Ontwerptechnieken

Inhoud

[Ontwerptechnieken 1](#_Toc379400049)

[Object Oriented Programming 2](#_Toc379400050)

[SOLID Principles 2](#_Toc379400051)

[GRASP 3](#_Toc379400052)

[Design Patterns 3](#_Toc379400053)

[Refactoring 4](#_Toc379400054)

[C# 4](#_Toc379400055)

[Test driven development: 4](#_Toc379400056)

# Object Oriented Programming

Abstract class

public abstract class Actie: Dit is een klasse die alleen afgeleide klassen kent.

Polymorphism

Volop gebruik, diverse klassen worden van elkaar afgeleid: Acties, Kaarten, etc.

Dynamic Binding, Overloading

Bij iedere functie van een afgeleide klasse in gebruik;

Overloading

Bijvoorbeeld in gebruik bij InvalidArrayLengthException: de constructor heeft verschillende varianten.

# SOLID Principles

Open / Closed principle

Door het gebruik van het Strategy pattern in bijvoorbeeld de acties kunnen we nieuwe functionaliteit toevoegen door een nieuwe actie te defeinieren, afgeleid van klasse Actie, zonder bestaande code te veranderen.

Single Responsibility principle

Weer is de Actie klasse een goed voorbeeld: deze implementeert de 'logica' van het spel en heeft geen andere, niet verwante functies.

Interface segregation principle

Dit wordt behoorlijk overtreden in onze applicatie. De Actie klasse heeft weet van de user interface; dit hadden we kunnen voorkomen door bijvoorbeeld de logging als lijsten van strings op te zetten. We hebben hier niet voor gekozen omdat de huidige oplossing simpeler is.

Dependency inversion

Een vakje krijgt een landingsactie; heeft verder geen weet van de werking, en alleen dependency op de Actie klasse. D it is een voorbeeld van dependency inversion, het vakje krijgt een Actie mee in zijn constructor.

# GRASP

Controller

Wij maken gebruik van een MVC pattern; MonopolyController is dus de controller die systeemacties (knoppen) afhandeld.

Creator

Kalsse Vakjes beheert een List van klasse Vakje, en is de enige diez e creeert en beheert.

High Cohesion /low coupling

Ons project heeft een lage 'cohesion' : klasses hebben een duidelijke gedfinieerde, eigen verantwoordelijkheid.

# Design Patterns

Strategy Pattern

De acties zijn een Strategy pattern: de functie VoerUit is een schoolvoorbeeld.

Observer Pattern

We hebben geprobeerd deze te implementeren. Dit bleek ietwat overtrokken, omdat de C# implementatie ook gebruik maakt van een 'message' klasse, waar wij niet direct een toepassing voor zagen.

We hadden wel zelf een implementatie kunnen bedenken. Dit zou betekenen dat de views zich aanmelden bij de model, die een lijst bijhoudt van zijn views, en bij wijzigingen deze meldt aan al zijn aangemelde views.

Composite Pattern

De actie GaNaarActie lijkt op een composite pattern omdat hij andere acties aanroept, maar is dit toch niet want de acties worden opgezocht in een andere lijst.

Model, view, controller:

gebruikt als basisopzet van de applicatie. Classes zijn duidelijk uit de code. Verantwoordelijkheden zijn geïmplementeerd volgens het boekje!

# Refactoring

Replace Inheritance with Delegation

Dit is toegepast bij de vakjes. Deze hadden logica voor een speler die op een vakje land. Deze logica is verhuist naar de klasse Actie. Een vakje heeft nu een member van het type actie. Hiermee vervangen we Inheritance (soorten vakjes) door Delegation (een klaase Vakje, met een member van klasse Actie).

# C#

Serializable: niet uitgewerkt. Dit hadden we kunnen doen om een Svae/Load functie te maken. Alhoewel de techniek ons duidelijk is, zijn we hier niet aan toe gekomen.

Generics (bv. List<T> 🡪 List<string>) : Hiervan maken wij gebruik, bijvoorbeeld in de List<Vakje> van de klasse Vakjes, en Kaarten: private List<Kaart> kaarten;

Exceptions: Wijhebben een eigen exception gemaakt. Deze gaat alleen af bij een coding fout, en we hebben er bewust voor gekozen deze niet af te vangen met een try/catch. het is namelijk geen gebruikersfout o.i.d. en daarom is het gewenst gedrag dat de applicatie stopt.

# Test driven development:

Unit tests: DobbelsteenTest en SpelersTest in MonopolyTestProject.