TDDD86 – Laboration #4

6 oktober 2020

I den här uppgiften får du skriva om en färdig version av det klassiska spelet "Robots" så att det använder polymorfism. Filerna du behöver för att komma igång finns som `labb4.tar.gz` på kurshemsidan.

Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 Lab 4 redovisning''` och en `git push`. Informera sedan din assistant genom att meddela honom/henne (använd @assistant_namn) i privata kanalen i labbgrupps teamet.

Polymorfa robotar

Den givna koden innehåller en implementation av det klassiska spelet "Robots". Kompilera spelet och spela det för att observera dess beteende. Nedanstående är ett utdrag från man-sidan för `robots` på 4.3BSD:

```
Robots pits you against evil robots, who are trying to kill you (which
is why they are evil).  Fortunately for you, even though they are evil,
they  are  not very bright and have a habit of bumping into each other,
thus destroying themselves.  In order to survive, you must get them  to
kill each other off, since you have no offensive weaponry.

Since  you  are  stuck without offensive weaponry, you are endowed with
one piece of defensive weaponry:  a  teleportation  device.   When  two
robots  run  into each other or a junk pile, they die.  If a robot runs
into you, you die.  When a robot dies, you get 10 points, and when  all
the  robots die, you start on the next field.  This keeps up until they
finally get you.
```

Använd tangenterna 'B', 'N', 'M', 'H', 'K', 'Y', 'U', 'I' för att förflytta dig. Ett tryck på 'J' gör att du aktivt står still en runda av spelet, vilket ger dig mer poäng om robotarna krockar. 'T' teleporterar dig till en slumpvis utvald position.

Bekanta dig sedan med koden. Titta åtminstone på klasserna `Unit`, `GameState` och `MainWindow`.

Lägg till polymorfism

Din uppgift är att utnyttja polymorfism i Robots-programmet. Det finns flera ställen där polymorfism kan vara lämpligt i det här programmet.

- Alla enheter på spelplanen behöver kunna rita ut sig, men alla gör det på olika sätt.
- Vektorerna med robotar och skräp kan egentligen lagras som en enda vektor med saker på spelbrädet. Vi kan säga att skräp "är en" robot som bara råkat sluta röra på sig. För att inte komplicera saker i onödan kan vi låta resultatet av två kolliderande robotar vara två skräphögar på platsen.

Vi föreslår följande arbetsgång:

1. Gör `draw()`, `moveTowards()` och `attacks()` virtuella i klassen `Unit`. Du kommer att behöva lägga till en virtuell åtkomstfunktion `draw()` i `Unit.cpp` (eller till och med i `Unit.h`) som inte gör någonting. Kompilera och testkör.
2. Låt `Junk` vara en subklass till `Robot`. Överskugga metoderna `moveTowards()` och `attacks()` så att de beter sig rätt för `Junk`. (Du kommer att behöva lägga till en konstruktor i `Robot`-klassen som anropas av konstruktorn i `Junk`). Kompilera och testkör.
3. Lägg till en virtuell funktion `unsigned getCollisionCount()` i `Robot`-klassen. Denna funktion ska returnera värdet `1`. Uppdatera även subklassen `Junk` så att funktionen returnerar värdet `0` för `Junk`. Kompilera och testkör.
4. Det behövs en hel del ändringar i klassen `GameState` så att bara en `vector<Robot*>` lagras. Genomför ändringarna. De mest ingående förändringarna krävs i konstruktorn och i `countCollisions`. Observera att det finns junk på en plats om det är två eller mer robotar där (oavsett vilken typ av robotar).
5. Gå igenom `GameState` metod för metod och se till att alla ändringar som krävs är genomförda. Kompilera och testkör.

6. Oturligt nog har ditt program nu fått en *minnesläcka*. Många robotar skapas, men de förstörs aldrig. Om en spelare skötte sig riktigt, riktigt bra, och spelade i timmar, skulle ditt program till sist få slut på minne och bli superslött.
   Om du vill kan du använda programmet Valgrind för att verifiera att du har en minnesläcka. (I Qt creator väljer du "Valgrind Memory Analyzer" från Analyze-menyn.)
7. I både `GameState(int numberOfRobots)`-konstruktorn och `countCollisions()`-metoden bör du kunna placera anrop till `delete` för att lämna åter några robotar till heapen. Gör detta.
8. Ovanstående åtgärd löser inte minnesläckan, men reducerar den. För att lösa den fullständigt behöver du skriva en lämplig kopieringskonstruktor, destruktor och `operator=` för klassen `GameState`. Gör detta med hjälp av polymorfisk clone method för `Junk`, `Robot` och `Unit`.
9. Testa dessa metoder igenom att använda valgrind. Ersätt först `gameState.moveRobots();` i början av `processMove` i `mainwindow.cpp` med `GameState copy=gameState; copy.moveRobots(); gameState=copy;`.

Möjliga utökningar

E6 — abstrakta klasser (2 poäng):
En abstrakt klass deklarerar virtuella funktioner men definierar dem inte. Den virtuella funktionsdeklarationen behöver inkludera den udda syntaxen `= 0` som följer:

```
virtual void draw() = 0; // Deklaration i Unit.h
```

Detta säger "Jag tänker inte skapa några instanser av klassen `Unit`. Subklasser måste definiera metoden `draw()`." Det går fortfarande att överföra `Unit` som parameter via referens eller via pekare.

1. Gör `Unit`-klassen abstrakt genom att göra de ändringar du finner lämpliga. Du kan behöva lägga till en ny klass eftersom användarens inmatning i nuläget överförs som en `Unit`.
2. Rensa up koden lite. Viss funktionalitet i `Unit`-klassen hör egentligen hemma i `Robot`-klassen. Flytta de metoder som bara robotar använder till `Robot`-klassen. (Kom ihåg att `hero` använder `moveTowards`-metoden.)

Skapa en ny branch som heter E6. Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 E6 redovisning''` och en `git push`. Skicka sedan ett mail till din assistent med ämnet: `[TDDD86] E6 redovisning`.