un sistema y cómo es que esto se puede llevar a cabo:





Posee los siguientes elementos gráficos:



Estos componentes son los siguientes:

- a) Boundary Object:** Es la interface entre el sistema y el mundo externo, por ejemplo: interfaces de pantallas o páginas web.
- b) Entity Objects:** Son las clases del modelo del dominio.
- c) Controladores:** Permiten la unión entre los objetos boundary y los objetos entidad.

Algunas reglas que se deben de tener en cuenta:

- An Actor can talk to a Boundary Object.
- Boundary Objects and Controllers can talk to each other (Noun <-> Verb).
- A Controller can talk to another Controller (Verb <-> Verb).
- Controllers and Entity Objects can talk to each other (Verb <-> Noun).

Ejemplo:

#### BASIC COURSE:

The user clicks the login link from any of a number of pages; the system displays the login page. The user enters their username and password and clicks Submit. The system checks the master account list to see if the user account exists. If it exists, the system then checks the password. The system retrieves the account information, starts an authenticated session, and redisplay the previous page with a welcome message.

#### ALTERNATE COURSES:

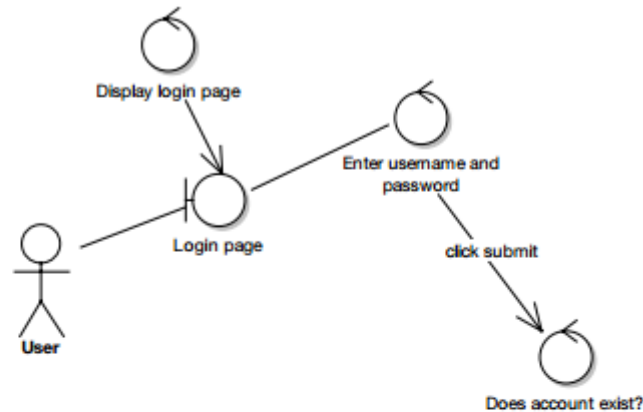
User forgot the password: The user clicks the What's my Password? link. The system prompts the user for their username if not already entered, retrieves the account info, and emails the user their password.

Invalid account: The system displays a message saying that the "username or password" was invalid, and prompts them to reenter it.

Invalid password: The system displays a message that the "username or password" was invalid, and prompts them to reenter it.

User cancels login: The system redisplay the previous page.

Third login failure: The system locks the user's account, so the user must contact Customer Support to reactivate it.



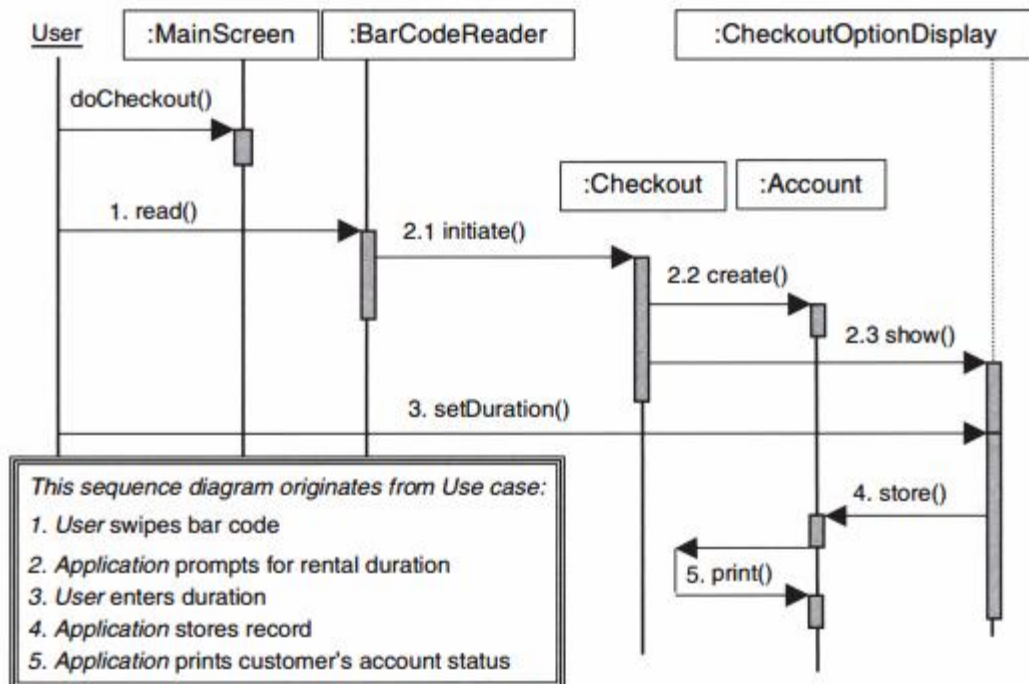
### 3. Diagrama de Actividad:

Con respecto a su trabajo verificar lo siguiente:

- En qué fase del RUP nos encontramos?
- Se han desarrollado diagramas de clases preliminares?
- La parte de determinación de requerimientos debiera estar finalizada
- Revisar la implementación de su propuesta, debe de haber avances.

Los diagramas de secuencia sirven para establecer un conjunto de acciones que se dan entre los distintos objetos que forman parte de un caso de uso. Corresponde a la parte del diagramado de comportamiento o **Behavioral model** del sistema.

Ejemplo: Diagrama de secuencia para el caso de uso de check-out de un sistema de alquiler:



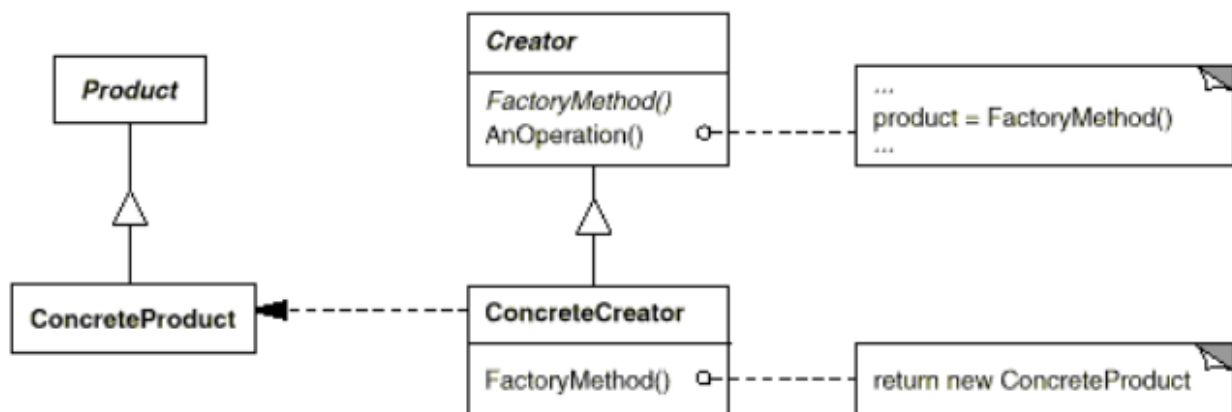
Algunas relaciones con clases y objetos que podríamos determinar del diagrama anterior serían los siguientes:

- El proceso de checkout podría ser realizado al interior de una clase del mismo nombre:
- El objeto BarCodeReader podría crear un objeto de la clase Checkout
- Se puede crear un método del tipo factory para crear o inicializar el objeto Checkout.

**Nota Adicional:** Qué es un método Factory?

Un método factory lo que hace es crear una interface que sirva para la creación de un objeto, lo que varía es que las subclases deciden que objeto desean instanciar.

Gráfica:



Un ejemplo simple:

```

class Pizza():
    def __init__(self):
        self.precio=0

    @property
    def precio(self):
        return self.precio

class Simple(Pizza):
    def __init__(self):
        self.precio=10

class Americana(Pizza):
    def __init__(self):
        self.precio=40

class Margarita(Pizza):
    def __init__(self):
        self.precio=30

class Factory():
    @staticmethod
    def crearPizza(tipo):
        if tipo=="Simple":
            return Simple()
        elif tipo=="Americana":
            return Americana()
        elif tipo=="Margarita":
            return Margarita()

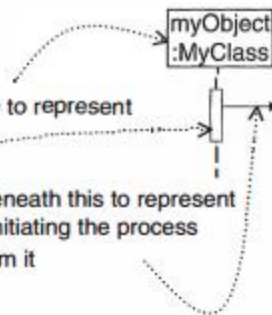
def main():
    fabricaPizzas=Factory()
    print fabricaPizzas.crearPizza("Simple").precio

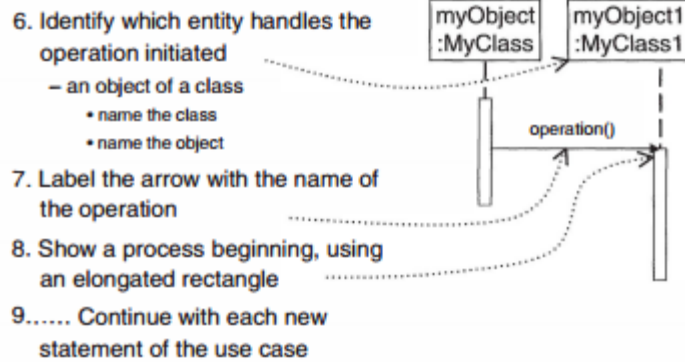
if __name__=="__main__":
    main()

```

Una secuencia de pasos bastante descriptiva para una diagrama de secuencia sería la siguiente:

1. Identify the use case whose sequence diagram you will build (if applicable)
2. Identify which entity initiates the use case
  - the user, or
  - an object of a class
    - name the class
    - name the object, if possible
3. If not the user, draw a rectangle to represent this initiating object at left top
  - use UML *object:Class* notation
4. Draw an elongated rectangle beneath this to represent the execution of the operation initiating the process
5. Draw an arrow pointing right from it





## Referencias

- La parte de Diagrama de Clases y CRC se obtuvieron del libro de Dennis, Wixom y Tegarden titulado “Systems Analysis and Design: An Object-Oriented Approach with UML”
- Una referencia bastante detallada de los temas que se siguen se puede encontrar en el libro de Hans van Vliet titulado “Software Engineering: Principles and Practice”
- El siguiente enlace: <https://www.edrawsoft.com/uml-collaboration.php> da una referencia de diagramas de colaboración bastante general.
- El libro de Craig Larman denominado “Applying UML and Patterns” es una fuente bastante detallada y concisa del uso de UML y cómo se aplica en diversas etapas del Unified Process, el cual es utilizado también en RUP.
- El libro de Doug Rosenberg y Matt Stephens titulado “Use Case Driven Object Modeling with UML” da una referencia sobre la técnica denominada ICONIX de la cual se ha descrito la sección de Robustness Diagram. Esta metodología es de tipo ágil y vendría a ser como un subconjunto de RUP.