**2. Operación send() de un cliente de correo**

1. **Extender *send()* con cada posible acción en una rama de instrucciones de selección. Discute inconvenientes:**

Hay varios inconvenientes por los cuales no deberíamos realizar dicho enfoque del problema. Uno de ellos es que hace los clientes mucho más largos y grandes y más complicados de mantener. Otro inconveniente es que, al ser más grandes los clientes, complica introducir nuevas características o modificar las que ya había anteriormente, además de que no es posible reutilizar los algoritmos. Por último, no queremos mantener algoritmos que no van a ser utilizados.

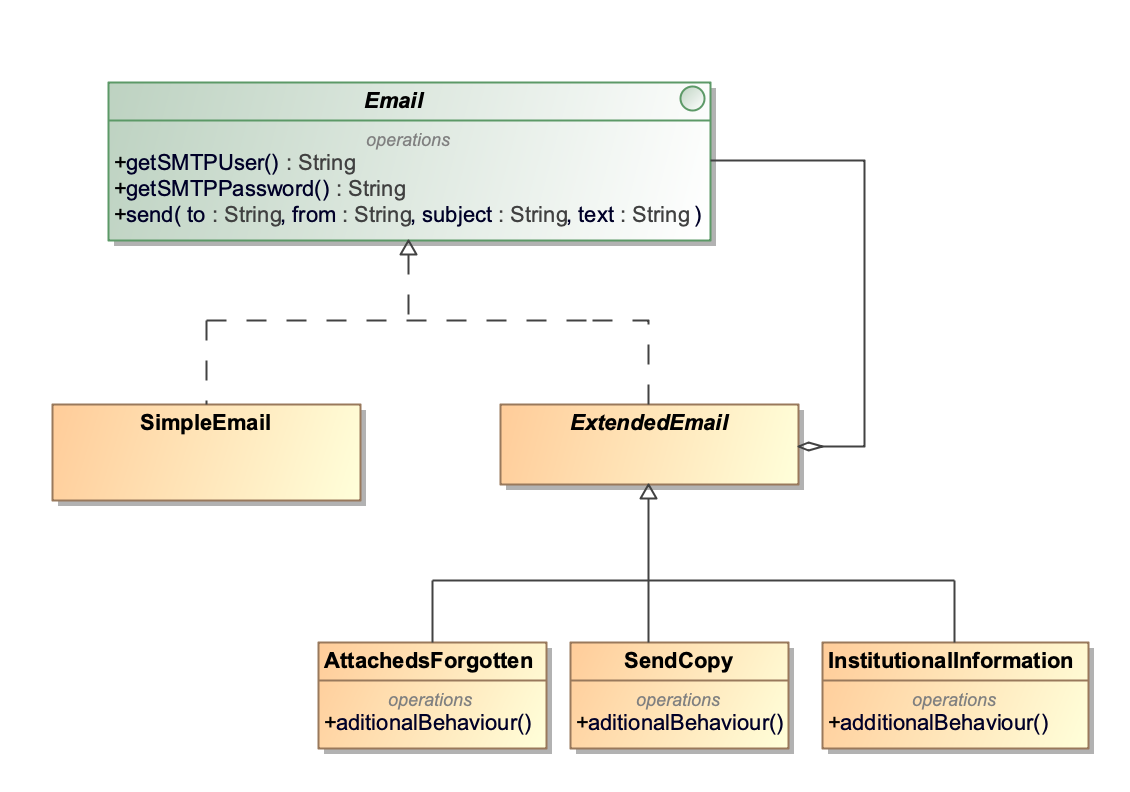
1. **Otra posible solución definir una especialización de la clase Email para cada una de las posibles características. ¿Qué ventajas tendría esta solución? ¿Inconvenientes?**

El problema de definir una especialización para cada característica es que sólo podría seleccionarse una única característica de entre todas ellas, y puede interesarnos tener más de una característica activada. La ventaja respecto a la anterior propuesta es que es más extensible y mantenible, a la hora de añadir nuevas características, bastaría con heredar de la clase Email e implementar la nueva funcionalidad. Otro inconveniente que tiene esta solución es que crea muchos pequeños objetos dentro de nuestro sistema.

1. **Describir un patrón de diseño, incluyendo UML, pseudocódigo y explicaciones textuales, que resuelvan adecuadamente la situación.**

Puesto que en nuestro problema se nos pide poder usar más de una característica a la vez, y puesto que hacer subclases directamente de Email es inviable, hemos decidido utilizar el patrón Decorador, que nos permite tener funcionalidades distintas y “encadenadas” que permiten al usuario usar todas las características que desee.

Diagrama UML de la solución:



Podemos observar como hemos convertido la clase *Email* en una interfaz que será nuestro Componente a extender. La clase SimpleEmail implementa el Email sin ningún comportamiento añadido, mientras que el ExtendedEmail nos permite crear subclases que hereden de la clase abstracta para poder añadir comportamiento adicional a nuestro Email. Si por algún motivo nuestro sistema quisiera ser ampliado, símplemente haría falta añadir nuevas subclases de ExtendedEmail que implementen nuevas

Pseudocódigo java:

public interface Email {

String getSMTPUser();

String getSMTPPassword();

void send(String to, String from, String subject, String text);

}

public class SimpleEmail implements Email {

// implementación concreta de la funcionalidad de los métodos

String getSMTPUser() { return “”; }

String getSMTPPassword() { return “”; }

void send(String to, String from, String subject, String text) { }

}

public abstract class ExtendedEmail implements Email {

private Email extendingEmailBehaviour;

public ExtendedEmail(Email toExtend) {

this.extendingEmailBehaviour = toExtend;

}

public String getSMTPUser() { return “”; }

public String getSMTPPassword() { return “”; }

public void send(String to, String from, String subject, String text) {

extendingEmailBehaviour.send(to, from, subject, text); }

}

// El resto de clases se implementan de la misma manera

// No hemos puesto una implementación concreta de las características adicionales, pero

// El patrón sigue este esquema

public class AttachedForgotten extends ExtendedEmail {

public AttachedForgotten(Email toExtend) {

super(toExtend);

}

private void additionalBehaviour() {

}

public void send(String to, String from, String subject, String text) {

additionalBehaviour();

super.send(to, from, subject, text);

}