Niklas Björk, WU21

Reflektioner inlämningsuppgift: Left To Do, 5 Yhp

Beskrivning av koden

Koden är skriven med språket C# som en konsolapplikation och körbar i .NET Core.
Koden består av en programklass, en menyklass (meny för terminal UI), en samlingsklass för listor med uppgifter som innehåll, en superklass (Task) som tre uppgiftsklasserna ärver från och till sist en klass för underuppgifter (SubTask) som checklistan grundar sig på. Totalt åtta olika klasser inkluderat programklassen. Klasserna, förutom programklassen är framtagna utifrån kravspecifikationen.
Substantiven har blivit klasser; en menyklass (ConsoleMenu), en samlingsklass för alla uppgifter (TaskCollection), eftersom det måste finnas något som hanterar alla uppgifter, samt fyra olika klasser för de definierade uppgifterna; tidlös uppgift (TimelessTask), tidsatt uppgift (DeadlineTask), uppgift med checklista (CheckListTask) samt checklistsuppgifterna (SubTask). Eftersom alla olika uppgifter har vissa delar gemensamt, det vill säga att de alla är uppgifter så skapade jag en abstrakt klass (Task) som de tre grunduppgifterna ärver egenskaper från (polymorfism). Egenskaperna i detta fallet är förklaring av uppgiften (taskInfo), om den är markerad som klar (done) och om den är arkiverad (archived).
Varje klass har sina egna metoder som just den klassen bör ha koll på. D.v.s. SubTask-klassen bör veta om den är klar/inte klar och vad uppgiften innebär. DeadlineTask har koll på sin deadline och precis som de två andra uppgifterna ärver metoder från superklassen (Task) för att ha koll på om den är avklarad, arkiverad samt sin uppgift (text). Uppgiften med en checklista (CheckListTask) har koll på sin checklista genom olika metoder.
Klassen (TaskCollection) är själva ”maskinklassen” som kaffemaskinen som fixar ihop kaffet från vatten och kaffe. I denna klassen finns listor som innehåller alla skapade uppgifter, metoder för att lägga till ny uppgift i lista, och även lägga till underuppgifter i uppgiften med en egen checklista. Den innehåller också metoder för att pricka av en lista som gjord, arkiverad samt kolla om den är gjord eller arkiverad. Det finns också metoder som kollar längd på listorna samt om alla underuppgifter är avklarade i checklist-uppgiften. Det är också denna metoden som det skapas en ny instans av när det ska göras en ny checklista. Skulle man i framtiden vilja skapa flera olika listor för olika personer i programmet, så görs det en ny instans av denna klassen.
Menyklassen (ConsoleMeny) är klassen som skapar UI för användaren. I denna klassen finns all kod som är specifik för Terminal UI. Klassen skapar olika menyer med switch-satser, håller dom igång med while-loopar. Klassen har metoder för olika menyval, men också metoder som återanvänder samma kod på flera olika ställen, så att inte lika kod behöver skrivas flera gånger om (DRY). Skulle man vilja använda ett annat UI så är det i princip bara denna klassen som behöver skrivas om.
I programklassen som körs i terminalen skapas bara en ny instans av klassen ConsoleMeny och sedan anropas metoden UserInterface() som innehåller och startar ConsoleUI.

Använda principer

I min lösning så använder jag den objektorienterade principen **arv**. Det gör jag genom att skapa en abstrakt klass (Task) som har gemensamma egenskaper för alla uppgifter och uppgifterna ärver då dessa egenskaper, och metoder utan att jag behöver skriva de igen i klassen. Med hjälp av ”: base” så använder den också konstruktorn i superklassen för att sätta informationen som skickas med som en parameter vid skapande av ett nytt objekt av klassen.
En annan objektorienterad princip som används är **inkapsling** (enkapsulation). Det gör jag genom att i klasserna hålla alla fält private, och istället ändra dem med publika metoder. Metoder som hämtar och sätter värden på variablerna. Detta gör att det inte går att förändra ett värde på alla instanser av en klass genom misstag, eftersom man bara kommer åt dem genom en metod och då bara förändrar värdet på just den instansen. Jag har använt mig av ”genvägen” {get; set;} på mina listor. Det är av två skäl; det ena är att jag vill visa att jag förstår hur de används. I de fallen jag använt den så är set; privat för att inte det ska gå att ändra, utan bara läsa. Det andra är att jag vill lära mig dem och hur det funkar och lite för att det var ett snabbare, lite finare och enklare sätt. Jag skulle likväl kunna gjort en metod som gjorde samma sak.
Jag lägger också de metoder som jag anser tillhör klassen i den specifika klassen. Det är den tidsatta uppgiftens uppgift att kunna meddela sin deadline, eller uppgiften med checklista att veta om alla SubTasks är ”done”, ändra dem till ”done” eller lägga till SubTasks. På detta sätt blir det enklare att hantera komplexitet. Jag binder inte hårt klasser till varandra, utan förändring av kod kan ske utan att den behöver påverka andra applikationer.
Polymorfism har jag också använt mig av. I detta fallet är det mycket sammankopplat med principen arv, som jag skrivit om ovan. Jag har inte hittat skäl till att ha många metoder med samma namn i samma klass men olika parametrar utan i detta fallet handlar det om att klasser ärver från en superklass. En metod som jag dock kör ”override” på är

Tester för klasser

Jag har lagt till två ytterligare tester förutom de tre som står i kravspecifikationen. Det ena testet **{CheckDaysToDeadlineForDeadlineTask()}** gör i princip det som det är döpt till. Eftersom de tre tidigare testerna kollar egenskaper som finns i grundklassen, alltså min superklass (Task) så vill jag med detta testet kolla den specifika egenskapen för just denna uppgiften, alltså att den innehåller ett deadline-datum. I ”Arrange” skapar jag en instans av TaskCollection och skriver innehållet i uppgiften som sätts till en variabel. Jag skapar också en variabel med datum som skickas med som parameter när uppgiften skapas samt en expected-variabel som innehåller förväntat antal dagar till deadline.
Under ”Act” lägger jag till uppgiften i min lista som finns i klassen ”TaskCollection”, så att jag använder mina befintliga klasser och listor när jag kör testet. Jag testar också min ToString() metod för

att hämta antal dagar till deadline. Det gör att jag behöver konvertera stringen till tal (int) för att kunna jämföra under ”Assert”. Med Assert.Equal jämför jag sedan de förväntade dagarna med de som jag får med metoden ToString(), och som jag senare måste använda i UI:t.
Testet är bra eftersom det kollar funktioner som jag behöver använda mig av när programmet ska fungera i verkligheten men ett par nackdelar. Det ena är att deadlinen förändras med tiden, så att det blir ”fail” dagen efter. Det andra är att ToString() behöver förändras lite, eftersom det i testet bara ska skrivas ut dagarna och i programmet en längre sträng. Det är inga problem om man vet om det, men om någon annan tittar på testet, så kanske det blir svårare. Det är en reflektion, och troligt skulle jag skriva om det, om det var till en kund.
För mitt andra test **{CheckIfAllSubtasksIsCheckedForCheckListTask()}**, går tankarna lika med det första testet. Jag vill göra ett test som testar den specifika egenskapen med en checklist-uppgift, alltså att det funkar att lägga till en checklista, och att den går att pricka av. Går den att pricka av, så måste det också finna uppgifter att checka av.
För denna skapar jag också en ny instans. Namn till en checklist-uppgift och namn till fyra (SubTask). En expected-bool som true, samt en bool-variabel som false och som ska ändras om alla SubTasks är done.
Under ”Act” så lägger jag till huvuduppgiften med hjälp av en metod i TaskCollektion-klassen samt lägger till fyra SubTask, också med hjälp av en metod i samma klass. Eftersom jag vet i vilket objekt som uppgifterna ska vara i så skriver jag in parametrarna som ska skickas med.
Jag ändrar sedan alla SubTasks till done, även här med en metod i TaskCollektion-klassen. Också här blir parametrarna fast inskrivna. Tredje delen under ”Act” är att kolla om alla SubTasks i en uppgift är ”done”, vilket också görs med en metod som returnerar true/false och sätts till samma bool som skapades false i början. Om SubTasks är ”done” så blir den true.
Under ”Assert” så jämförs de två bool-variablerna med Assert.Equal(). Är alla SubTasks ”done” så är båda true, och testet går igenom.
Båda de extra testen jag har gjort baserar sig på egenskaper som ska användas i programmet, samt med metoder och klasser som också används i det verkliga programmet. Det gör testerna både användbara, verkliga och nyttiga i utvecklingen av programmet.

När jag började utveckla koden så gjorde jag det utifrån testerna. Jag började skriva testerna med metoder som inte fanns. När testet sedan var skrivet, började jag uppifrån att skapa klasser och metoder utifrån testet. Det känns lite tidsödande i början när det kliar i fingrarna att börja med allt direkt, men jag upplevde det mycket positivt, givande, roligt och också tidsbesparande när jag sedan satte igång med hela uppgiften.