# Laboratorio de Ingeniería del Software

## Diseño flexible (*supple design*)

# Contenidos

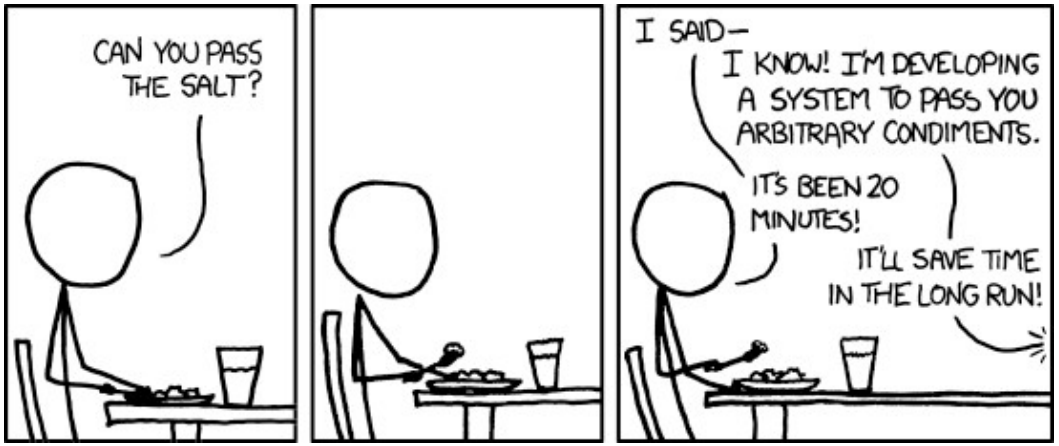
- El diseño flexible
- Interfaces reveladoras
- Funciones libres de efectos secundarios
- Aserciones
- Clases independientes
- Clausura de operaciones

# El diseño flexible

- El software tiene que servir a sus usuarios
  - Y su diseño tiene que servir a sus desarrolladores
- En procesos que enfatizan la refactorización y el refinamiento iterativo, y que requieren mantenimiento y evolución, el diseño es especialmente importante
  - Es decir, cuando desarrollamos software importante y que va a tener una vida larga
- Un mal diseño hace más difícil cambiar cosas
  - Se duplica código por no poder deducir con seguridad lo que harán los cambios en una parte existente
  - Se duplica código cuando el diseño es monolítico y no se pueden recombinar sus partes
    - Pero si el código se particiona mucho, es difícil asignar responsabilidades claras a todas estas pequeñas partes
  - Si el diseño no está claro, los desarrolladores tienen miedo de cambiar nada para no dejarlo aún peor

# El diseño flexible

- Lo que necesitamos es un diseño flexible
  - Que invite al cambio en lugar de complicarlo
- Muchos excesos de ingeniería se han justificado en nombre de la flexibilidad
  - Todos los problemas de la informática se pueden resolver añadiendo un nivel más de abstracción
    - Excepto el problema de tener demasiados niveles de abstracción
- No hay una fórmula mágica para hacer un diseño flexible
  - Pero sí que hay patrones que ayudan a proporcionar esa flexibilidad




https://xkcd.com/974/

# Interfaces reveladoras

# Interfaces reveladoras

- Los objetos bien diseñados encapsulan complejidad y ayudan a que el código que los usa sea simple e interpretable en base a conceptos de alto nivel
- Si la interfaz del objeto no le dice al desarrollador del código que lo usa lo que necesita para usarlo de manera efectiva, este tendrá que averiguar cómo funciona y sacar a la luz los detalles internos
  - Igual que cualquiera que luego lea este código
- Hay que luchar contra la sobrecarga cognitiva
  - Cuantas menos cosas tenga que tener en la cabeza un desarrollador en un momento determinado, más fácil será su trabajo
- **Los nombres de las clases y las operaciones deben revelar su efecto y su propósito**
  - Sin referencia a los medios para hacer lo que prometen
  - Esto permite al desarrollador que lo usa no tener que entender cómo es por dentro
- Estos nombres deben pertenecer al lenguaje compartido
  - Esto ayuda a que todo el mundo conozca lo que significan
- Escribir un test antes de crear un comportamiento (*behavior driven testing, test driven design*) ayuda a ponerse en el lugar del desarrollador que usará nuestro objeto

# Funciones libres de efectos secundarios

# Funciones libres de efectos secundarios

- Simplificando mucho hay dos tipos de operaciones
  - Los comandos, que realizan cambios en el sistema
  - Las consultas, que devuelven información sobre el sistema
- Cuando en informática hablamos de efecto secundario (efecto lateral, *side effect*) normalmente nos referimos a cualquier cambio observable en el estado del sistema (intencionado o no)
  - Nos podemos focalizar en los cambios que afectarán a futuras operaciones del sistema
- Las operaciones que devuelven resultados sin efectos secundarios se llaman funciones
  - Se pueden llamar muchas veces y, con los mismos parámetros, siempre devolverán el mismo resultado
  - Unas funciones pueden llamar a otras con varios niveles de anidamiento y aún así su resultado será más fácil de entender
  - Es más fácil escribir pruebas automáticas para funciones que para operaciones con efectos secundarios

# Funciones libres de efectos secundarios

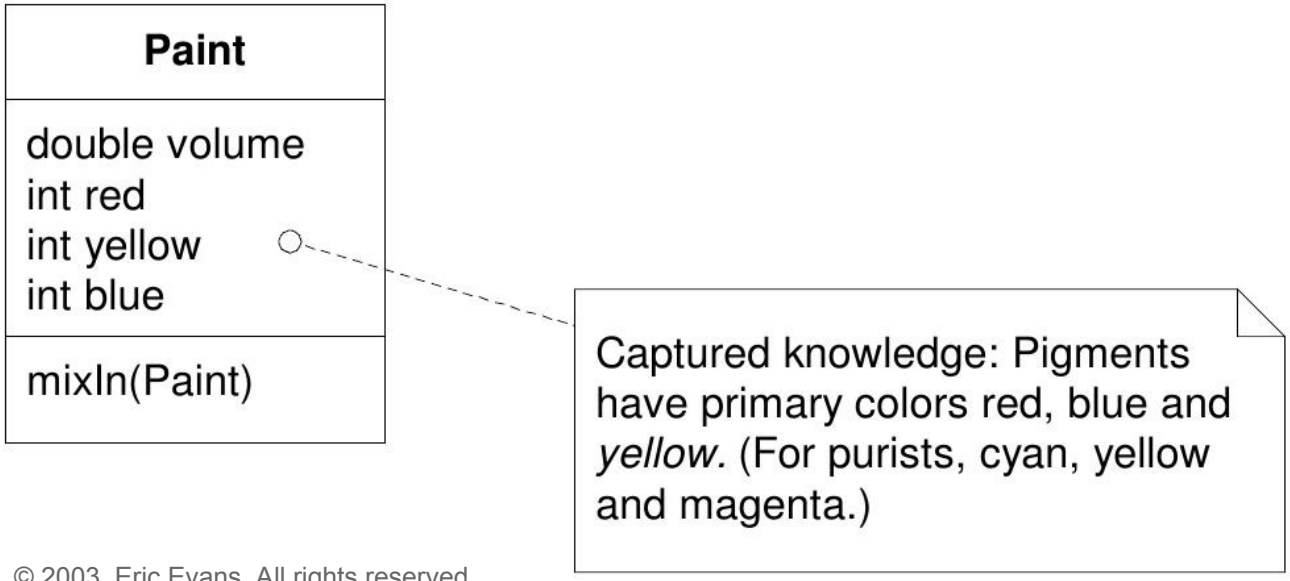
- En la mayoría de sistemas de información no podemos evitar los comandos, pero podemos mitigar algunos de sus problemas potenciales
  - Segregando comandos y consultas en operaciones distintas
    - Los métodos que producen cambios no devuelven datos del dominio
    - Las consultas y cálculos se hacen en métodos que no causen efectos secundarios observables
  - Buscando diseños alternativos que eviten la necesidad de modificar objetos existentes
    - Por ejemplo, creando objetos valor nuevos para representar el resultado de cada computación
- Los objetos valor son inmutables y, por tanto, aparte de su creación todas sus operaciones son funciones

# Funciones libres de efectos secundarios

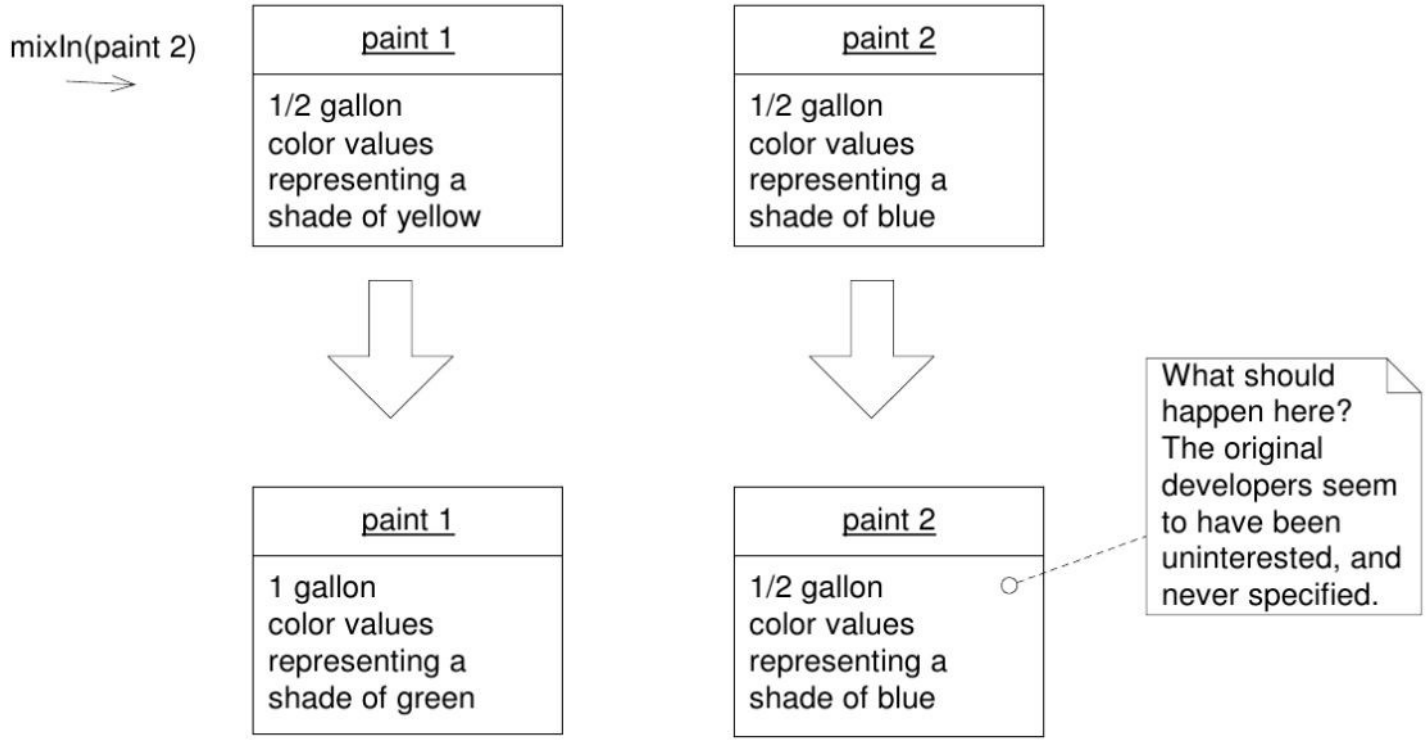
- Trata de poner tanta lógica del programa como puedas en funciones
- Segrega los comandos (que modifican el estado observable) en operaciones simples
  - Y que estas no devuelvan información del dominio
- Mueve la lógica compleja a objetos valor
- Las funciones en objetos valor inmutables permiten combinar operaciones de forma segura
  - Y si tienen una interfaz reveladora, son más fáciles de usar sin entender los detalles de su implementación

# Refactorización de diseño de mezcla de pinturas

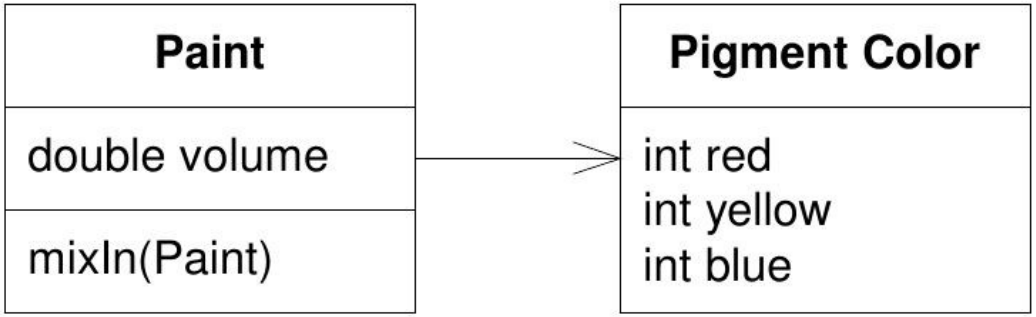
- Modelo de objetos original
  - Sirve para mezclar pinturas estándar (de catálogo) y obtener el color exacto que buscamos



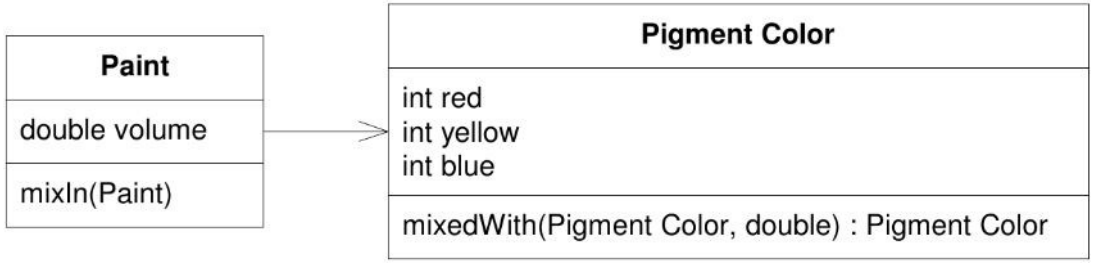
```
public void mixIn(Paint other) {
  this.volume = this.volume + other.volume;
  // Líneas con la lógica de mezcla de colores
  // que terminan con la nueva asignación de valores
  // de rojo, azul y amarillo
}
```



- El volumen del objeto `paint2` se queda en el limbo, no sabemos exactamente qué pasa con él
  - Se podría pensar que lo lógico es que se quedara a 0, pero parece que a los desarrolladores originales esto no les hizo falta considerarlo
- También parece que el color es algo importante en este dominio, así que convendría separarlo
  - Como en este contexto no es lo mismo un color de luz que uno de pigmento (no se mezclan igual), además le damos un nombre apropiado



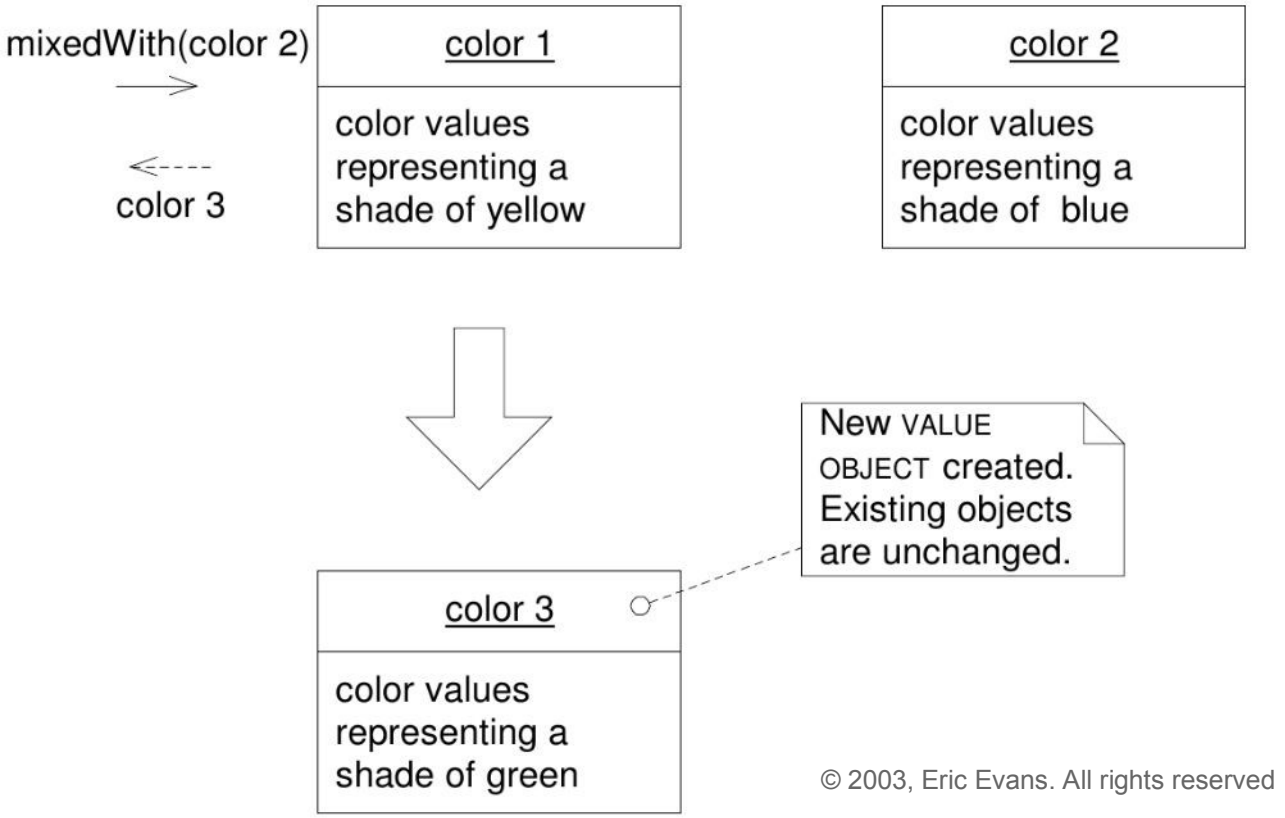
- La versión anterior comunica más, pero el código de `mixIn()` es el mismo. `Pigment Color` es un objeto valor, así que deberíamos hacerlo inmutable. Cuando mezclábamos pintura, el volumen de esta cambiaba (`Paint` tiene estado mutable). Si mezclamos colores, el resultado es otro color, no hace falta modificar nada:






```
public class PigmentColor {
  public PigmentColor mixedWith(PigmentColor other, double ratio) {
    // Lógica de mezcla de colores que termina con la creación de un
    // nuevo objeto PigmentColor con los valores apropiados
  }}
public class Paint {
  public void mixIn(Paint other) {
    this.volume = this.volume + other.volume;
    double ratio = other.volume / this.volume;
    this.pigmentColor =
             this.pigmentColor.mixedWith(other.pigmentColor, ratio);
  }}
```

- El nuevo código traslada el conocimiento sobre mezcla de colores a `Pigment Color`, que además proporciona una función libre de efectos secundarios (fácil de entender y de probar, y segura para combinarla con otras)



# Aserciones

# Aserciones

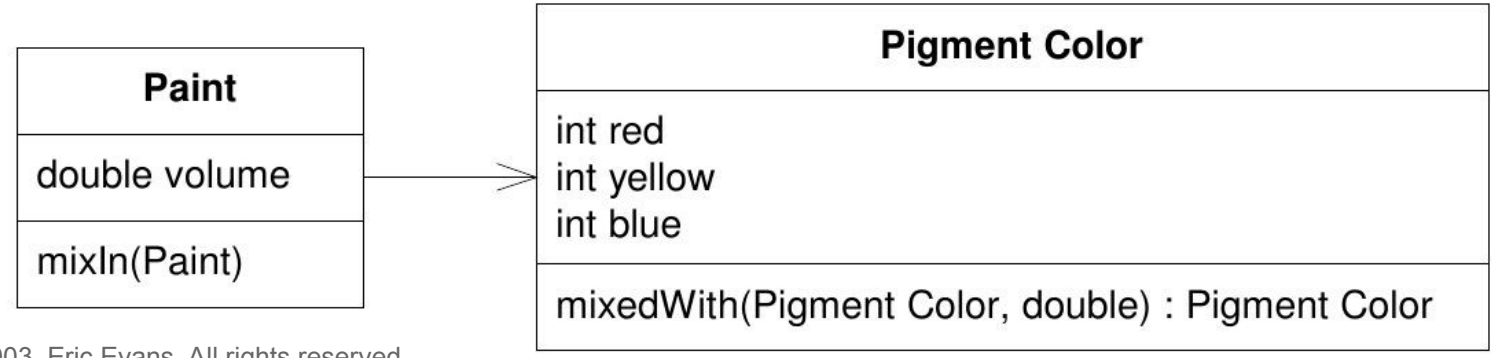
- Una vez hemos creado funciones sin efectos secundarios, todavía nos quedan algunos comandos en las entidades que sí producen efectos secundarios
  - Las aserciones nos permiten tratar con ellos de manera más segura
    - Permitiéndonos definir los requisitos y resultados de una operación explícitamente
- Una aserción es un predicado que debería ser cierto
- Recordatorio: el diseño por contrato sugiere hacer aserciones sobre clases y métodos que el desarrollador garantiza que serán ciertos
  - Post-condiciones: describen los efectos secundarios de una operación. Lo que se garantiza que será cierto tras llamar a un método
  - Precondiciones: condiciones que deben ser satisfechas para garantizar que se cumplirá la post-condición
  - Invariante de clase: aserciones sobre el estado de un objeto al final de cualquier operación del mismo
    - También tendremos invariantes para los agregados
- Las aserciones describen estado y no procedimientos, así que en general son más sencillas de analizar
- Los invariantes de clase ayudan a expresar las responsabilidades de la misma

# Aserciones

- Expresa las pre- y post-condiciones de las operaciones (especialmente de las que causan efectos secundarios observables) y los invariantes de clases y agregados
  - Escribe test unitarios para las aserciones. Prueba qué pasa cuando se cumplen y cuando no
  - Escribe las aserciones en la documentación, los diagramas o donde encaje con tu proceso de desarrollo
  - Procura que tus aserciones sean comprensibles para alguien que conozca el modelo

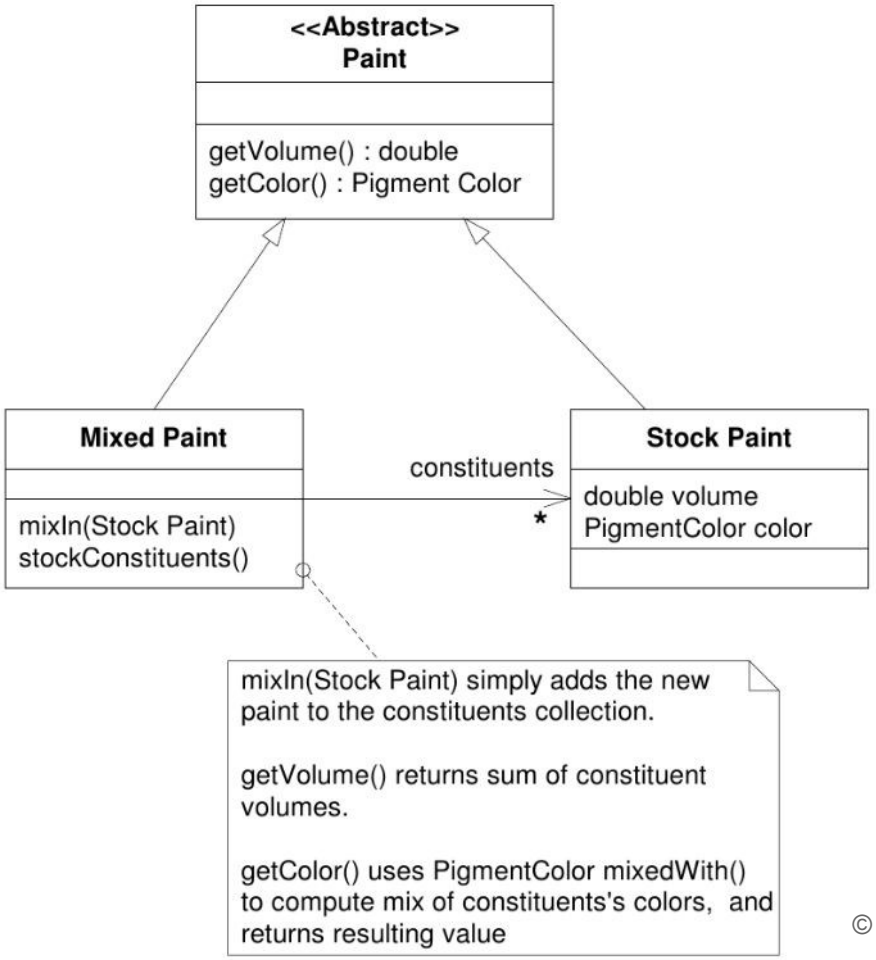
# Ejemplo: más pintura

- En el ejemplo de la pintura, nos habíamos quedado con la duda sobre lo que pasaba a la pintura donante que mezclábamos con la receptora
  - La receptora aumentaba su volumen, pero la donante se quedaba igual, cuando lo lógico podría ser que se quedase a cero



- Podemos explicar lo que pasa ahora en la post-condición de `mixIn()`:
  - *Después de `p1.mixIn(p2)`, el volumen de `p1` será el actual más el volumen de `p2`. El volumen de `p2` no cambia*
- El problema de esto es que no concuerda con lo que esperamos de la realidad. La solución inmediata sería alterar el volumen de `p2` y dejarlo en cero. **Modificar los argumentos que le han pasado a una operación es, en general, una mala práctica**, pero en este caso podría funcionar. Podríamos tener este invariante:
  - *El volumen total de pintura no cambia al mezclar*
- El problema es que al final descubrimos que lo que hay ahora tiene sentido, porque al final de todo, el programa muestra la lista de pinturas que fueron añadidas a la mezcla
  - El propósito del programa es mostrar qué pinturas hay que mezclar para sacar cierto color, no simular una mezcla física de las mismas
- Nos podríamos quedar con un comportamiento poco intuitivo, y confiar en la documentación (incluyendo la post-condición de arriba) para compensarlo
  - A veces no hay otra opción

- Pero en este caso, esta cosa extraña que tenemos nos está indicando que nos faltan conceptos. Concretamente estamos dándole a `Paint` dos responsabilidades. Separemos:



- Ahora el único comando (efectos secundarios) que tenemos es `mixIn()`, que se limita a añadir un objeto a una colección (las pinturas de stock que van a una mezcla determinada)
  - Además el modelo comunica más sobre el dominio (ahora queda claro que para algo necesitamos las pinturas originales de la mezcla), y los invariantes y post-condiciones no violan la ley de conservación de la masa :-)
- Una prueba para confirmar una de las aserciones del diagrama anterior (expresadas en la nota) podría ser así:

```
public void testMixingVolume {
  PigmentColor yellow = new PigmentColor(0, 50, 0);
  PigmentColor blue = new PigmentColor(0, 0, 50);
  StockPaint paint1 = new StockPaint(1.0, yellow);
  StockPaint paint2 = new StockPaint(1.5, blue);
  MixedPaint mix = new MixedPaint();

  mix.mixIn(paint1);
  mix.mixIn(paint2);
  assertEquals(2.5, mix.getVolume()); // ¿Cuál es el fallo en esta línea?
}
```

# Clases independientes

# Clases independientes

- La interdependencia hace que los modelos sean más difíciles de entender, de probar y de mantener
  - Cada asociación, cada argumento de cada método, cada valor de respuesta de una función etc. es una interdependencia
- Con una dependencia tienes que pensar en dos clases a la vez. Con dos, tienes que pensar en tres clases...
  - Si estas a su vez tienen otras dependencias, acabas teniendo que tener muchas cosas en cuenta
  - La sobrecarga cognitiva que esto le produce al desarrollador limita la complejidad que puede manejar
- Los paquetes y los agregados ayudan a limitar esta telaraña de interdependencias
  - Pero incluso dentro de un paquete podemos tener demasiadas dependencias

# Clases independientes

- Si el número de dependencias se puede reducir a cero, una clase se entenderá completamente por si misma
  - Junto a algunas primitivas y conceptos básicos de biblioteca (números, cadenas, colecciones...)
- Hay que eliminar las dependencias no esenciales, no todas
- Las dependencias con otras clases del mismo paquete son menos dañinas que con clases externas
- El acoplamiento bajo es fundamental para el diseño de objetos
  - Cuando se pueda, hay que llevarlo al extremo y diseñar clases independientes
- El patrón clausura de operaciones es un ejemplo de técnica para reducir dependencias manteniendo una interfaz rica

# Clausura de operaciones

# Clausura de operaciones

- Un conjunto está cerrado (o clausurado) bajo cierta operación si el resultado de esa operación siempre es un miembro de ese conjunto
  - Por ejemplo, los reales están cerrados bajo la multiplicación
    - Pero no para la división

# Clausura de operaciones

- Las dependencias no son todas evitables, pero muchas de las evitables se introducen en las interfaces
- Esto se evitaría haciendo que las interfaces solo usaran tipos primitivos (que no contamos como dependencias problemáticas)
  - Pero empobrecería el diseño de las interfaces de una forma que no nos podemos permitir
  - Los objetos más interesantes hacen cosas que normalmente no pueden caracterizarse solo mediante primitivas

# Clausura de operaciones

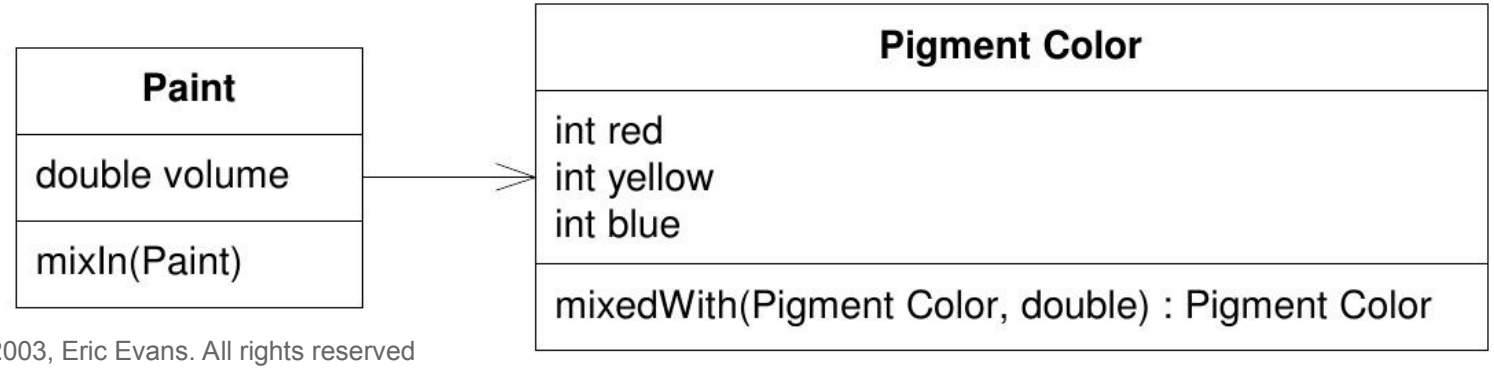
- Donde esto encaje, define tus operaciones de manera que devuelvan objetos del mismo tipo que sus argumentos
  - Si el objeto que implementa la operación tiene un estado que se usa en ella, entonces ese objeto es un argumento (`this, self...`)
    - En es caso, los argumentos y el valor de respuesta deberían ser todos del mismo tipo que `this`
- Esas operaciones son cerradas para el conjunto de instancias (objetos) de la clase que las incluye
  - Una operación cerrada proporciona una interfaz con alto nivel de abstracción sin introducir dependencias con otros conceptos
- Por ejemplo, la operación `concat` de la clase `String` de Java `public String concat(String str)` es de clausura para el conjunto de los Strings en Java
  - Concatenar dos `Strings` siempre da un `String`

# Clausura de operaciones

- La clausura se suele aplicar en operaciones de objetos valor
  - Las entidades tienen un ciclo de vida largo, y no es común que creemos algunas nuevas como respuesta a una operación
  - No son el tipo de cosas que serán el resultado de una computación
- "Media clausura" es mejor que nada
  - Si algún argumento, o quizás el tipo de respuesta, se pueden hacer coincidir con el del objeto que implementa la operación, un tipo extra que nos evitamos tener en la cabeza cuando razonemos sobre la operación
  - Y si el resto de argumentos son tipos primitivos, casi tendremos las mismas ventajas (minimizar la sobrecarga cognitiva) que con una clausura y podemos considerarla como tal

# Ejemplo de clausura de operaciones

- En este ejemplo, que ha salido anteriormente, la clase `PigmentColor` está cerrada respecto a la operación `mixedWidth()`
  - También podemos decir que `mixedWith()` es una operación de clausura sobre el conjunto de instancias de `PigmentColor`



# Bibliografía


- Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*
  - Capítulo 10