Dönem Sonu Projesi

160202034 - Ata Gülalan

Bu projede, tek projede tüm verilen tasarım örüntüleri kullanılmaya çalışılmıştır.

Bu projede SOLID prensibinin ilk elemanı olan Single-Responsibility Principle kullanılmıştır.

Burada aşağıdaki açıklamalara bakarak bu prensibe uyup uymadığımızı anlayabiliriz:

> *Sınıfı adlandırmak ne kadar kolay? Bir sınıfı adlandırmak zorsa, muhtemelen çok fazla şey yapıyor demektir.*

Adlandırmalar basit ve anlaşılabilir.

> *Sınıfta kaç tane genel yöntem var? 7 +/- 2 iyi bir kural. Sınıf bundan daha fazlasına sahipse, onu birkaç sınıfa bölmeyi düşünmelisiniz.*

Herhangi bir sınıf 5 methodu geçmiyor.

> *Ayrı bağlamlarda kullanılan ortak yöntemlerin uyumlu grupları var mı?*

Ortak yöntemler var, ancak farklı komut satırları içeriyorlar.

> *Sınıf karmaşık bir iç yapıya sahipse, muhtemelen iç kısımları daha küçük sınıflara paketlenecek şekilde yeniden düzenlemelisiniz.*

Sınıflar, herhangi bir karmaşık iç yapıya sahip değil.

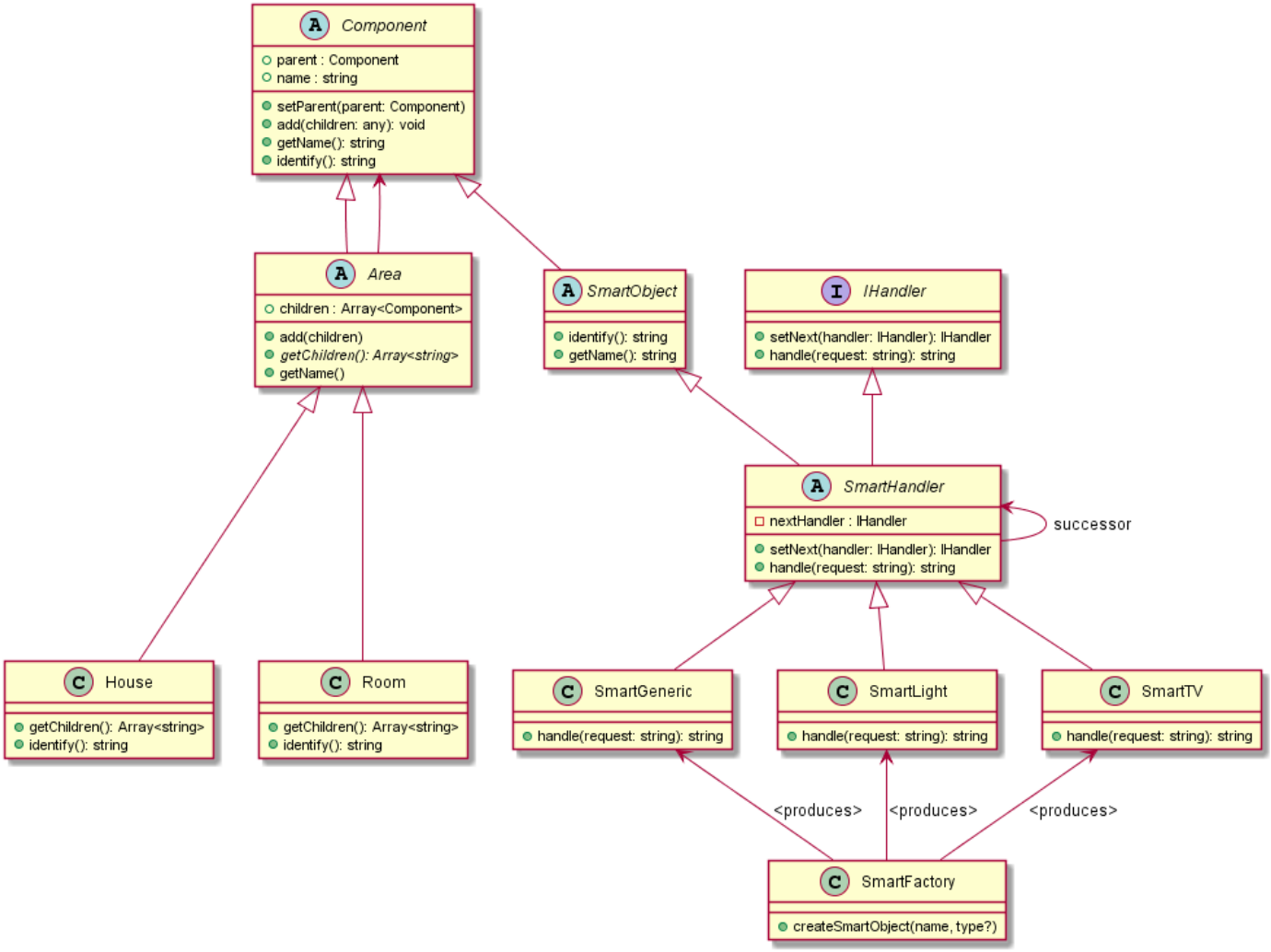
> *Ve en kolay kural: sınıf ne kadar büyük? Birkaç yüz satırdan daha uzun tek bir sınıf içeren bir dosyanız varsa, muhtemelen bölmelisiniz.*

Her sınıf birkaç satırdan oluşuyor.

Bu projede aynı zamanda chain of responsibility tasarım örüntüsü kullanılmıştır. Chain of Responsibility tasarım örüntüsü, istekleri bir zincir gibi sıralayarak çağırmamızı sağlayan davranışsal bir tasarım modelidir. Bir istek alındığında, her işleyici ya isteği işlemeye ya da zincirdeki bir sonraki işleyiciye geçirmeye karar verir. Bu işlem evdeki akıllı nesnelerin sırasıyla işlemi yapıp yapmayacağına karar vermesini sağlar.

Son olarak, projede kullanılan Composite tasarım örüntü, akıllı ev sistemini oluşturmayı sağlamıştır. Bir akıllı ev, sonrasında branch için akıllı evin altında bulunan birden çok oda ve son olarak leaf olarak akıllı cihazlar kullanılmıştır. Bu akıllı cihazların hangi odada olduğu veya hangi odada hangi akıllı cihazların olduğu listelenebilir.

UML



Dosyalar

Bu başlık altında projede kullanılan klasörler listelenmiştir.

Components

Bu klasörde bulunan sınıflar, Component sınıfından türetilmiştir. Composite lorem

---

Area.ts

Bu sınıf, Component sınıfından extend edilir. Bir alanı ifade eder. Bu alanın içerisinde akıllı cihazlar bulunabilir. Bu açıdan composite tasarım örüntüsünün dal kısmına benzetilebilir. Bu sınıfın çocuk objeleri vardır ve add methodu çağırılarak bu obje dizisine yeni objeler eklenebilir.

```
children: Array<Component> = []
// (1)
add(...children) {
  children.forEach((child) => {
    this.children.push(child)
    child.setParent(this)
  })
}
```

1. Spread operatörü kullanılarak her bir parametrede verilen çocuk objeler, children dizisine daha sonra kullanılmak üzere atılır.

SmartObject.ts

Bu sınıf, aynı Area sınıfı gibi Component sınıfından extend edilmiştir. Bu sınıfın Area sınıfından farkı, composite tasarım örüntüsündeki branch'e değil, leaf'e benzemesidir. Bu sınıfın çocuk sınıfları yoktur, ancak Component sınıfından türetildiğinden setParent methodunu içirisinde barındırır.

```
...
  getName(): string {
    return `\x1b[35m${this.name}\x1b[0m` // (1)
  }
...
```

1. Burada gösterilen karmaşık karakterler, bu objenin isminin konsolda renkli gözükmesini sağlar.

SmartDevices

Bu klasör altında bulunan sınıflar ise Chain of Responsibility tasarım örüntüsünü çağırıştırır. Odalarda bulunan akıllı objelerin, verilen komutları dinleyip, eğer kendisine ait bir komut içermiyorsa bir sonrakine iletmesi prensibine dayalı çalışır.

SmartGeneric.ts

Bu sınıf, SmartHandler'ı extend etmiştir. Böylece handle metodu override edilerek genel tanımlanmış bir akıllı nesnenin, "Open [Akıllı Nesne Adı]" komutu kullanılarak açılması sağlanmıştır.

```
// (1)
class SmartGeneric extends SmartHandler {
  ...
  // (2)
  public handle(request: string): string {
    if (request === 'Open ' + this.name) {
      return `Opening generic: ${request}.`
    }
    return super.handle(request)
  }
}
```

1. SmartHandler sınıfından extend edilir. Böylece setNext metodu ana sınıftan alınır.
2. Handle metodu override edilerek kriteri sağlaması halinde handle edilmesi sağlanır.

SmartLight.ts

Bu sınıf, üstteki sınıfa çok benzer. Tek farkı ise "open" anahtar kelimesinin yerine "Dim Lights [Yüzde]" kullanılarak çalıştırılmasıdır. Örnek verecek olursak, "Dim Lights %30", zincirdeki en yakın ışığı kısar.

```
class SmartLight extends SmartHandler {
  ...
  public handle(request: string): string {
    if (request.startsWith('Dim Lights')) {
      // (1)
      return `Dimming ${this.parent.getName()} Lights`
    }
    return super.handle(request)
  }
}
```

1. Composite tasarım örüntüsü kullanıldığından, ışık hangi odada olduğunun çıktısını verebilir.

Classes

Burada klasör tüm sınıfları tutacak şekilde konumlandırılmıştır. Ana dizindeki klasörlerin dışında, tüm projede kullanılan sınıfları içerir.

Components.ts

Composite tasarım örüntüsünün olmazsa olmazı component sınıfını tanımlar. Bir ebeveyn, bir isim ve kendini ifade edebilmesi için identify metodunu bulundurur. Burada getName ve identify metotlarının override edilmesi zorunludur.

```
abstract class Component {
  parent: Component
  name: string
  constructor(name) {
    this.name = name
  }
  // (1)
  setParent(parent: Component) {
    this.parent = parent
  }
  add(...children): void {}
  abstract getName(): string
  abstract identify(): string
}
```

1. Composite tasarım örüntüsünde kullanılan objeye evebeyn atama işlemi.

SmartFactory.ts

Konum olmayan, ancak kullanmak istediğim bir tasarım örüntüsü olan factory tasarım örüntüsünü kullandım. Eğer tip değişkeni atanmışsa bu tipe ait sınıftan oluşturulur. Eğer tip atanmamış ise obje, SmartGeneric sınıfından üretilir.

```
class SmartFactory {
  // (1)
  createSmartObject(name, type?) {
    if (type) return new type(name)
    return new SmartGeneric(name)
  }
}
```

1. Bir SmartObject oluşturan, Factory tasarım örüntüsünde görülen bir metot.

SmartHandler.ts

Bu sınıf, Chain of Responsibility tasarım örüntüsünde kullanılan Handler sınıfını tanımlar. Bir setNext ve bir handle metodu bulundurur. Burada bulunan setNext metodu, nextHandler değişkenine atama yapar ve bu handler'ı döndürür. Böylece objeleri bir zincir gibi birbirine bağlayabilir ve birbiri ardına çalışmasını sağlayabiliriz. Handle override edilmediğinde otomatik bir sonraki objenin handle metodunu çağırır.

```
// (1)
interface IHandler {
  setNext(handler: IHandler): IHandler
  handle(request: string): string
}

abstract class SmartHandler extends SmartObject
  implements IHandler {
  private nextHandler: IHandler

  // (2)
  public setNext(handler: IHandler): IHandler {
    this.nextHandler = handler
    return handler
  }

  public handle(request: string): string {
    // (3)
    if (this.nextHandler) {
      return this.nextHandler.handle(request)
    }

    return null
  }
}
```

1. Handler'ı tanımlamak ve setNext metodunda yalnızca bir handler içeren parametre almak için bir interface tanımlıyoruz

1. Handler'ı tanımlamak ve setNext metodunda yalnızca bir handler içeren parametre almak için bir interface tanımlıyoruz.
2. Bu metot nextHandler değişkenine atama yapar ve bu handler'ı döndürür.
3. Handle override edilmediğinde otomatik bir sonraki objenin handle metodunu çağırır.

Demo

Bu başlık altında demoya ait kodun açıklamasını ve demonun çıktısını bulabilirsiniz.

---

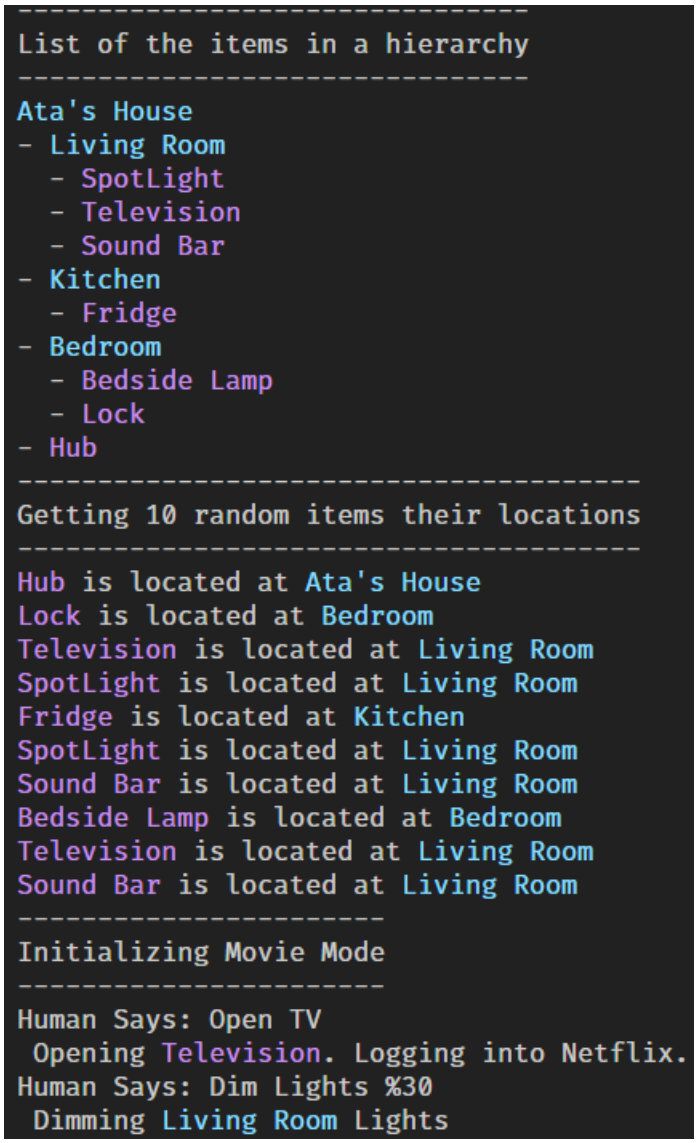
demo.ts

Demo dosyası, projede kullanılan sınıfların istenilen kriterlere uygun çalışıp çalışmadığını anlamamıza yardımcı olur. Bu dosyanın açıklamalarını aşağıda bulabilirsiniz.

```
...

function createSmartHouse() {
  // (1)
  const smartHouse = new House("Ata's House")

  const livingRoom = new Room('Living Room')
  const kitchen = new Room('Kitchen')
  const bedroom = new Room('Bedroom')

  const smartFactory = new SmartFactory()

  const smartHub = smartFactory.createSmartObject('Hub')
  const smartLight = smartFactory.createSmartObject('SpotLight', SmartLight)
  const smartTV = smartFactory.createSmartObject('Television', SmartTV)
  const smartSoundBar = smartFactory.createSmartObject('Sound Bar')
  const smartFridge = smartFactory.createSmartObject('Fridge')
  const smartLock = smartFactory.createSmartObject('Lock')
  const smartBedsideLamp = smartFactory.createSmartObject('Bedside Lamp')

  livingRoom.add(smartLight, smartTV, smartSoundBar)
  kitchen.add(smartFridge)
  bedroom.add(smartBedsideLamp, smartLock)

  smartHouse.add(livingRoom, kitchen, bedroom, smartHub)

  // (2)
  smartHub
    .setNext(smartLight)
    .setNext(smartTV)
    .setNext(smartSoundBar)
    .setNext(smartFridge)
    .setNext(smartLock)Sm
    .setNext(smartBedsideLamp)

  return { house: smartHouse, hub: smartHub }
}

function listDevices(component) {
  // (3)
  dashedLog('List of the items in a hierarchy')
  console.log(component.identify())
}

// (4)
function getRandom10(component) {
  dashedLog('Getting 10 random items their locations')
  for (let i = 0; i < 10; i++) {
    let smartDevice = getRandomSmartDevice(component)
    whereIs(smartDevice)
  }
}

...

function initializeMovieMode(hub) {
  dashedLog('Initializing Movie Mode')
  let modeOptions = ['Open TV', 'Dim Lights %30']
  for (const option of modeOptions) {
    console.log(`Human Says: ${option}`)
    const result = hub.handle(option) // (5)
    if (result) {
      console.log(` ${result}`)
    } else {
      console.log(` Couldn't understand command.`)
    }
  }
}

...
```

1. Bir composite oluşturarak proje üzerinde bir akıllı ev, sonrasında bu akıllı eve odalar, ve bu odalara akıllı cihazlar konulur.
2. Bir chain of responsibilty oluşturularak hub çağırıldığında tüm akıllı cihazları dönen bir handle metodu çağırılır.
3. En tepedeki obje çağırılarak, bu objenin altında bulunan diğer objeleri yazırabiliriz.
4. 10 rastgele cihaz seçilerek bu cihazların yerleri ekrana yazdırılır.
5. Yalnızca ilk eleman olan hub'a ait handle metodu çağırılarak bir nesne ilgili komutu işleyene kadar tüm akıllı nesnelere iletilir.

Çıktı Ekran Görüntüsü



Last updated 2020-06-03 01:47:39 +0300