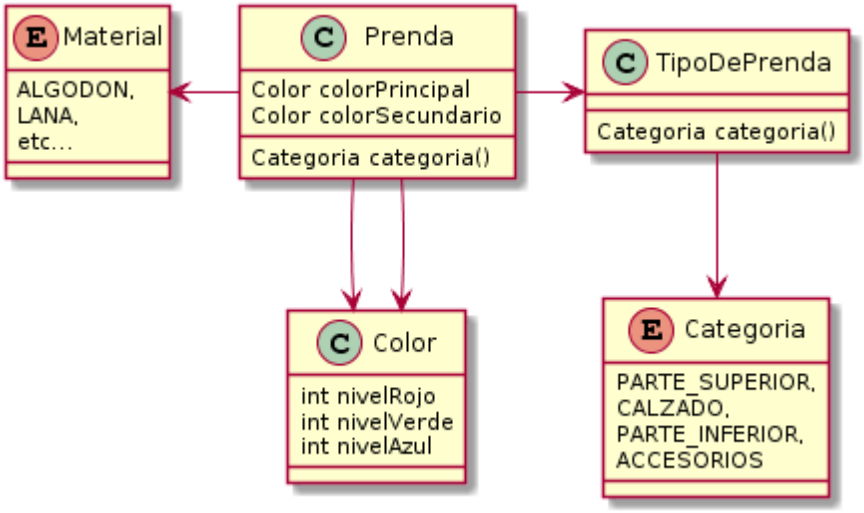
Qué Me Pongo

Alternativas de diseño a la segunda Iteración

En la iteración pasada construimos el modelo inicial de QuéMePongo:



En este modelo todos los objetos son (al menos, por ahora) inmutables:

1. o bien son enumeración *stateless* (sin estado);
2. o bien son objetos cuyos atributos no pueden (porque *no necesitan*) ser cambiados luego de creados.

Esto último también nos llevó a no escribir un sólo setter y a poner lógica de validaciones en el constructor de `Prenda`[1].

Sin embargo, hasta ahora este modelo no resuelve ningún problema muy interesante. En primer lugar, ¿para qué queremos prendas? ¡Ataquemos los siguientes requerimientos!

[1] Lo mismo podríamos hacer con los constructores de `TipoDePrenda` y `Color`.

Primera parte: las tramas

La segunda iteración inicia nuevamente con un requerimiento conocido:

> *Como usuarie de QuéMePongo, quiero poder cargar prendas válidas para generar atuendos con ellas.*

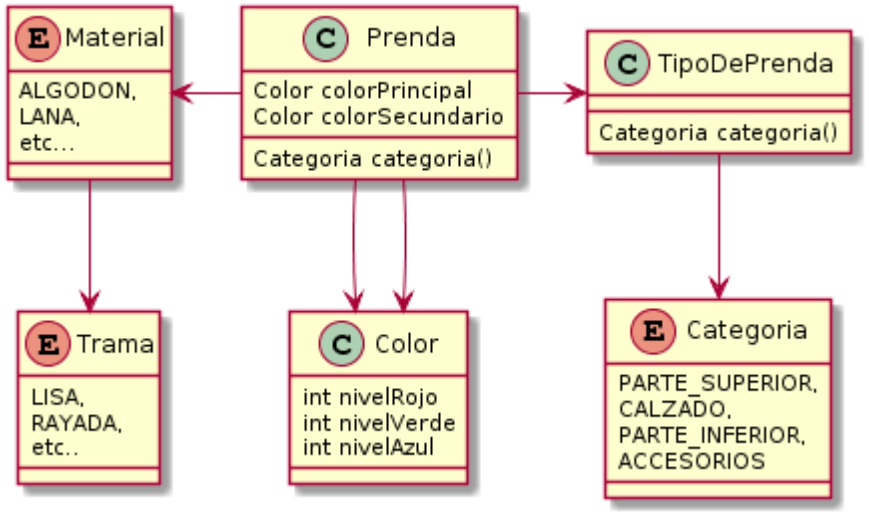
Pero se agregan nuevas dificultades, la primera de ella es el modelado de tramas: *Como usuarie de QuéMePongo, quiero especificar qué trama tiene la tela de una prenda (lisa, rayada, con lunares, a cuadros o un estampado).*

Esto plantea dos problemáticas: cómo modelar la trama y quién y cómo la conocerá:

1. Para lo primero, por ahora podríamos pensarla como un *enum*, dado que los valores parecen enumerables (`LISA`, `RAYADA`, etc.) y no tenemos más información.
2. Para lo segundo, debemos preguntarnos: ¿la trama es una propiedad del material o de la prenda?

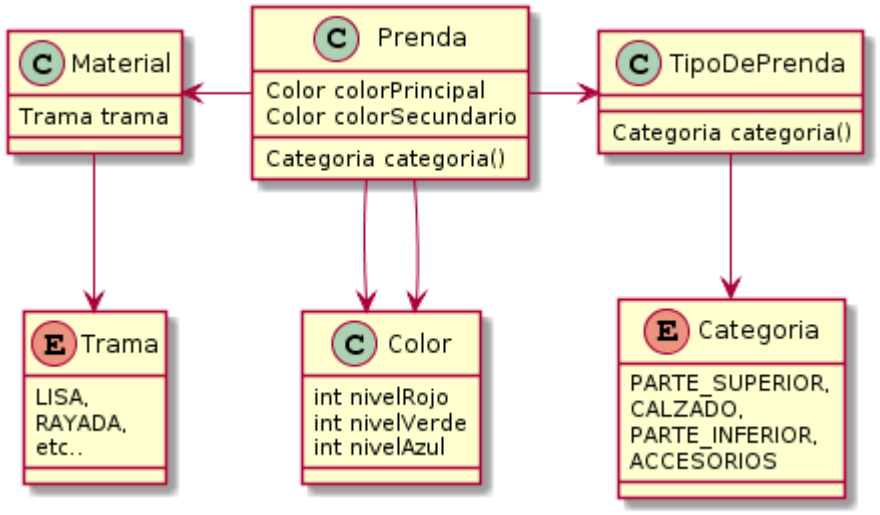
Alternativa 1 : Trama del Material

Analicemos las opciones: por un lado el texto parece sugerirnos que la trama es propiedad de la tela (*material*). Esto coincide con nuestra intuición física sobre los materiales y sus tramas:



Sin embargo, esto no parece funcionar: los materiales son enumeraciones, y a priori deberían ser convertidas en clases[2]:

[2] Técnicamente, como se muestra acá, esto no es cierto, y podríamos mantener esta relación entre material y trama siendo ambas enumeraciones, pero no es algo relevante en este momento.



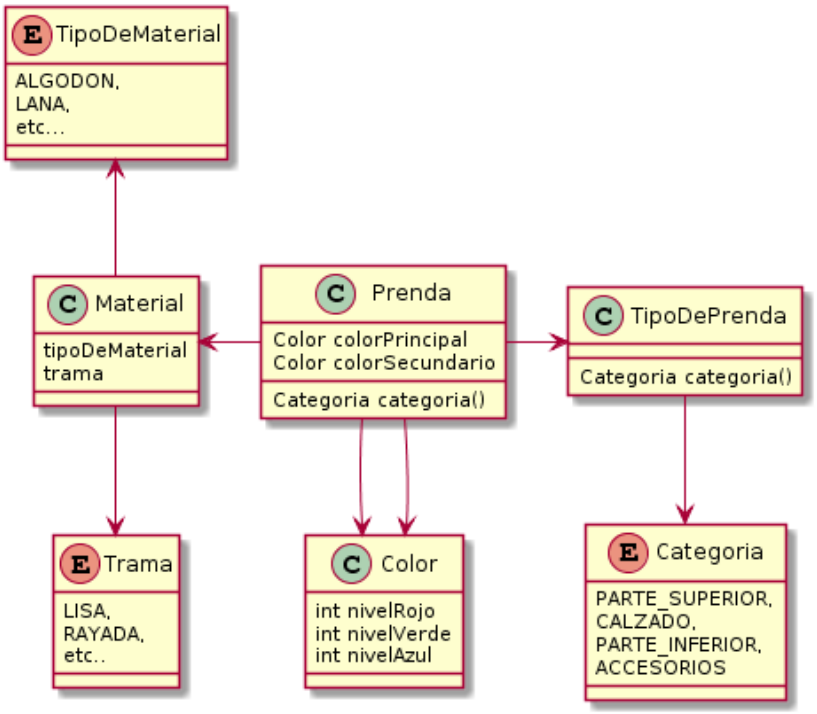

```
class Material
    const Material ALGODON = new Material(...)
    const Material LANA = new Material(...)
    const Material CUERO = new Material(...)
   // etc..
```

Pero aún con este cambio, no sería suficiente: antes contábamos con un objeto único y bien conocido[3] que representaba al algodón, pero ahora vamos a necesitar uno por cada combinación con sus tramas posibles:

```
class Material
    const Material ALGODON_LISO = new Material(Trama.LISA)
    const Material ALGODON_RAYADO = new Material(Trama.RAYADA)
    const Material ALGODON_ESTAMPADO = new Material(Trama.ESTAMPADA)

    const Material LANA_LISA = new Material(Trama.LISA)
    const Material LANA_RAYADA = new Material(Trama.RAYADA)
   // etc..
```

¡Esto es totalmente inmantenible! Quizás, para resolverlo podríamos hacer algo que ya aplicamos anteriormente: diferenciar el concepto de *material* (el ente **real** o sensible, que representa el material como es físicamente) y *tipo de material* (el ente ideal, que representa de forma abstracta a cada uno de los materiales ALGODON, LANA, etc):

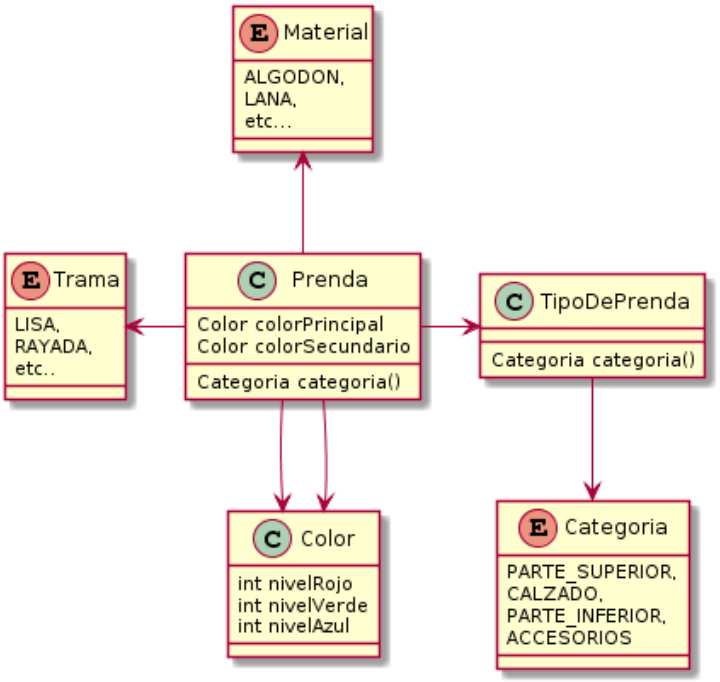
[3] WKO, Well Known Object



De esta manera el `TipoDeMaterial` pasa a tomar el rol que antes tomaba el `Material`. ¿Pero se justifica agregar más complejidad al modelo? ¿Nos aporta realmente algo este enfoque más allá de modelar de forma más cercana al mundo real? Ya el modelo era bastante complejo con prendas y tipos de prendas como para incluir materiales y tipos de material, ¡y aún no resuelve casi nada!

Alternativa 2: Trama de la Prenda

Yendo por un enfoque de favorecer la simplicidad, proponemos entonces volver un paso atrás y hacer que sea la prenda quien conozca a la trama, que al momento actual permite representar exactamente las mismas situaciones:



La única contra es que ahora deberemos agregar un quinto parámetro al constructor de la `Prenda`. En Java esto se vería así:

```
public class Prenda {
  public Prenda(
    TipoDePrenda tipoDePrenda,
    Material material,
    Color colorPrincial,
    Color colorSecundario,
    Trama trama) {
     // ...validaciones...
     this.tipoDePrenda = tipoDePrenda;
     this.material = material;
     this.color = color;
     // etc...
  }
}
```

¡Esto se está volviendo cada vez más engorroso de usar! Pero en breve discutiremos qué podemos hacer al respecto.

Segunda parte: los borradores

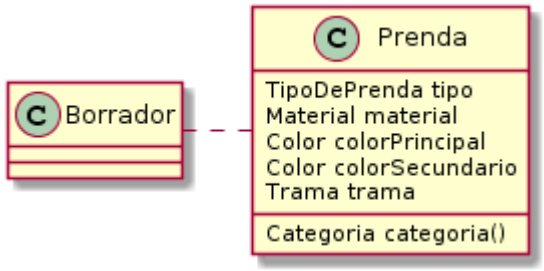
Los siguientes requerimientos introducen un nuevo concepto, los *borradores*:

- *Como usuarie de QuéMePongo, quiero guardar un borrador de la la última prenda que empecé a cargar para continuar después.*
- *Como usuarie de QuéMePongo, quiero poder guardar una prenda solamente si esta es válida.*

Hasta ahora, toda nuestra energía estuvo puesta en tener un único punto para crear la prenda, de forma **atómica**: la prenda o bien se crea con todas sus dependencias válidas y satisfechas, o bien no se crea. Pero este requerimiento parece pedir todo lo contrario. ¿Todo lo que pensamos estuvo mal?

Además ambos requerimientos parecen contradictorios entre sí: por un lado queremos que la prenda sólo exista si es válida (lo cual es lo que estuvimos haciendo hasta ahora), pero pero el otro nos piden que podamos guardar versiones intermedias de la misma. ¿Qué podemos hacer?

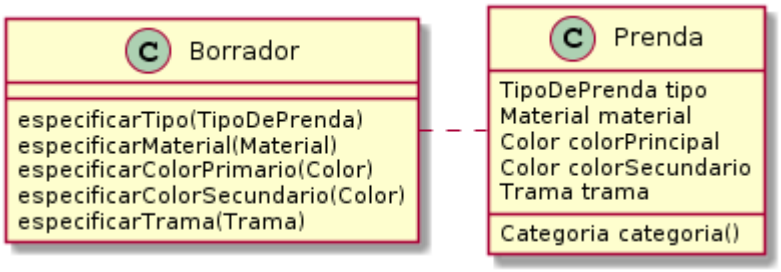
Quizás lo que necesitemos entonces sea tener nuestra prenda, pero por el otro, un objeto que represente al *borrador*:



Mientras que la prenda es inmutable y se crea de forma atómica, el segundo es mutable y se crea en **pasos**, tal como lo insinúan los siguientes requerimientos:

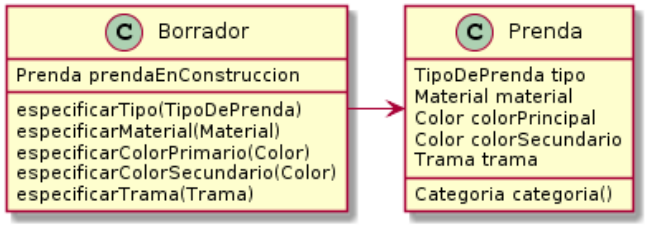
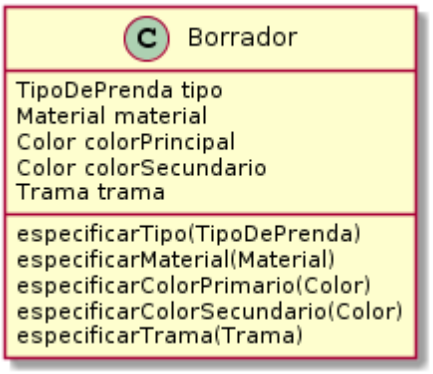
- *Como usuarie de QuéMePongo, quiero crear una prenda especificando primero de qué tipo es.*
- *Como usuarie de QuéMePongo, quiero crear una prenda especificando en segundo lugar los aspectos relacionados a su material (colores, material, trama, etc) para evitar elegir materiales inconsistentes con el tipo de prenda.*

Entonces quizás podamos darle al borrador el comportamiento para posibilitar esa creación en pasos, con un mensaje por cada uno de ellos:

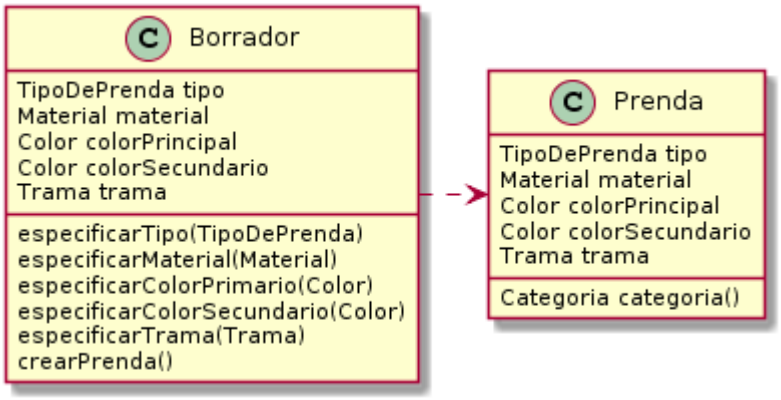


¿Qué hará cada uno de estos métodos? En principio, configurar el estado interno del borrador, para que registre lo que se ha especificado hasta ahora. ¿Cómo lo hará? Eso depende mucho de cada implementación:

- si la prenda fuera mutable, podría conocer a la prenda en construcción, e ir *seteando* sus atributos sobre la marcha
- si la prenda es inmutable (como en nuestro caso), deberá tener un conjunto de atributos similar al de la prenda

	
Si fuera la prenda fuera mutable	Nuestra situación: la prenda es inmutable

¿Y cómo obtenemos la prenda resultante? Podríamos exponer un mensaje `crearPrenda`, que en el primer escenario, simplemente devolverá la prenda en contrucción, y en el segundo escenario (el que se nos presenta ahora), la instanciará allí mismo:




```
class Borrador
    // ... demás atributos y métodos
    method crearPrenda
        return new Prenda(tipo, material, colorPrincipal, colorSecundario, trama)
```

¿Y con ésto qué ganamos? Por ahora, sólo **la construcción en pasos** y poder **representar a una prenda de forma mutable**. ¡Pero podemos hacerlo mejor!

Dado que ahora contamos con múltiples puntos para interceder en el proceso de construcción (el constructor de `Borrador`, cada uno de los `especificarAlgo` y el `crearPrenda`), podríamos agregar validaciones en cualquiera de ellos, siguiendo el principio de fail fast. Por ejemplo, podríamos:

1. Agregar validaciones de no nulos en cualquiera de los mensajes `especificarAlgo`. De esa forma, garantizamos que los errores de paso de nulos se detecten antes de `crearPrenda`
2. Mover todas las validaciones que estaban originalmente en el constructor de la prenda a `crearPrenda`, de esa forma quitándole responsabilidades y simplificando el constructor de `Prenda`, que pasa a sólo permitir la inyección de dependencias.
3. Llevar la configuración del tipo de prenda al constructor del borrador, dado que debe ser lo primero en especificarse
4. Validar aspectos más complejos, como la consistencia entre el material y el tipo de prenda (que no haya remeras de cuero o zapatos de seda), si así lo desearamos, al momento en que se establece el material

Combinando todas estas ideas, llegamos al siguiente pseucódigo:

```
class Borrador
    // atributos...

    constructor(tipoDePrenda) // [3]
       validateNonNull(tipoDePrenda)
       this.tipoDePrenda = tipoDePrenda

    method especificarColorPrincipal(color)
       validateNonNull(color) // [1]
       this.colorPrincipal = color

    method especificarMaterial(material)
```

```
        validateNonNull(material) // [1]
        this.validarMaterialConsistenteConTipoDePrenda(material) // [4]
        this.material = material

    method crearPrenda
        // resto de las validaciones que sigan siendo necesarias [2]
        // y que ya no son necesarias en Prenda
        // dado que SIEMPRE construiremos la prenda a través del borrador
        return new Prenda(tipo, material, colorPrincipal, colorSecundario, trama)
```

Como se puede apreciar, todos estos cambios propuestos están orientados a separar dos responsabilidades bien claras:

- Los problemas de comportamiento los resolverá la `Prenda` (aún no surgieron en estas dos iteraciones)
- Mientras que los problemas creacionales los resolverá fundamentalmente el `Borrador`.

Finalmente, recuperemos el último requerimiento pendiente de los borradores:

- *Como usuarie de QuéMePongo, quiero poder no indicar ninguna trama para una tela, y que por defecto ésta sea lisa.*

Ahora implementar este requerimiento es fácil: podríamos dejar configurado por defecto una trama, de forma que aunque no se envíe el mensaje `especificarTrama` tengamos una trama lisa. Incluso podríamos reemplazar cambiar completamente la lógica de `especificarTrama` para que nos resguarde contra `null`s:

```
class Borrador
    // ... demás atributos
    attribute trama = Trama.LISA

    //... demás métodos
    method especificarTrama(trama)
        this.trama = if trama is null then Trama.LISA else trama
```

De esa forma, tendremos siempre por defecto una trama lisa:

```
borrador = new Borrador(TipoDePrenda.REMERA)
borrador.especificarTrama(null) // se envía especificarTrama con null
//...resto las configuraciones...
remeraLisa = borrador.crearPrenda()

borrador = new Borrador(TipoDePrenda.PANTALON)
// no se envía especificarTrama
//...resto las configuraciones...
pantalonLiso = borrador.crearPrenda()

borrador = new Borrador(TipoDePrenda.CHOMBA)
```

```
borrador.especificarTrama(Trama.RAYADA) // en este caso se configura la trama dada
//...resto las configuraciones...
chombraRayada = borrador.crearPrenda()
```

Esta estructura de resolución de problemas creacionales es tan frecuente que lleva un nombre: patrón Builder, el cual está desarrollado en el apunte de patrones creacionales.

Para cerrar, repasemos algunas ventajas que nos dió el uso del patrón Builder.

- Tener una visión mutable de un objeto inmutable: aunque la Prenda es inmutable, el borrador es mutable.
- Posibilitar la construcción por pasos
- Asegurar un único punto de creación y validación: el mensaje crearPrenda (`build`)
- Simplificar la construcción implementando valores por defecto y calculables
- Independizarse de la representación interna del objeto construido: por ejemplo, si en el futuro el constructor de la clase Prenda cambia, o pasa a ser una clase abstracta con subclases, el `Borrador` podría amortiguar esos cambios, y mantener su interfaz de forma que quienes construyan prendas a través de éste no noten la diferencia.

Es importante resaltar que **no es necesario que se den todas estas características** al mismo tiempo en todo builder. Por ejemplo, los objetos de la clase `Calendar.Builder` de Java construyen `Calendars`, que de por sí ya son mutables, pero lo hace con el fin de simplificar la construcción.

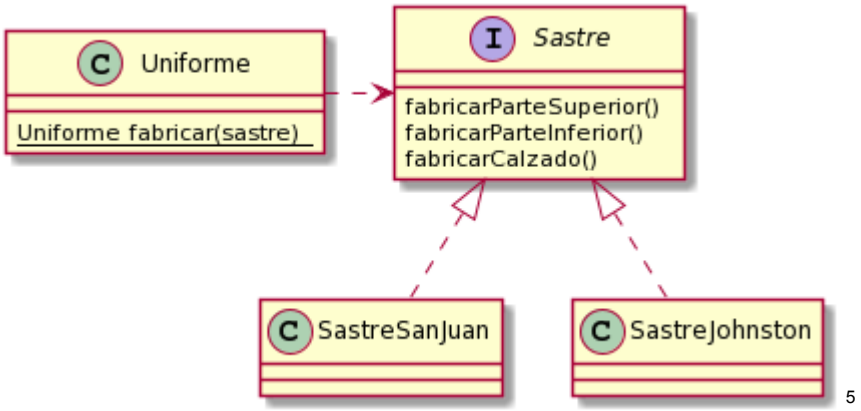
Tercera parte: los uniformes

Para este requerimiento dejamos dos soluciones igualmente viables como puntapié para seguir leyendo. Queda de tarea compararlas en materia de extensibilidad y simplicidad: ¿qué cambios o agregados al modelo facilita o dificulta cada una de ellas? ¿a qué costo?

Alternativa 1: Solución creacional basada en composición y Abstract Factory

```
class Uniforme
    attribute prendaSuperior
    attribute prendaInferior
    attribute calzado

    classmethod[4] fabricar(Sastre sastre)
       new Uniforme(
          sastre.fabricarParteSuperior(),
cale
          sastre.fabricarParteInferior(),
          sastre.fabricarCalzado())

interface Sastre
    method fabricarParteSuperior()
    method fabricarParteInferior()
    method fabricarCalzado()

class SastreSanJuan implements Sastre
     method fabricarParteSuperior()
          borrador = new Borrador(CHOMBA)
          borrador.especificarColor(new Color(....))
          borrador.especificarMaterial(PIQUE)
          return borrador.crearPrenda()

    method fabricarParteInferior()
          borrador = new Borrador(PANTALON)
          borrador.especificarColor(new Color(....))
          borrador.especificarMaterial(ACETATO)
          return borrador.crearPrenda()

     //etc...
```

[4] Un método de clase es un método que se define para la clase en sí y no para sus instancias, y por tanto el mensaje correspondiente puede ser enviado solamente a la clase: `Uniforme.fabricar(...)` y no `unUniforme.fabricar(...)`. No todos los lenguajes de programación lo soportan; en Java el equivalente más próximo son los métodos `static`. Ver apunte de Java



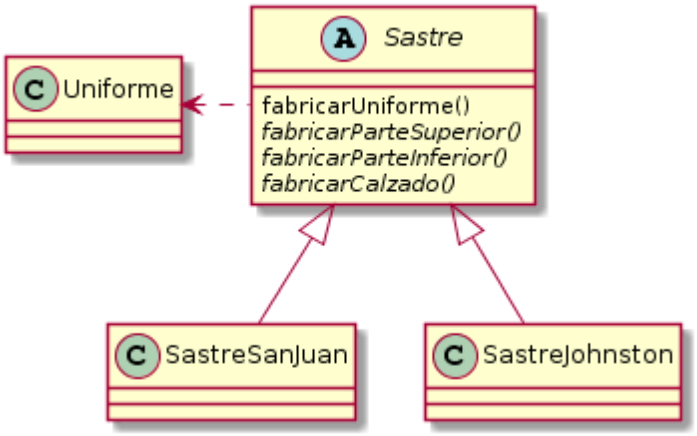
[5]

Alternativa 2: Solución creacional basada en herencia y Factory Method

```
class Uniforme
    attribute prendaSuperior
    attribute prendaInferior
    attribute calzado


abstract class Sastre
    method fabricarUniforme()
       new Uniforme(
            this.fabricarParteSuperior(),
            this.fabricarParteInferior(),
            this.fabricarCalzado())

    // estos método están aquí para ser implementados por las subclases,
    // y no para ser utilizados por otros objetos
    protected abstract method fabricarParteSuperior()
    protected abstract method fabricarParteInferior()
    protected abstract method fabricarCalzado()


class SastreSanJuan inherits Sastre
     method fabricarParteSuperior()
          borrador = new Borrador(CHOMBA)
          borrador.especificarColor(new Color(....))
          borrador.especificarMaterial(PIQUE)
          return borrador.crearPrenda()

    method fabricarParteInferior()
          borrador = new Borrador(PANTALON)
          borrador.especificarColor(new Color(....))
          borrador.especificarMaterial(ACETATO)
          return borrador.crearPrenda()
```

[5] En UML los métodos de clase se señalan son un subrayado

```
    //etc...
```


Sastre
Uniforme
fabricarUniforme()
fabricarParteSuperior()
fabricarParteInferior()
fabricarCalzado()
SastreSanJuan
SastreJohnston