Lek med geometri

Skapande av interaktiva geometriska figurer på en JPanel

av

Kristoffer Danbrant

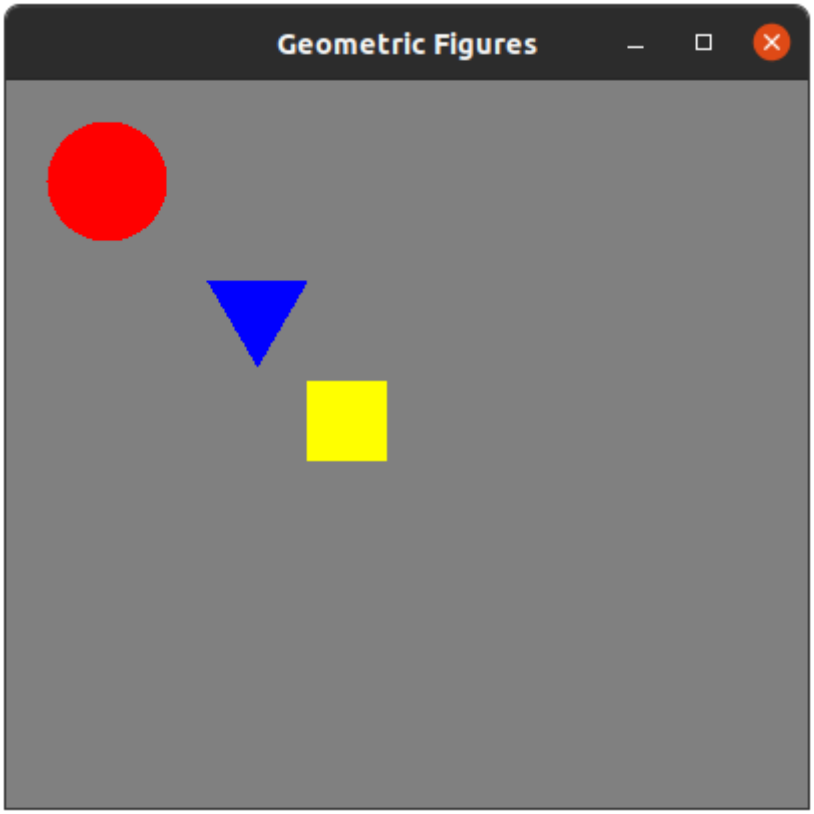

Figure 1: Skärmbild ifrån programmet

Department of Information Technology

Uppsala Universitet

mars 2023

Innehållsförteckning

Sammanfattning

Programmet "Lek med geometri" demonstrerar kraften i objektorienterad programmering med hjälp av Java genom att tillåta användaren att manipulera geometriska figurer på en panel. Denna rapport utforskar grundläggande koncept inom objektorienterad programmering, som klasser, objekt, arv och polymorfism och visar hur de används för att skapa programmet. Rapporten diskuterar också hur man utformar och implementerar klasser för geometriska figurer, inklusive användning av gränssnitt för att definiera gemensamt beteende och hur man använder samlingar för att hantera flera objekt. Dessutom täcker rapporten viktiga programmeringsprinciper såsom inkapsling, abstraktion och arv. Slutligen avslutas rapporten med en diskussion om hur man kan utöka programmet genom att lägga till nya geometriska figurer och anpassa deras beteende. Sammantaget ger programmet "Lek med geometri" en engagerande och praktisk introduktion till objektorienterad programmering med hjälp av Java.

Introduktion

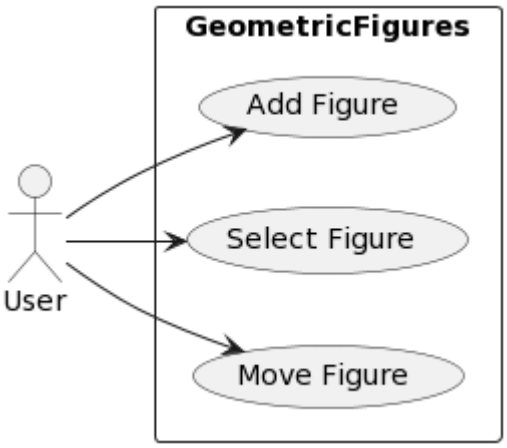

Figure 2: Use case diagram

Programmet "Lek med geometri" är en enkel men kraftfull demonstration av objektorienterad programmering med hjälp av Java. Det tillåter användaren att skapa, välja och flytta geometriska figurer på en panel med hjälp av musen. Programmet är organiserat i flera klasser som demonstrerar viktiga koncept inom objektorienterad programmering, såsom inkapsling, arv och polymorfism. Det är dock viktigt att notera att programmet för närvarande endast har tre fördefinierade geometriska figurer och att användaren inte kan lägga till eller ta bort figurer vid detta tillfälle. Detta kan begränsa användarens interaktion med programmet och begränsa dess användbarhet i verkliga scenarier. För att göra programmet mer användbart och anpassningsbart, skulle det vara nödvändigt att implementera funktionalitet som gör det möjligt för användaren att lägga till eller ta bort geometriska figurer, till exempel genom att använda en meny eller dialogruta.

Java och IDE-information

Implementeringen av programmet "Lek med geometri" utvecklades med hjälp av Java-programmeringsspråket version 11.0.18. Javac-kompilatorversionen som användes var också 11.0.18. IDE:n som användes för att utveckla programmet var Visual Studio Code version 1.76.2, som kördes på en Linux-maskin med Ubuntu 20.04.6 LTS.
Användningen av en IDE för utveckling av programmet möjliggjorde enklare kodorganisation och underhåll. Funktioner som kodkomplettering och felsökningsverktyg möjliggjorde en mer effektiv arbetsflöde och snabbare identifiering och lösningsförslag på fel. Dessutom bidrog förmågan att enkelt navigera genom kodbasen och spåra ändringar med versionshanteringssystem som Git till en mer strömlinjeformad utvecklingsprocess.

Användningen av en IDE tillät också enkel integrering av externa bibliotek som Swing-biblioteket för att skapa den grafiska användargränssnittet. Förmågan att snabbt importera och använda externa bibliotek utan att manuellt hantera beroenden möjliggjorde en mer effektiv utvecklingsprocess.

Översikt över klasser och metoder

Programmet "Lek med geometri" innehåller flera klasser som samverkar för att skapa en rolig och interaktiv applikation för att manipulera geometriska figurer.

GeometricFiguresPanel: Denna klass är en JPanel som innehåller de geometriska figurerna. Den har tre instansvariabler: figurer (en lista över Figure-objekt), valdFigur (den för närvarande valda

figuren) och musposition (den aktuella positionen för musen på panelen). Den tillhandahåller metoder för att lägga till nya figurer i listan, välja en figur på en specifik punkt och måla panelen och alla dess figurer.

Figure: Detta är en abstrakt klass som representerar en generisk geometrisk figur. Den definierar metoderna som alla geometriska figurer ska implementera, såsom paint (), contains () och moveBy (), och ger standardimplementationer för dem. Den inkluderar också instansvariabler för figurens färg och position.
Circle, Triangle och Square: Dessa klasser utökar Figure-klassen och representerar specifika geometriska figurer. Var och en har sina egna unika egenskaper, som radie för Circle, hörn för Triangle och sidolängd för Square. De överskriver metoderna som definieras i Figure-klassen för att implementera deras specifika beteende.

GeometricFigures: GeometricFigures: Denna klass representerar panelen som innehåller de geometriska figurerna. Den innehåller logiken för att lägga till figurer på panelen, välja figurer och uppdatera positionerna för figurerna.

Circle: Denna klass representerar en cirkel. Den innehåller egenskaperna för en cirkel (x, y, radie och färg) och logiken för att måla, kontrollera om en punkt är inuti cirkeln och flytta cirkeln.

Figure: Denna interface representerar en generisk geometrisk figur. Den specificerar metoderna som alla geometriska figurer ska implementera, såsom målning, kontroll om en punkt är inuti figuren och flytt av figuren. moving the figure.

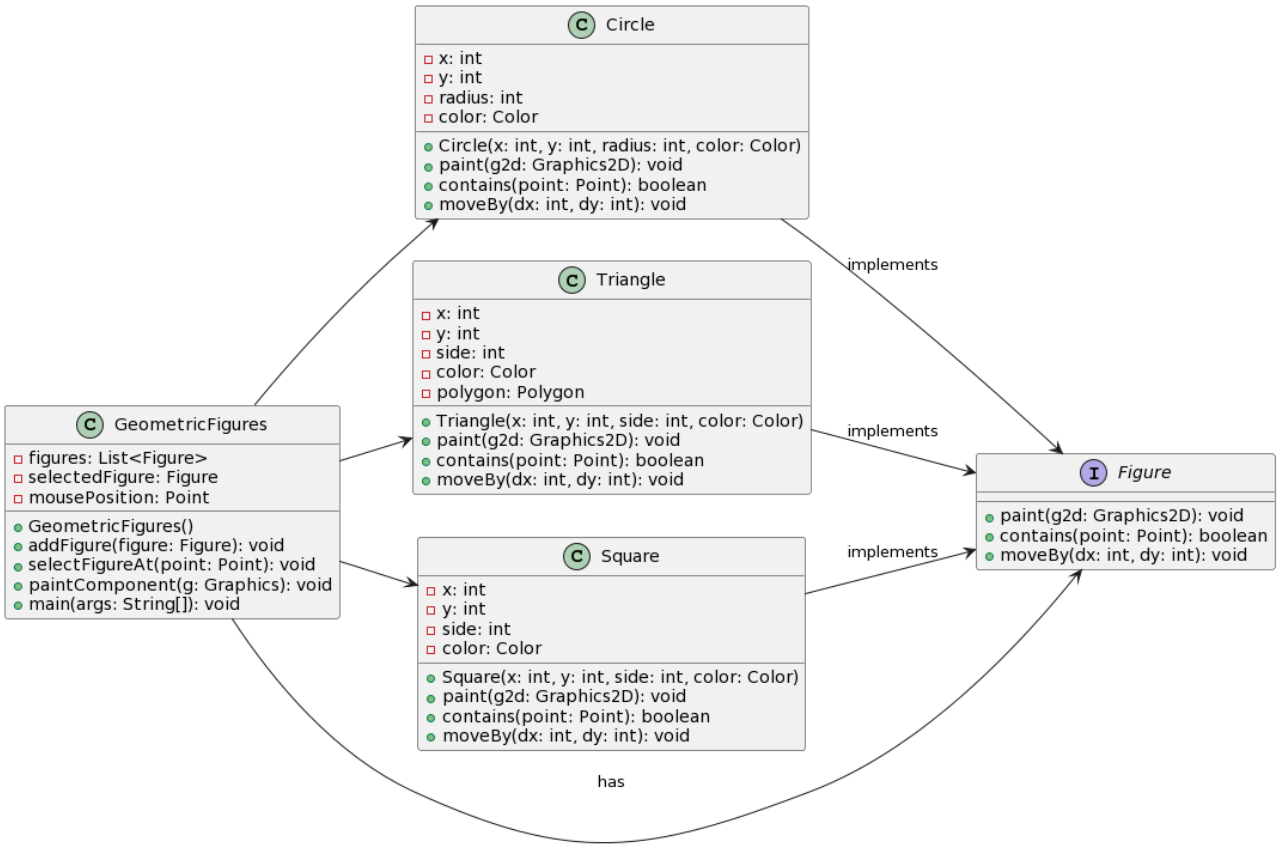

Figure 3: UML-diagram

Designbeslut och motiveringar

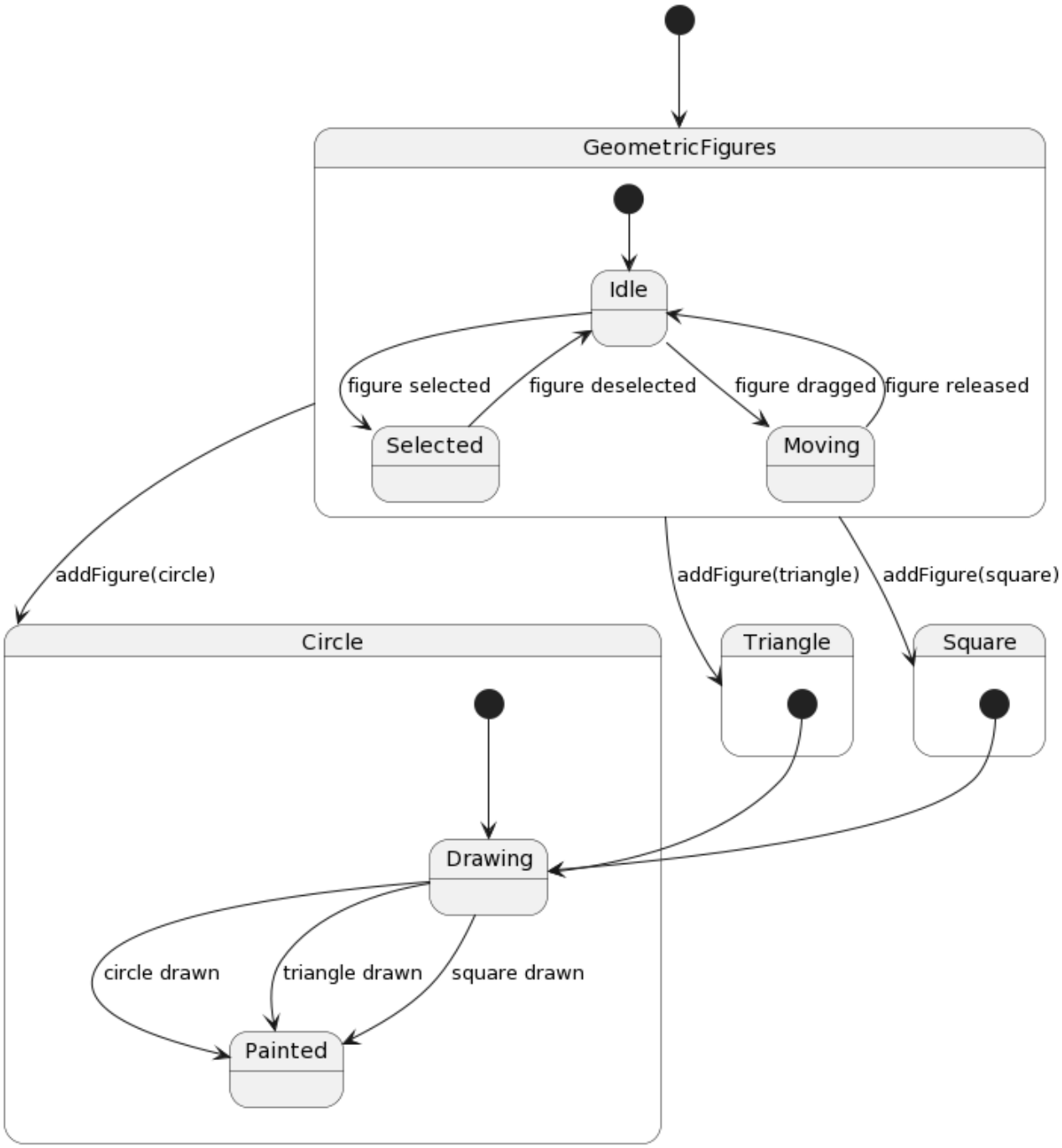

Figure 4: State-diagram

Programmet följer goda designprinciper som demonstrerar viktiga koncept inom objektorienterad programmering. Separeringen av geometriska figurer i sina egna klasser främjar lätt utbyggnad och modularitet. Dessutom definierar användningen av en abstrakt klass gemensam beteende för alla figurer samtidigt som specifika implementationer tillåts i subklasser. Användningen av samlingar för att hantera flera objekt och gränssnitt för att definiera gemensamt beteende över olika typer av objekt ökar ytterligare programmets flexibilitet.

Brister i koden

Även om programmet är funktionellt finns det flera problem som kan åtgärdas för att förbättra dess prestanda och tillförlitlighet. Till exempel:

- Brister i modularitet: Programmets brist på modularitet gör det svårläst och svårt att underhålla. I ett live-projekt skulle det starkt rekommenderas att bryta ned koden i mindre, hanterbara funktioner och lägga till kommentarer för att förklara vad varje del av koden gör. Men eftersom jag var under tidsbegränsningar på grund av andra skoluppgifter och åtaganden valde jag att gå för ett minimalt fungerande produkt.
- Felhantering: Programmet hanterar inte fel eller undantag, vilket kan leda till oväntat beteende eller krascher. För att förhindra dessa problem rekommenderas det starkt att lägga till felhantering och undantagshantering i ett live-projekt.
- Optimering: Programmet använder inte några optimeringstekniker, vilket kan påverka dess prestanda på större datamängder. Även om detta inte är ett problem för den nuvarande implementationen som bara hanterar tre figurer, rekommenderas det att utforska sätt att optimera programmet för att hantera ett större antal figurer.

Generella förbättraingar

För att adressera bristen på modularitet kan vi använda gränssnitt för att definiera gemensamt beteende över olika typer av objekt och förbättra programmet flexibilitet. Vi kan också använda arv för att skapa nya klasser som ärver egenskaper och beteenden från befintliga klasser, vilket skapar mer effektiv och återanvändbar kod samtidigt som vi undviker alltför komplexa och tätt sammankopplade klasshierarkier. Dessutom kan vi använda abstrakta metoder i Figure-klassen för att tvinga underklasser att implementera vissa beteenden som är unika för varje typ av geometrisk figur, vilket möjliggör variation i beteende samtidigt som en gemensam gränssnitt definieras för alla geometriska figurer. Genom att lägga till felhantering och hantering av undantag kan vi göra programmet mer robust och motståndskraftigt mot fel, vilket förhindrar oväntat beteende eller krascher. Vi kan också tillämpa designmönster som observatörsmodellen för att uppdatera panelen när en figur flyttas och fabriksmönstret för att skapa nya geometriska figurer, vilket förbättrar programdesignen och funktionaliteten. Slutligen kan vi utforska sätt att optimera programmet för större datamängder, som att använda datastrukturer som träd eller hash-tabeller för att lagra geometriska figurer eller bara måla de figurer som för närvarande är synliga på skärmen för att förbättra dess prestanda.

Abstrakta metoder och polymorfism

Abstrakta metoder är metoder som deklareras i en superklass men inte har någon implementation. Vi kan använda abstrakta metoder i Figure-klassen för att tvinga underklasser att implementera vissa beteenden som är unika för varje typ av geometrisk figur. Detta gör det möjligt för oss att definiera en gemensam gränssnitt för alla geometriska figurer samtidigt som vi tillåter variation i beteende. Polymorfism är förmågan hos ett objekt att anta olika former. I Java kan vi uppnå polymorfism genom metodsökning och metodöverbelastning. Genom att använda abstrakta metoder

i Figure-klassen och överlagra dem i Circle-, Square- och Triangle-underklasserna kan vi demonstrera polymorfism i vårt program.

Vi kan förbättra programmodulariteten och flexibiliteten genom att använda gränssnitt. Gränssnitt definierar en uppsättning beteenden som en klass måste implementera, vilket gör det möjligt för oss att definiera gemensamt beteende över olika typer av objekt. Till exempel kan vi definiera ett gränssnitt som heter Movable som har en moveBy() metod. Gränssnitten definierar en uppsättning beteenden som en klass måste implementera, vilket gör det möjligt för oss att definiera gemensamma beteenden över olika typer av objekt. Till exempel kan vi definiera ett rörligt gränssnitt som har en moveBy () -metod. Detta skulle tillåta oss att skapa ett mer modulärt och utbyggbart program som enkelt kan rymma nya typer av objekt med olika beteenden.

Arv

Arv gör det möjligt för oss att skapa nya klasser som ärver egenskaper och beteenden från befintliga klasser. Genom att använda arv kan vi skapa mer effektiv och återanvändbar kod. Det är dock viktigt att använda arv klokt för att undvika överdrivet komplexa och tätt kopplade klasshierarkier.

Fel- och undantagshantering

Felhantering och undantagshantering är viktiga för att se till att programmet beter sig som förväntat och inte kraschar oväntat. Till exempel kan om användaren försöker flytta en figur utanför panelen, kan detta resultera i oväntat beteende eller en krasch. Genom att lägga till felhantering och undantagshantering kan vi hantera denna situation och göra programmet mer robust och motståndskraftigt mot fel. Andra potentiella fel som kan uppstå i programmet inkluderar null pekar undantag, array index utanför gränserna undantag och aritmetiska fel. Genom att förutse dessa fel och lägga till lämplig felhantering och undantagshantering kan vi skapa ett mer pålitligt program.

Designmönster

Designmönster är återanvändbara lösningar på vanliga programmeringsproblem som hjälper till att förbättra kodkvalitet och underhållbarhet. Genom att tillämpa olika designmönster, såsom fabriksmönstret för att skapa nya geometriska figurer, observermönstret för att uppdatera panelen när en figur flyttas och dekoratörmönstret för att lägga till ytterligare egenskaper och beteenden till figurer, kan vi förbättra vårt programs design och funktionalitet.

Användning av fabriksmönstret: Fabriksmönstret är ett skapelsesdesignmönster som tillåter skapandet av objekt utan att specificera deras konkreta klasser. I sammanhanget av "Play with Geometry"-programmet kan vi använda fabriksmönstret för att skapa nya geometriska figurer utan att behöva känna till deras specifika implementation.

För att illustrera hur användningen av gränssnitt förbättrade programmodulariteten och flexibiliteten kan vi överväga exemplet med det rörliga gränssnittet. Genom att definiera ett Movable-gränssnitt med en moveBy()-metod kan vi säkerställa att alla klasser som implementerar detta gränssnitt kan

flyttas på panelen. Detta innebär att vi enkelt kan lägga till nya typer av objekt på panelen som har olika rörelsebeteende utan att behöva ändra den befintliga koden. Till exempel kan vi lägga till en Text-klass som implementerar det rörliga gränssnittet och rör sig horisontellt över panelen. Detta skulle vara enkelt att göra genom att skapa en ny klass som implementerar gränssnittet och lägga till den i listan över rörliga objekt på panelen.

Användning av observatörsdesignmönstret: Observatörsdesignmönstret är ett beteendemönster som tillåter objekt att meddelas när en annan objekts tillstånd förändras. I sammanhanget av "Play with Geometry"-programmet kan vi använda observatörsdesignmönstret för att uppdatera panelen när en figur flyttas.

Optimering

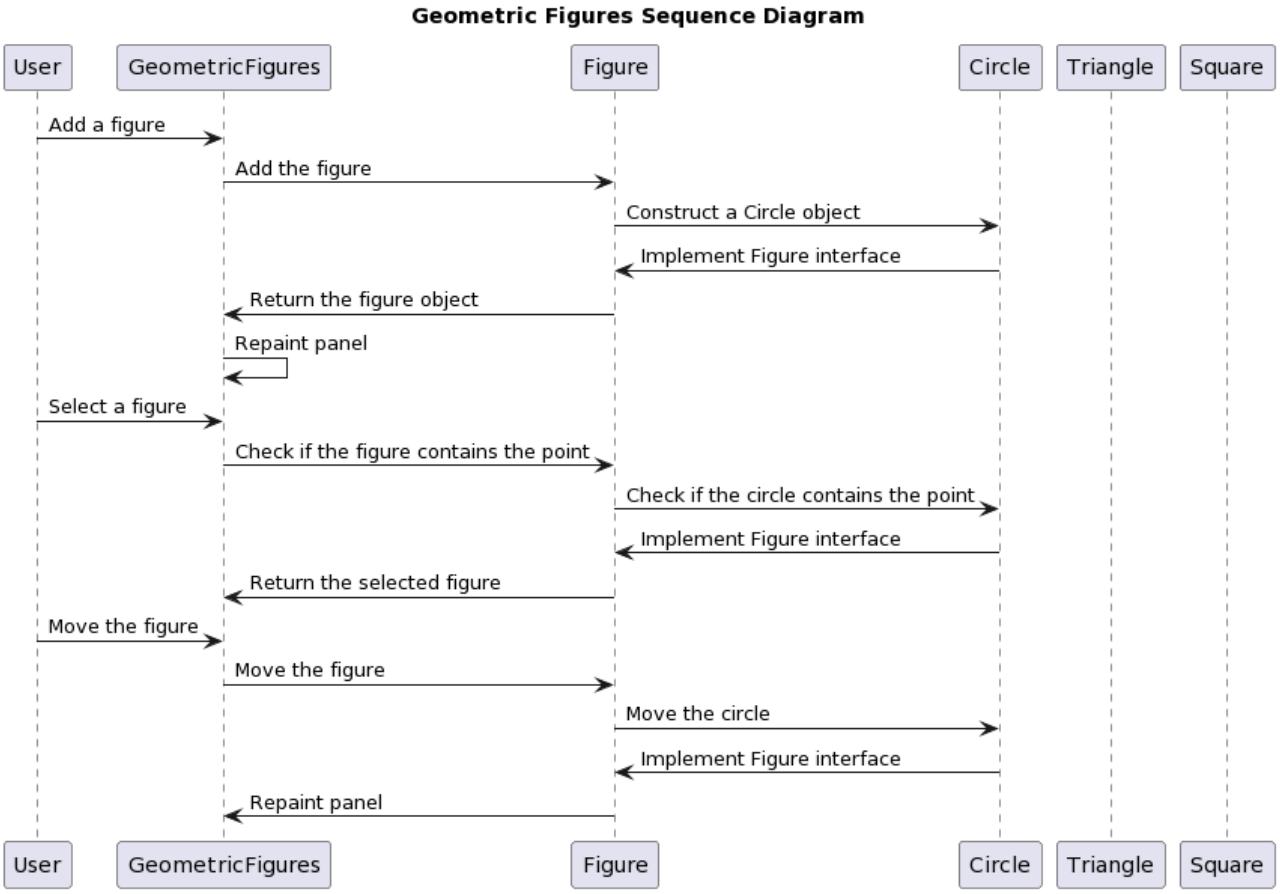

Figure 5: Sekvensdiagram

För att optimera programmet för större datamängder kan vi överväga att använda datastrukturer som träd eller hash-tabeller för att lagra geometriska figurer. Eftersom detta dock är en del av mer avancerade kurser beslutade jag att inte implementera dem. Dessa datastrukturer skulle tillåta oss att utföra operationer såsom sökning och sortering mer effektivt än att använda en enkel lista.
Dessutom kan vi utforska sätt att optimera panelmålningen genom att bara måla de figurer som för närvarande är synliga på skärmen, istället för att måla alla figurer varje gång panelen uppdateras.

Slutsats och slutgiltiga tankar

Slutligen är "Play with Geometry"-programmet ett utmärkt exempel på hur objektorienterade programmeringskoncept som arv, abstraktion och polymorfism kan användas för att skapa en flexibel och utbyggbar applikation. Genom att designa klasser för geometriska figurer, använda abstrakta metoder och gränssnitt för att definiera gemensamt beteende och tillämpa designmönster som observer-mönstret har vi skapat ett program som enkelt kan rymma nya typer av objekt och funktionalitet.

Det finns dock flera områden som kan förbättras i programmet. Till exempel kan vi ytterligare optimera programmet genom att utforska tekniker såsom datastrukturer för att lagra figurerna och att bara måla synliga figurer. Dessutom kan vi implementera felhantering och undantagshantering för att göra programmet mer tillförlitligt och robust.

Framåt kan vi också utforska mer avancerade ämnen såsom generiska datatyper, lambda-uttryck och strömmar för att ytterligare förbättra programfunktionaliteten och prestandan. Sammantaget demonstrerar "Play with Geometry"-programmet kraften i objektorienterade programmeringskoncept för att skapa en engagerande och praktisk applikation.