TDDD86 – Laboration #4

28 september 2023

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 `labb4.2023.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 maila honom/henne.

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, tilsammans, 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. Lägg till om det behövs, och gör `isAlive()`, `doCrash()`, `isToBeJunked()`, `draw(QGraphicsScene*)` och `moveTowards()` virtuella i klassen `Unit`. Du kan välja ett lämligt beteende för `Unit`, e.g., `isAlive()` och `isToBeJunked()` returnerar `false` medan `doCrash()`, `draw(QGraphicsScene*)` och `moveTowards()` gör ingenting. Din kod borde kunna kompileras och testköras.
2. Överskugga i `Junk` metoderna `moveTowards()`, `doCrash()`, `isToBeJunked()`, och `isAlive()` så att de beter sig rätt för `Junk`. Notera att `Junk` svarar alltid `false` till `isToBeJunked()`. Din kod borde kunna kompileras och testköras.
3. Det behövs en hel del ändringar i klassen `GameState` så att enbart en `vector<Unit*> units` behövs. Denna vector lagrar pekare till både Robots och Junk. Genomför ändringarna. De mest ingående förändringarna krävs i konstruktorn, `updateCrashes()`, och `junkTheCrashed()`.
4. Gå igenom `GameState` metod för metod och se till att alla ändringar som krävs är genomförda. Din kod borde kunna kompileras och testköras.
5. 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.

Använd programmet Valgrind för att verifiera om du har en minnesläcka. (I Qt creator väljer du "Valgrind Memory Analyzer" från Analyze-menyn, här behöver du inte ta hansyn till de eventuella externa errors som Valgrind kan rapportera).
6. Placera anrop till `delete` i `GameState.cpp` för att lämna åter några robotar till heapen.
7. För att lösa problemet med minnesläckan behöver du skriva en lämplig kopieringskonstruktor, destruktor och `operator=` för klassen `GameState`. Gör detta med hjälp av polymorfisk clone method för `Unit`.
8. Testa dessa metoder med hjälp av Valgrind medan du får din "hero" att gå igenom en nivå.

Möjliga utökningar

E6 — abstrakta klasser, flytta semantik (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;
virtual Unit* clone() const = 0;
virtual bool isAlive() const = 0;
virtual void doCrash() = 0;
virtual bool isToBeJunked() const = 0;
```

Detta säger, när dessa deklarationer läggs till filen `Unit.h`, att "Jag tänker inte skapa några instanser av klassen `Unit`. Subklasser måste definiera dessa metoder. Det går fortfarande att överföra `Unit` som parameter via referens eller via pekare.

- Gör `Unit`-klassen abstrakt genom att deklarera dessa metoder som abstrakta metoder.

Flytta semantik (move semantics) möjliggör att resurser överförs från tillfälliga objekt. Detta kan göras i en "move constructor" vid skapandet av objektet eller i en "move assignement operator" vid tilldelning.

```
GameState(GameState&&);
GameState& operator=(GameState&&);
```

- Lägg en "move constructor" och en "move assignment operator" till din `GameState`. Kolla om/var de anroppas i koden.

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`.