MooGame Projektet

Vi påbörjade projektet med att stegvis refaktorisera koden för att följa principerna i Clean Code (Martin Fowler, 2008). Målet har varit att förbättra läsbarhet, testbarhet och underhållbarhet, samtidigt som ansvar fördelats tydligare mellan klasser och komponenter.

Första steget var att förbättra namngivningen på alla variabler och vissa metoder så att det blev lättare för hjärnan att förstå vad programmet faktiskt gjorde. Sen kommenterade vi en massa för att få en övergripande förståelse över vad applikationen faktiskt gjorde för att kunna planera hur applikationen ska se ut.

Andra steget var att skapa en mental modell av hur appen ska se ut. Vi ritade upp på drawio våran första skiss över hur appen ska se ut (se källförteckningen).

Sen började vi refaktorisera, att flytta ut saker vi ansåg var sin egen del var en bra början, för då gick vi åt “single responsibility principle” (Clean Code, Martin Fowler, 2008).
Medan vi refaktoriserade koden så försökte vi att jobba enligt TDD (https://martinfowler.com/bliki/TestDrivenDevelopment.html), men insåg rätt snabbt att det var för mycket på en gång - att lära sig hur man skriver tester men också hur man skriver bra kod. Vi gav upp på det och började med att skriva kod först och skrev tester på det senare.

Det som blev tydligt i programmet var att de olika delarna hade sitt eget ansvar. Utmaningen låg i att få delarna att samarbeta utan att bli hårdkopplade till varandra. En central princip i Clean Code är att klasser ska vara små, ha ett enda ansvar, och metoder ska vara korta och utföra en enda uppgift (Martin, 2008, s. 34–35, 136, 139).

För att uppnå detta började vi skapa interfaces. Vi visste redan tidigt att flera (egentligen alla) delar behövde testas, och för att kunna testa dem korrekt måste de vara separerade från varandra. I början förstod vi inte riktigt varför detta var viktigt. Det blev dock tydligt senare i projektet när vissa klasser var hårdkopplade till GameController. Då krävdes det att betydligt fler objekt behövde instansieras enbart för att kunna testa GameController.

Genom att lösa upp beroendena blev det mycket enklare: vi kunde skriva ett test, få det att gå igenom och sedan lämna den klassen tills en förändring faktiskt krävdes.

En utmaning jag (Alexander) stötte på var Cargo Cult-antipatternen, där jag gjorde många förändringar utan att förstå varför (Wikipedia, 2023). Jag följde lärarens och klasskamraternas exempel utan att riktigt reflektera, vilket ledde till att koden fick en felaktig struktur.

Efter viss vägledning från min lärare blev det tydligare att tester är helt avgörande för ett välstrukturerat program. Genom tester insåg jag(Alexander) varför det var nödvändigt att använda interfaces på rätt ställen, och hur viktigt det var att ta bort kod som inte hörde hemma i en viss klass.

När vi refaktoriserade fokuserade vi på att ge meningsfulla namn till variabler och metoder, så att de tydligt beskrev vad de var och vad de gjorde. Vi arbetade med att minska storleken på metoderna så långt det gick och såg till att de utförde det de lovade genom sina namn.

På så sätt kunde vi eliminera onödiga kommentarer och låta koden tala för sig själv, vilket också är en rekommendation i Clean Code (Martin, 2008, s. 55). Målet var att koden skulle vara lättläst, utan onödig textmassa i en enskild klass eller metod.

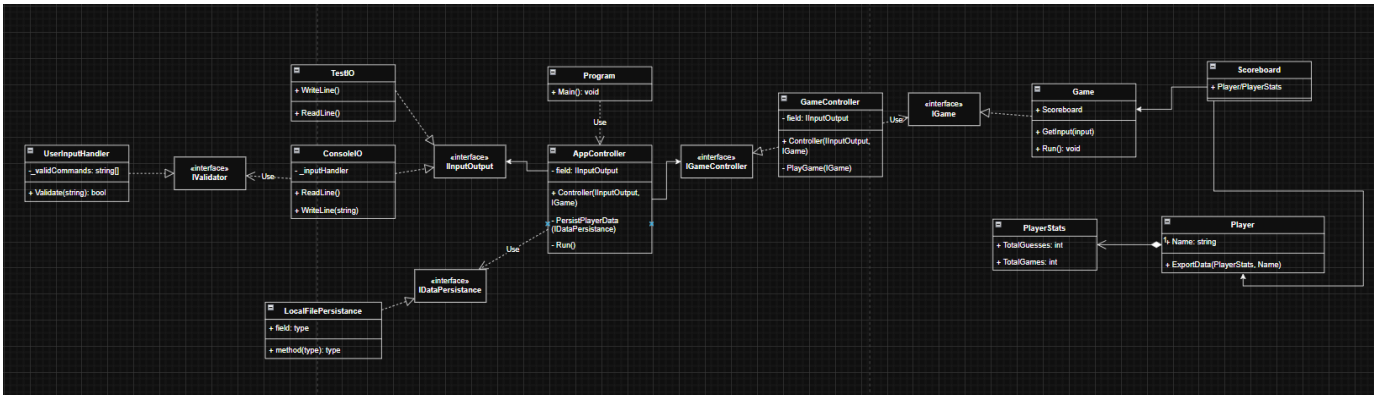
En aspekt jag (Alexander) känner att jag behöver arbeta mer med framöver är felhantering, något som Martin också betonar som en central del av ren kod (2008, s. 103–112).

I UserInputHandler har vi enkapsulerat ett mellanlager mellan våran View och övriga delar av programmet för att möjliggöra att vi kan implementera validerings logik för input samt städa den input vi får separat från vårt användargränssnitt. Vi har också skapat den av anledningen att ha ett gränssnitt för övriga delar av koden så att de inte behöver känna till konsolens existens och på så sätt också följa “Separation of Concern” principen.

I metoden GetInput() har vi använt oss utav strategy pattern(https://refactoring.guru/design-patterns/strategy) för att möjligheten att vilket spel som helst ska kunna skicka in sin egen validerings logic för input från spelaren genom våran UserInputHandler och på samma sätt göra hela klassen mer flexibel genom att använda samma teknik för att implementera grundläggande metoder i UserInputHandler som GetYesNo().

I Scoreboard har vi använt oss av Dependency Inversion Principle () genom att injicera ett IUserInputHandler. Detta gör att klassen blir lättare att testa och mindre beroende av den underliggande infrastrukturen.
I metoder som WriteResult() och GetPlayers() har vi implementerat guard clauses för att tidigt fånga felaktiga parametrar, vilket följer principen “fail fast”(https://www.agile-academy.com/en/agile-dictionary/fail-fast/) och gör koden mer robust och lättare att förstå.
Genom att bryta ut logik i små hjälpmetoder som EscapeCsv() och SplitCsv() följer vi Single Responsibility Principle och får kod som är mer sammanhållen, enkel att läsa och enkel att enhetstesta.
Vid formateringen i Print() har vi separerat ansvaret för att hämta och beräkna data från själva presentationen, vilket liknar ett Presentermönster (https://samvera.github.io/patterns-presenters.html). Detta gör koden tydligare och skapar en renare separation mellan datahantering och visning.

Drawio ritningen:



Källförteckning

- Martin, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008

- Wikipedia. “Cargo cult programming.” https://en.wikipedia.org/wiki/Cargo_cult_programming

- Microsoft Docs. “Dependency injection in .NET.” https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection