SEPTEMBER 2023

Skrevet for Word | Dark Mode

EKSAMEN

EMNE 3

STUDENT 1012

Oppgave 1B

Single Responsability Principle (SRP)

SRP er et sentralt prinsipp i objektorientert programmering, en del av SOLID-designprinsippene.

Prinsippet fastslår at hver modul, klasse eller funksjon skal ha kun ett spesifikt ansvarsområde. Hovedmålet er å oppnå løs kobling mellom komponentene, for å skape vedlikeholdsbare og lettfattelige kodesystemer.

Når en klasse håndterer flere ansvarsområder, blir oppgavene sammenvevd, som bidrar til en økende sårbarhet. For eksempel vil en brukerklasse som håndterer både lagring og formatering av brukerdata, bryte med SRP.

Løsningen er å dele disse ansvarsområdene i separate klasser som UserData og UserDisplayFormatter.

Ved å følge SRP skapes et enklere, forståelig og utvidbart system.

Isolering av hver klasse og modul reduserer kompleksiteten ved endringer, bidrar til vedlikehold og sikrer en solid og skalerbar kode.

Windows Presentation Foundation (WPF)

Om en skulle endre denne applikasjonen til WPF, kan Program, Paddle og Particle klassene fint gjenbrukes. Dette er grunnet deres distinkte ansvar i spilllogikken.

En omskriving av Game vil være nødvendig for å skille spilllogikk fra brukergrensesnittlogikk

FallingParticles

class Program

Denne klassen starter spillet på en ansvarlig måte, og tilpasser seg SRP ved å ha et enkelt ansvar.

class Game

Håndterer ulike oppgaver som spilllogikk, initialisering, tegning, partikkelbevegelse, kollisjonssjekker og brukerinndata.

Dette bryter med SRP. For å løse dette, bør man vurdere å dele det inn i forskjellige klasser.

class GameLogic - håndtere spillets kjernelogikk

class GameRenderer - administrere tegning av spilltilstanden.

class UserInput - håndtere brukerinteraksjoner.

class Paddle

I henhold til SRP er denne klassen eksklusivt ansvarlig for padlens bevegelse og tegning.

class Particle

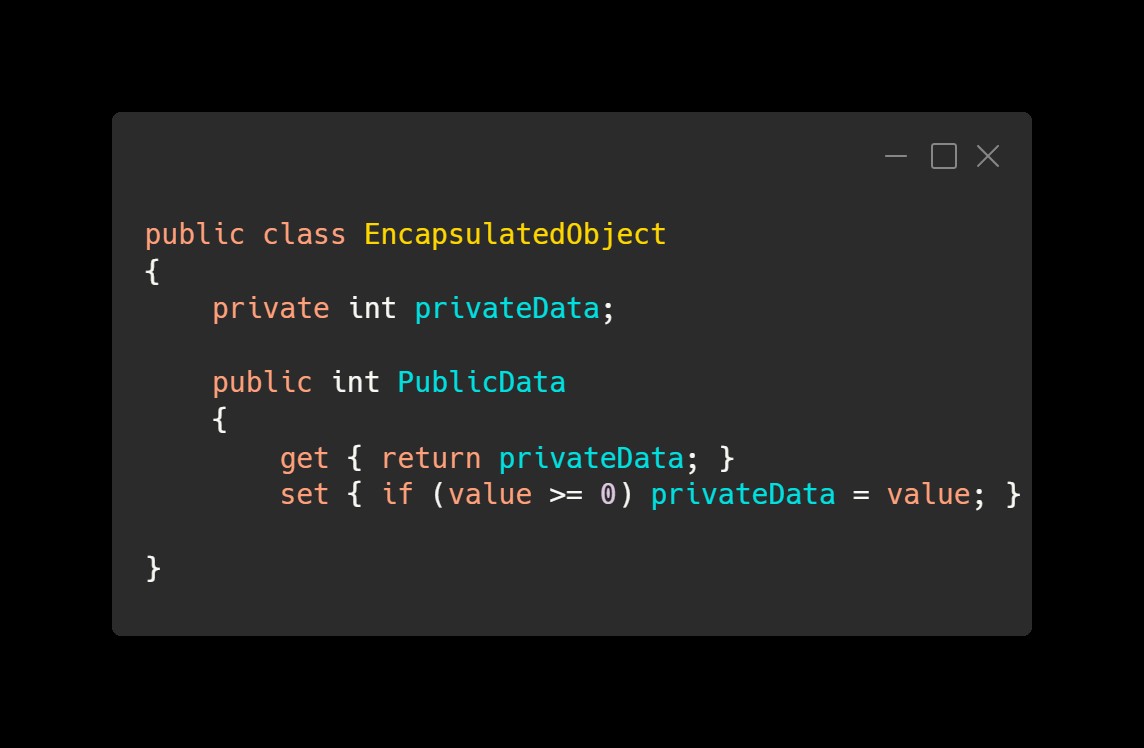
På samme måte følger denne klassen SRP ved å styre bevegelsen og tegningen av en partikkel.

Oppgave 2B

Innkapsling er et grunnleggende konsept innen objektorientert programmering (OOP). Det refererer til bunting av data, og metodene som opererer på disse dataene, til en enkelt enhet kalt et objekt. Det er mekanismen som begrenser direkte tilgang til objektet sine komponenter.

Innkapsling i C# kan oppnås ved å bruke tilgangsmodifikatorer:

1. Public
2. Private
3. Protected
4. Internal



Eksempel for innkapsling i C#.

I dette eksemplet er privateData et privat medlem, så det kan ikke nås direkte utenfor klassen. I stedet gis det en offentlig eiendom PublicData som tillater kontrollert tilgang til privateData.

Fordeler med innkapsling

**Dataskjuling**

Innkapsling skjuler den indre implementeringen av klassen, slik at brukeren ikke har innsyn i hvordan klassen lagrer verdier i variablene. Tilgang skjer kun gjennom accessorer, som deretter håndterer den private variabelen.

**Økt fleksibilitet**

Variablene i klassen kan enkelt gjøres skrivebeskyttede eller skrivebare basert på kravene. Ved å inkludere eller utelate set eller get aksessoren i egenskapen, tilpasses klassen fleksibelt etter behov.

**Gjenbrukbarhet**

Innkapsling forbedrer gjenbrukbarheten av koden og gjør den enklere å tilpasse når nye krav oppstår.

**Enhetstesting**

Kode som er innkapslet, er lettere å teste for enhetstesting.

StudentSøknad

Prosjektet har en effektiv innkapsling ved å bevare private til dataene og public til metodene for kontrollert tilgang og manipulasjon.

**StudentStatus**

Gir en public måte å få verdien av statusvariabelen som er private.

Det er ingen offentlig setter for denne egenskapen, noe som betyr at når et StudentStatus objekt er opprettet, kan ikke status endres direkte utenfor klassen.

**ChangeStatus**

Metoden gir en kontrollert måte å endre status på, med en sjekk for lovlig overgang mellom gjeldende og ny status

**HadStatus**

Metoden gir en måte å sjekke om en bestemt status har vært aktivert, uten å avsløre den interne listen over tidligere statuser.