1. (20 pts) Dos de los patrones básicos muy utilizados que estudiamos en clases son Fábrica (Factory) y Fábrica Abstracta (Abstract Factory). Respecto a ellos conteste las siguientes preguntas en la forma mas corta y precisa posible:

   a) Por qué se necesita "Factory" cuando podríamos simplemente usar constructores

   b) Muestre un ejemplo concreto de "Factory" (diagrama de clases y breve explicación) distinto al que se vio en clases

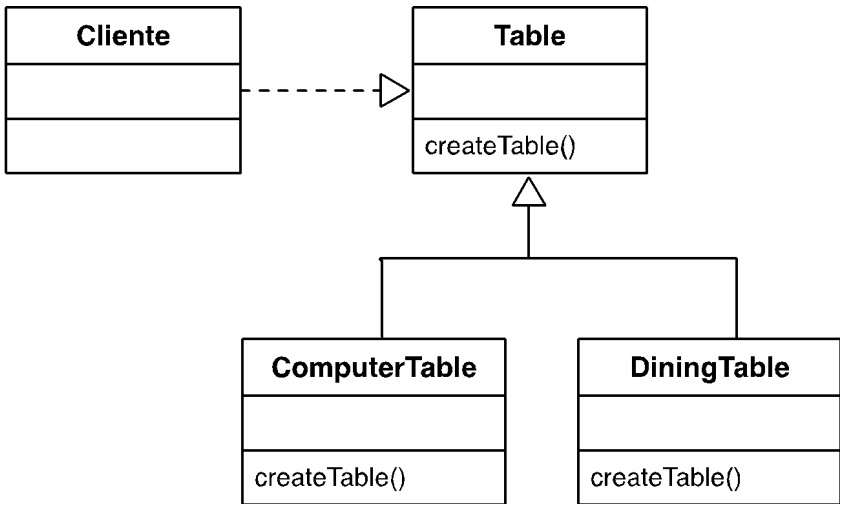
   c) Por qué se necesita "Abstract Factory" y cual es la diferencia con "Factory"

   d) Muestre un ejemplo concreto de "Abstract Factory" (diagrama de clases y breve explicación) distinto al que se vio en clases

---

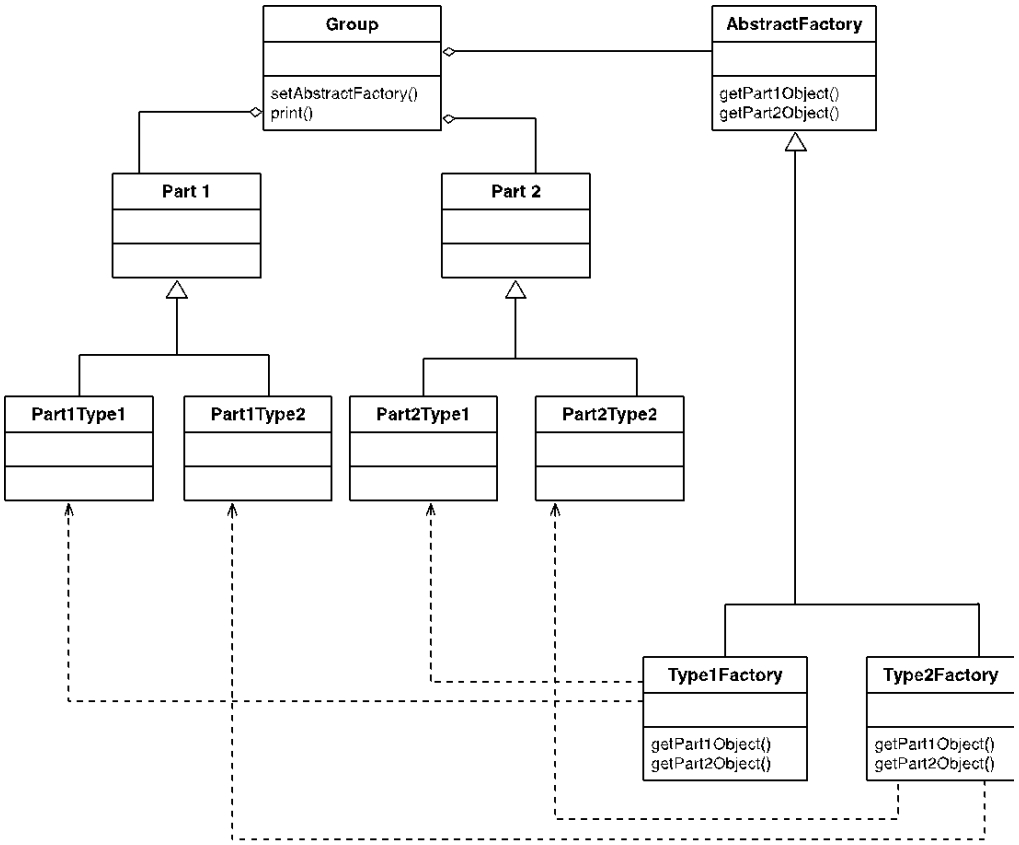
a) Los constructores son inadecuados cuando se requiere crear objetos de subclases que son determinados en runtime.

b) Supongamos que hay una clase Table con subclases ComputerTable y DiningTable. El método createTable produce una nueva Table de acuerdo a lo que se necesite.



c) El propósito de "Abstract Factory" es proveer una interfaz para crear familias de objetos relacionados en runtime sin especificar los objetos concretos. Esto se logra creando una clase para esta Abstract Factory que contiene una operación para cada clase en la familia

d) En la figura el cliente invoca al Grupo (objetos Part1 y Part2) para imprimir objetos Part1Type1. Para ello usa la clase AbstractFactory para invocar la operación virtual getPart1 que en realidad va a corresponder a getPart1 de Type1Factory.



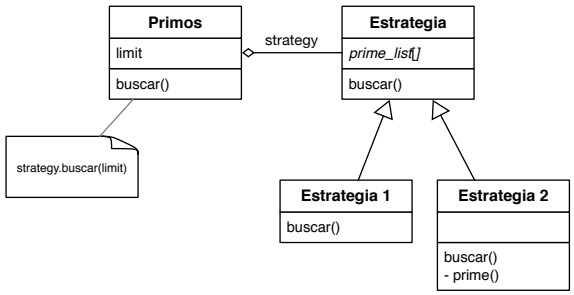
2. (30 pts) Se quiere escribir una pieza de código que sirva para calcular números primos sacando partido del conocido patrón de diseño Estrategia (Strategy). La idea es que podamos tener un algoritmo de cálculo separable del resto, de modo de poder calcular los númerus primos menores a 30 con a lo menos los siguientes dos algoritmos:
   - algoritmo hard_coded – los primeros números primos están guardados en forma estática en una lista
   - algoritmo standard – se verifica si el número es divisible por cada uno de los números comenzando con 2.

   La idea es poder hacer algo como lo siguiente:

```
primos = Primos.new(Estrategia1.new)
primos.buscar
primos = Primos.new(Estrategia2.new)
primos.buscar
```

   a) (10 pts) Describa usando UML la solución de este problema usando el patrón Strategy.
   b) (20 pts) Escriba el código completo de la implementación en Ruby para que funcione como en el ejemplo

a)



b)

```
class Primos
  attr_reader :limit
  attr_accessor :strategy

  def initialize (strategy)
    @limit = 30
    @strategy = strategy
  end

  def buscar
    @strategy.buscar(@limit)
  end
end
```

```
class Estrategia
  @@prime_list = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,109, 113]
end

class Estrategia1 < Estrategia

  def buscar (limit)
    i = 0
    while i < limit
      puts @@prime_list[i]
      i += 1
    end
  end

end

class Estrategia2 < Estrategia

  def buscar (limit)
    i = 0
    n = 2
    while i < limit
      if prime(n)
        puts n
        i += 1
      end
      n += 1
    end
  end

  private

  def prime (n)
    success = true
    from = 2
    to = (Math.sqrt n).floor
    divisor = from
    while divisor <= to
      if n/divisor*divisor == n
        success = false
        break
      end
      divisor += 1
    end
    return success
  end

end
```

3) Considera una aplicación que permita a los clientes de un banco realizar todo tipo de transacciones —desde consultas de saldo y estados de cuenta, pasando por compras tradicionales e inversiones en la bolsa, y hasta el pago de multas e incluso de los impuestos—, usando cualquier tipo de dispositivo conectable a Internet —desde un computador personal, pasando por una *tablet*, y hasta un teléfono celular o un reloj. Por supuesto, esta debe ser una aplicación que tenga toda la funcionalidad necesaria y además ser fácil de usar, y, dada su naturaleza, también debe ser muy confiable y tener excelente desempeño.

Con respecto a la actividad de análisis arquitectónico durante el desarrollo de esta aplicación, responde:

a) Identifica, y justifica que es así, dos puntos de variación.

Todas las interfaces con sistemas de terceros, sobre los cuales la aplicación no tiene ningún control: p.ej., las APIs de (los sitios Web de) los negocios a los cuales se tiene acceso a través de la aplicación; sistemas validadores de formas de pago

b) Identifica, y justifica que es así, dos puntos de evolución.

Servicios que aún no están incluidos (y que a lo mejor son de una naturaleza diferente); p.ej., si queremos que la aplicación funcione fuera de Chile, entonces habrá que preocuparse de cuáles facilidades / dificultades existen en otros países para pagar los impuestos por Internet; o el servicio de poder conectarse a nuevos servicios.

Dispositivos de I/O aún no disponibles.

c) Define dos escenarios de calidad, más allá de los obvios sobre desempeño y tiempo medio entre fallas.

El 90% de nuestros clientes declara estar 90% satisfecho con la usabilidad de la aplicación (encuestas mensuales a partir de los 3 meses de funcionamiento).

El 95% de los problemas técnicos reportados tanto por los clientes como por los operadores del servicio deben ser resueltos en un plazo máximo de 6 horas.

La aplicación debe estar disponible el 99% del tiempo (aprox., 360 días al año).

d) Con respecto a los patrones arquitectónicos *cliente-servidor* y *peer-to-peer*, ¿cuál es el más apropiado en este caso y por qué?

Cliente-servidor: Es una aplicación distribuida, con un módulo central servidor —que soporta todos los servicios mencionados— y un módulo cliente —que interactúa directamente con los clientes del banco y está replicado en cada dispositivo de estos clientes— cuyos roles son muy diferentes y no tiene sentido técnico tratar de hacerlos similares; p.ej., los módulos clientes no ofrecen servicios ni interactúan entre ellos; el módulo servidor no solicita servicios a los clientes. En cambio, podemos aprovechar las diferencias y especializar cada uno de estos dos módulos para hacerlos muy eficientes en sus respectivos roles.

4) Recuerda la aplicación para jugar el juego "The Trivium". Queremos ofrecer esta aplicación bajo la modalidad *software-as-a-service*, de manera que los jugadores puedan estar geográficamente distribui-dos (p.ej., uno en Santiago, otro en Arica, otro en Buenos Aires, etc.,) y acceder al juego a través de la Internet empleando un *browser*. Considera el lado del servidor de esta aplicación.

a) ¿Cuáles aspectos/funcionalidades/responsabilidades de la aplicación pertenecen al servidor y por qué? Sé tan específico como puedas con respecto al juego "The Trivium".

Llevar el control del juego: ¿qué jugador tiene el turno? ¿en qué casilla está y cuántos puntos tiene cada jugador? ¿cuál fue el resultado del lanzamiento del dado? ¿qué pregunta se le está haciendo al jugador con el turno y cuál es la respuesta correcta?

b) ¿Qué patrón arquitectónico es recomendable para el servidor y por qué? Enuncia —da su nombre y descríbelo de manera breve y precisa— un criterio o principio de diseño que guíe esta recomendación.

El patrón *Arquitectura de tres capas* (lógicas): presentación, lógica, y persistencia (base de datos).

El criterio de *separación de intereses* (o responsabilidades).

c) ¿Cuáles aspectos/funcionalidades/responsabilidades del servidor, identificados en a), quedan mejor ubicados en cada uno de los componentes del patrón arquitectónico identificado en b)?

La responsabilidad de llevar el control de juego, especificado en a), es de la capa lógica.

La responsabilidad de recibir los inputs hechos por los jugadores es de la capa de presentación (un servidor Web).

La responsabilidad de mantener toda la información a los largo del tiempo es de la capa de persistencia.

d) Queremos organizar el código de la aplicación propiamente tal siguiendo el patrón arquitectónico *modelo-vista-controlador* (MVC). ¿Por qué queremos hacer esto? Responde de manera precisa.

- Definición cohesiva del modelo, focalizada en los procesos del dominio
- Desarrollo independiente de las componentes del modelo y de la UI
- Reducción del impacto de cambios en los requisitos de la UI sobre la componente del dominio
- Fácil conexión de nuevas vistas a la componente del dominio
- Posibilidad de múltiples vistas simultáneas sobre el mismo objeto del modelo
- Ejecución de la componente del modelo independiente de la componente de la UI
- Portabilidad de la componente del modelo a otro *framework* para UI

e) Explica el rol que desempeña cada componente del patrón MVC, pero no en general, sino que particularmente durante la ejecución del caso de uso *Jugar*:

1. El Jugador que tiene el turno lanza el dado
2. El Juego muestra el resultado del lanzamiento del dado
3. El Juego muestra las casillas a las que el Jugador puede mover su ficha
4. El Jugador mueve su ficha a una de esas casillas
5. El Juego hace una pregunta correspondiente al tema de la casilla
6. El Jugador (responde y luego) pide al Juego que muestre la respuesta correcta
7. El Juego muestra la respuesta correcta
8. El Jugador indica que su respuesta ha sido incorrecta
9. El Juego pasa el turno a otro jugador

En los pasos 1, 4, 6 y 8, la Vista recibe inputs del jugador y se los envía al Controlador. En cada caso, el Controlador interpreta el input y le pide al Modelo que cambie de estado: en 1, cambia el valor actual del dado; en 4, cambia la casilla en la que está el jugador; en 6, no hay cambio de estado; y en 8, cambia el jugador que tiene el turno.
En los pasos 1, 4 y 9, el Modelo notifica a la Vista su cambio de estado para que la Vista se actualice; p.ej., el paso 2.
Los pasos 3, 5 y 7 no implican un cambio de estado, por lo que el Controlador, a partir de información obtenida desde el Modelo, puede instruir directamente a la Vista para que ésta muestre la información correspondiente: las casillas a la que el jugador puede mover su ficha, la pregunta, y la respuesta correcta.

f) ¿Qué patrón (o patrones) de diseño (más detallado) son normalmente aplicados en cada componente del patrón MVC?

Los tres componentes están relacionados mediante el patrón *Observador*: el controlador y la vista son observadores del modelo.

Vista y controlador están relacionados mediante el patrón *Estrategia*: el controlador es la estrategia para la vista.

Finalmente, la vista se implementa con el patrón *Compuesto*.