

Informatica Werktuigen

Visual Basic .NET - Oefenzitting 2

1 Inleiding

In de vorige oefenzitting heb je leren werken met Visual Basic .NET en Visual Studio. Deze oefenzitting zal je op weg helpen in het debuggen van een Visual Basic-applicatie. Dit kan een grote hulp zijn wanneer je aan je practicum werkt en je broncode een fout bevat die je moeilijk kan terugvinden.

Bovendien zullen veel van de concepten die je hier aanleert nuttig zijn in je verdere studentenloopbaan. De besproken concepten worden immers bij het debuggen in andere ontwikkelomgevingen ook gebruikt. Eventuele fouten die je in andere projecten en programmeertalen tegenkomt kun je dus aan de hand van soortgelijke methodes opsporen.

2 Debugging

2.1 Wat is debugging?

Debugging is het proces van het systematisch opsporen van softwarematige fouten oftewel *bugs* in de broncode. Naargelang applicaties meer functionaliteit hebben zal de broncode sterk aangroeien. Hierdoor wordt het opsporen van deze bugs almaar moeilijker.

Daarom bieden de meeste (goede) ontwikkelomgevingen werktuigen aan om op een systematische manier deze bugs op te sporen. Dit maakt het veel eenvoudiger om snel de oorzaak van een softwarematige fout te onderzoeken.

Een bug kan vele redenen hebben, maar vaak worden ze veroorzaakt door een logische denkfout van de programmeur, een verkeerde operator of het verkeerd doorgeven van informatie aan berekeningsalgoritmen. Dankzij de werktuigen van de ontwikkelomgeving kunnen dit soort fouten (en veel andere!) sneller worden opgespoord en opgelost.

2.2 De opstelling

In deze oefenzitting gaan we gebruik maken van een bestaand project om jullie te laten kennismaken met het debugging-proces. Dit geeft jullie de kans om op een zinnigere manier op zoek te gaan naar bugs en leert jullie kennismaken met een (iets) groter softwareproject dan je gewoon bent uit de vorige oefenzitting.

Het is niet belangrijk dat jullie de broncode van dit project helemaal begrijpen - dit zou te veel tijd vergen en zou het doel van deze oefenzitting voorbij schieten. Wel wordt er verwacht dat je de werktuigen die Visual Studio je aanbiedt om fouten op te sporen onder de knie hebt en ze kunt gebruiken voor andere projecten.

Op Toledo vinden jullie de broncode van het project onder de naam **Memory.zip**. Het is een implementatie van het spel waarbij je plaatjes moet omdraaien en zo de paren met dezelfde afbeelding moet ontdekken en aanduiden.

Het spel bevat echter nog enkele bugs die wij er "per ongeluk" hebben ingelaten. Het is aan jou om deze fouten op te sporen aan de hand van de werktuigen die wij hier op de volgende pagina's zullen presenteren.

Openen van het project Om het project te openen, moet je eerst **Memory.zip** unzippen¹. Vervolgens start je Visual Studio op en druk je op **File** → **Open Project**. Ga naar de map waar je het project hebt geunzipd en selecteer het ***.vbproj**-bestand. Klik op **open** om het project te openen.

Structuur Wanneer je het project geopend hebt, kan je al wat op verkenning gaan naar hoe het programma opgebouwd werd. De formulieren van het programma zijn te vinden in *GameForm.vb*, *NewGameForm.vb* en *AboutBox.vb*. Daarnaast is er broncode om het algemene spel te sturen (*Game.vb*, *Player.vb*), de kaarten te tekenen (*GameGrid.vb*, *Grid.vb*, ...) en voor artificiële intelligentie (*ArtificialIntelligence.vb*, ...) voorzien. Verder vind je in *Configuration.vb* enkele speleigenschappen.

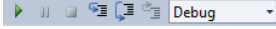
Over de inhoud van deze bronbestanden gaan we hier niet verder uitwijden. Het is immers niet de bedoeling van deze oefenzitting dat jullie op het einde de broncode volledig begrijpen. Bovendien zal een groot deel van de code ook duidelijker worden wanneer je aan het debuggen bent.

3 Debug build versus Release build

Je kan een programma in Visual Studio uitvoeren op twee manieren:

- **Release modus:** De ontwikkelomgeving zal zoveel mogelijk optimaliseren en zal geen rekening houden met alle instructies die specifiek zijn ontworpen om het debugproces te vereenvoudigen.
- **Debug modus:** Omdat optimalisatie het debuggen moeilijker maakt, zal de ontwikkelomgeving dit achterwege laten in deze modus. Er wordt ook rekening gehouden met de debuginformatie.

Standaard zal een applicatie die je in Visual Studio uitvoert (via de F5 of play-knop) in debug modus worden gecompileerd. Wil je toch een versie in release modus creëren, dan kan dit door naar **Build** → **Configuration Manager...** te gaan en de Active Solution op **Release** te zetten. Vervolgens kan je een **Build** → **Build Solution** doen. In de **bin\Release** submap van je project kan je nu de Release build vinden.

Voor deze oefenzitting zullen we echter voldoende hebben met de debugging werktuigen die worden aangeboden door Visual Studio. Om deze te gebruiken, moet je voor de **Debug** keuze gaan in de uitvoeringstoolbar (deze ziet er ongeveer als volgt uit: ) en kan je het programma gewoon met **F5** of de **play**-knop uitvoeren.

4 Go to definition / Find references

Wanneer je de broncode van een programma aan het bekijken bent, kom je soms oproepen van procedures en functies tegen die je niet kent. Om de code te bekijken van deze functies of procedures kan je gebruik maken van de menu-optie **Go to definition**. Hiervoor klik je met de rechtermuisknop op de functie of procedure die je wil bekijken.

Je kan ook het omgekeerde doen. Wanneer je de menu-optie **Find all references** kiest bij een functie- of procedureoproep, zal Visual Studio op zoek gaan naar alle oproepen² van diezelfde functie of procedure binnen de broncode.

Wil je nu terug navigeren naar het vorige stukje code, klik dan op de pijltjes (◀ ▶).

¹Het kan zijn dat je problemen ervaart als je in een PC-klas het project unzipd naar een locatie die geen subfolder is van **C:\Workdir**. In dat geval zal je wellicht het programma niet kunnen starten.

²Maar ook de definitie van de functie of procedure!

5 Verschil tussen breakpoints en compilatiefouten

Probeer nu het programma uit te voeren. Doe dit door op **F5** of **Debug** → **Start Debugging** te drukken. Je krijgt onmiddellijk de melding dat er **build-errors** waren. Dit wil zeggen dat Visual Studio niet in staat was de broncode te compileren. Dit noemt men ook wel een **compile-time error**. Als antwoord op de melding klik je op **No** (niet verder uitvoeren).

Dit type van fouten wordt doorgaans niet gezien als softwarebugs omdat wanneer een programma niet gecompileerd geraakt, de compiler je meestal wel kan wijzen naar de precieze plaats in de broncode waar het foutloopt. Dit kan je ook zien in de zogenaamde **error list** die Visual Studio beneden in het scherm toont³.

De lijst bevat één element, namelijk **'Randomiz' is not declared. It may be inaccessible due to its protection level**. Als je hierop dubbelklikt wordt je naar de exacte locatie in de code gebracht waar het fout gaat. De compiler geeft aan dat het geen procedure vond die **Randomiz** heet. Het gaat natuurlijk om een typfout. Je kan deze verbeteren door hier de oproep **Randomiz()** te vervangen door een oproep **Randomize()**.

Bugs zijn vaak veel minder makkelijk om op te sporen omdat het hier gaat om logische fouten waarbij de compiler niet op voorhand weet dat het gaat om een fout. De compiler kan broncode immers enkel omzetten in machinecode die uitvoerbaar is, maar kan onmogelijk weten wat de echte *bedoeling* is van het programma dat je hebt geprogrammeerd. We overlopen daarom in deze oefenzitting werktuigen die het mogelijk maken om deze bugs makkelijk op te sporen.

6 Breakpoints

Een eerste belangrijke concept aan debugging is het breakpoint. Een breakpoint is een locatie in de broncode, op een regel waar een instructie staat beschreven, en waar je de uitvoering van het programma wil pauzeren. Wanneer je in debugmodus het programma uitvoert, zal het programma onderbreken op het punt juist vóór deze instructie wordt uitgevoerd en word je naar de locatie van het breakpoint gebracht.

In de broncode kan je meerdere breakpoints plaatsen. Omdat het niet nuttig is om een breakpoint op een blanco regel te plaatsen, zal deze zich binden aan de eerst volgende instructie.

Het gebruik van breakpoints laat niet alleen toe om te controleren of een programma een bepaald gedeelte van de broncode uitvoert, maar kan ook gebruikt worden om de waarde van alle variabelen te controleren op het ogenblik van de onderbreking.

6.1 Een breakpoint plaatsen

Start nu terug het programma. We willen een spel spelen. Druk daarom binnen het programma op **F2** en op **Start**.

Waarschijnlijk zal je merken dat het spel niet wil beginnen en het dialoogvenster gewoon afsluit. Er gaat dus iets verkeerd in de broncode. We gaan dit proberen op te lossen door middel van breakpoints de precieze fout te achterhalen.

Het dialoogscherm dat we zojuist hebben opgestart is in de broncode geprogrammeerd als **NewGameForm**. We willen weten welke stukken code er worden uitgevoerd en welke niet. De methode **GetNewGameData** in de klasse **NewGameForm** zal normaal deze gegevens afhandelen en doorgeven zodat het spel kan worden opgebouwd. We willen weten of dit wel effectief gebeurt, aangezien er blijkbaar niets gegenereerd wordt.

Plaats een breakpoint op de eerste instructie van de methode **NewGameForm.GetNewGameData()** (dit is **Dim form = New NewGameForm()**). Je kan dit op drie manieren doen:

³Als dit niet aanwezig is, kan je dit scherm openen door op **Ctrl + W** en vervolgens op **Ctrl + E** te drukken of te gaan naar **View** → **Error List**.

- Dubbelklik in de marge links van deze instructie
- Zet je cursor op de regel van deze instructie, en druk op **F9**
- Zet je cursor op de regel van deze instructie, en ga naar **Debug** → **Toggle Breakpoint**

Je kan je programma nu uitvoeren⁴ en terug hetzelfde dialoogvenster oproepen.

Het programma wordt nu onderbroken en je wordt automatisch naar de locatie van het breakpoint gebracht. Je bent dus zeker dat dit onderdeel van de broncode wordt aangedaan. Dat is ook logisch, omdat de eerste instructie ervoor zorgt dat het dialoogvenster wordt opgestart. De bug zal zich dus op een 'latere' locatie in de broncode bevinden.

Bekijk de verdere code in `NewGameForm.GetNewGameData()`. Deze functie bevat een **if-then-else**-clausule. Misschien gaat er hier wel iets mis. Plaats nu breakpoints om te controleren of beide gevallen worden aangedaan.

Wat merk je op? Worden beide clausules aangedaan? Waaraan kan dit liggen? Los het probleem op⁵.

6.2 De geplaatste breakpoints bekijken

Soms is het handig om te weten te komen hoeveel breakpoints je al in de broncode hebt geplaatst en waar ze zich bevinden. Visual Studio biedt een scherm aan dat deze informatie bevat.

Ga naar **Debug** → **Windows** → **Breakpoints** of druk **Ctrl + Alt + B** in om dit scherm weer te geven. Het scherm zal het bestand en het regelnummer van alle breakpoints die je in je project hebt geplaatst tonen. Hier zal je de breakpoints van de functie `NewGameForm.GetNewGameData()` terugvinden.

6.3 Een breakpoint buiten werking stellen

Omdat je nu het probleem hebt kunnen oplossen, is het nogal nutteloos dat het programma elke keer onderbreekt wanneer je een nieuw spel wil starten. Soms is het nuttig om bepaalde breakpoints tijdelijk uit te schakelen. Dit kan je dan later terug inschakelen als blijkt dat er op dat punt toch mogelijk iets misloopt.

Een breakpoint tijdelijk uit te schakelen doe je als volgt:

- Klik met je rechtermuisknop op een breakpoint en kies voor de optie **Disable Breakpoint**.
- Gebruik de breakpoint explorer (die je in Sectie 6.2 hebt opgestart). Je moet gewoon het vinkje voor het juiste breakpoint uitzetten.

Op een soortgelijke manier kan je later de breakpoints ook terug aanzetten.

6.4 Een breakpoint verwijderen

We hebben de in Sectie 6.1 geplaatste breakpoints eigenlijk niet meer nodig. We kunnen ze verwijderen door te dubbelklikken op de breakpoints zelf of door dit aan te geven in de breakpoint explorer.

⁴Merk op dat je de breakpoints ook nog kan toevoegen *gedurende* het uitvoeren van het programma. In dat geval zal er een onderbreking plaatsvinden de volgende keer dat de instructie wordt aangedaan.

⁵Hint: het heeft iets te maken met `Windows.Forms.DialogResult`.

6.5 Andere opties met breakpoints

Er zijn nog andere mogelijkheden die je met breakpoints hebt. We sommen ze hier kort even voor je op:

- Je kan ervoor kiezen een zekere **conditie** aan het onderbreken van een uitvoering vast te hangen. Het breakpoint wordt dan enkel gerespecteerd indien er aan de conditie wordt voldaan. Deze conditie neemt dezelfde vorm aan als een clause die bijvoorbeeld in een **if-then-else**- of **while**-construct wordt gebruikt.
- Met een **Hit Count** kan je instellen dat een breakpoint pas wordt gerespecteerd wanneer een breakpoint verschillende keren wordt aangedaan. Dit is onder meer handig wanneer je in lussen of recursieve functies aan het debuggen bent.
- Een **Filter** laat het geavanceerd gebruik van een breakpoints toe. Pas wanneer er aan bepaalde omgevingsvariabelen wordt voldaan, zal zo'n breakpoint gerespecteerd worden. Dit zal je niet nodig hebben om je practicum tot een goed einde te brengen.
- Je kan er ook voor kiezen om een bepaalde actie uit te voeren (bijvoorbeeld tonen van een bericht) wanneer een breakpoint wordt aangedaan met de **When Hit...**-clause.
- Breakpoints worden gegroepeerd met behulp van een **Label**. Zo kan je alle breakpoints die gerelateerd zijn aan een bepaalde softwarebug makkelijker samen olijsten of uitschakelen.

7 Break on exception

Voer het programma nu verder uit. Na het tonen van een dialoog voor het configureren van het spel wordt het ook daadwerkelijk opgestart. Je krijgt een bord met plaatjes voorgeschoteld, die allemaal nog omgedraaid op tafel liggen. Wanneer je op de plaatjes klikt, wordt je echter prompt terug in de broncode geworpen op een regel binnen de functie `TileControl.GetInteriorImage()`.

Een melding geeft weer dat er een onbehandelde `ArgumentOutOfRangeException` in de code gevonden werd. In de code tracht men een waarde uit een array te lezen door een index op te geven die niet bestaat.

Wanneer er een het programma een interne fout voorkomt die een Exception veroorzaakt, zal Visual Studio de uitvoering van een programma in de meest voorkomende gevallen pauzeren juist voordat ze afgehandeld. De afhandeling van een exception betekent immers vaak ook de stopzetting van het programma.

Pauzering kan hier nuttig zijn om op zoek te gaan naar de oorzaak van de exception. Je kan dan de waarden inspecteren van de variabelen om na te gaan in welke zin deze meespeelden in het veroorzaken van de fout. Dit gaan we in de volgende sectie doen om te achterhalen hoe deze bug werd veroorzaakt.


8 Locals en Watches gebruiken

Het is natuurlijk niet de bedoeling dat er hier een foutmelding wordt gegeven. We willen weten waarom er op de opgegeven index in de array `Configuration.Icons` geen element te vinden is. Deze array zou immers normaal de afbeeldingen van de plaatjes moeten bijhouden die aan de gebruiker worden getoond. Intern worden de plaatjes als een nummer bijgehouden, en de nummers worden vervolgens gebruikt om de juiste afbeelding te laden. Het gaat in deze functie echter mis om aan de hand van het nummer de afbeelding op te vragen.

8.1 De waarde van de index achterhalen

We willen dus eerst eens kijken wat de waarde van de index `_gameTile.Contents` is. Het kan immers zijn dat dit een negatief getal is (en dat daarmee de adressering in de array foutief is). We kunnen deze waarde nagaan door ze op te zoeken in het **Locals**-scherm. Open dit door op **Alt + 4** of in het menu **Debug** → **Windows** → **Locals** te kunnen. Je krijgt twee elementen te zien:

- **Me** toont de instantievariabelen van de actieve klasse.
- `GetInteriorImage` is de naam van de functie waar we momenteel inzitten.

Klik op het -je voor **Me** om het te expanderen. Je krijgt nu een hele lijst van elementen. De blauwe blokjes zijn de variabelen van de actieve klasse, de andere icoontjes zijn de eigenschappen die aan de klasse zijn toegekend.

Expandeer nu het element `_gameTile`. Omdat dit zelf een object is van de klasse `GameTile`, heeft het zelf variabelen en eigenschappen en daarom kunnen we dit element expanderen. We krijgen een achttal elementen te zien, waaronder de eigenschap `Contents`. Dit heeft geen negatief getal⁶ en dus lijkt het dat de index van de array niet noodzakelijk de oorzaak zou moeten zijn.

8.2 De waarde van de array bekijken

We willen dus kijken of er niets misgaat met de waarde van `Configuration.Icons`. Dit is echter een verwijzing naar een statische variabele. Omdat dit geen locale variabele is en we ze tussen de waarden in **Locals** dus niet zullen terugvinden, moeten we hiervoor een andere manier gebruiken. We kunnen dit doen door een **watch** in te stellen.

Een watch laat toe om de waarden van variabelen gedurende de uitvoer van een programma te inspecteren. Het nut ervan zal pas echt duidelijk worden later in deze oefenzitting, wanneer je door de broncode leert 'stappen' gedurende de uitvoering. Voor het inspecteren van waarden uit een module (zoals `Configuration`) kunnen we ze echter ook gebruiken.

We gaan nu een watch toevoegen. Ga terug naar de broncode in de functie `TileControl.GetInteriorImage()`. Zet je cursor op de `Icons`-sleutelwoord in de groengekleurde lijn. Je kan hier nu op vier manieren een watch voor toevoegen:

- Door in de broncode met de rechtermuisknop op `Icons` te klikken en te kiezen voor **Add Watch**
- Door in de broncode met de rechtermuisknop op `Configuration` te klikken en te kiezen voor **Quickwatch**. Er verschijnt nu een dialoogvenster waarin aangegeven staat dat `Configuration` niet in een expressie kan gebruikt worden. Voeg daarom aan de expressie de `Icons`-eigenschap toe. De volledige expressie is dan `Configuration.Icons`. Klik vervolgens op **Reevaluate** en op **Add Watch** en sluit het venster af.
- Door via het menu **Debug** → **Quickwatch** te gaan en de variabelenaam in te geven die je wenst in de watch te plaatsen. In dit geval is het de expressie `Configuration.Icons`. Klik vervolgens op **Reevaluate** en op **Add Watch** en sluit het venster af.
- Door op **Shift + F9** te drukken en de expressie `Configuration.Icons` in te geven in het Quickwatch-venster. Klik vervolgens op **Reevaluate** en op **Add Watch** en sluit het venster af.
- Een andere manier die op dit moment niet mogelijk is maar die je kan gebruiken wanneer je een watch op instantievariabelen wil opstellen, is het opstellen van een watch door in het **Locals**-scherm met de rechtermuisknop op de gezochte variabele te klikken en te kiezen voor **Add Watch**.

⁶Je zou er een waarde tussen nul en zestien te zien moeten krijgen.

Klik nu in het menu op **Debug** → **Windows** → **Watches** → **Watch 1**. Er verschijnt een scherm waarin de geïnspecteerde variabelen instaan. Hierin is de waarde van de eigenschap **Icons** weergegeven. Je kan zien dat het nul elementen heeft en dat er geëxpandeerd zelfs een foutboodschap bijstaat die zegt dat het een lege array is.

8.3 Op zoek naar het probleem

Er moet dus iets misgaan met de toekenning van elementen in de array. Hiervoor zullen we verder in de code moeten spitten om te ontdekken wat er precies is misgegaan. Omdat **Icons** een eigenschap is van de module **Configuration**, moet er bij de initialisatie iets fout lopen.

Ga dus naar de procedure **New** in **Configuration**. Hier zullen we naar de toekenning aan de variabele **_icons** moeten kijken om te kijken wat er misgaat. Dit gebeurt helemaal op het einde van de procedure. Zet een breakpoint op de regel **_icons.Add(XmlToImage(node))** en voer het programma opnieuw uit.

Je komt weer onmiddellijk terecht op de exception bij de instructie **Return Configuration.Icons(_gameTile.Contents)**. Dit wil dus zeggen dat de lus waarin de instructie **_icons.Add(XmlToImage(node))** wordt uitgevoerd dus niet doorlopen werd. Zet daarom een breakpoint op de lus zelf (dus één plaats erboven) en voer het programma terug uit.

Je wordt onmiddellijk op het begin van de lus gewezen met de uitvoeringscursor (aangegeven door een gele pijl). De voorgaande code werd dus wél uitgevoerd en het aantal overlopen elementen moet dus nul zijn opdat de **For**-lus niet wordt uitgevoerd. Dit wil zeggen dat de functie **config.SelectNodes("/Configuration/Icons/Icons")** niets teruggeeft.

We hebben nu de oorzaak van de fout gevonden. Er is een foute aanspreking van het bestand dat de locaties van de afbeeldingen aangeeft. Om deze aanspreking te verbeteren verander je de regel **For Each node As XmlNode In config.SelectNodes("/Configuration/Icons/Icons")** door: **For Each node As XmlNode In config.SelectNodes("/Configuration/Icons/Icon")**

Verwijder nu ook alle breakpoints.

9 Step over/into/out

Start nu het spel opnieuw op. Je merkt dat je geen foutmelding meer krijgt, maar wanneer je klikt op een tweede plaatje gebeurt er niets.

Het zou toch handig zijn moest je ze toch kunnen omdraaien. Ook deze fout zullen we dus in de broncode moeten opsporen en verbeteren. Een startpunt hiervoor kan je vinden in de functie **OnTileClicked** van de klasse **GameForm**.

We vragen ons af of er wel een moment is waarop de waarde van **ActivePlayer** in de **if**-clausule niet gelijk is aan **Nothing**. Zet daarom een breakpoint op de **if**-clausule van de procedure **GameForm.OnTileClicked()** en start een nieuw spel.

9.1 Step over

Wanneer het spel gestart wordt, klik je op een plaatje. Je wordt onmiddellijk op het breakpoint gewezen. Dezelfde procedure wordt dus opgeroepen telkens er op een plaatje wordt gedrukt. Wij willen natuurlijk weten wat er gebeurt wanneer er op een tweede plaatje wordt geklikt. Druk dus opnieuw op **F5** of de **play**-knop om het programma voort te zetten en selecteer een tweede plaatje.

Opnieuw wordt je onmiddellijk op het breakpoint gewezen. De procedure wordt dus al zeker opgeroepen wanneer er voor een tweede plaatje op het speelbord wordt geklikt. Maar wij willen natuurlijk weten waar het misgaat.

Je zou nu gewoon een breakpoint kunnen plaatsen op de volgende mogelijke locaties om te weten te

komen hoe de verwerking verder gebeurt, maar de debugger biedt handigere werktuigen om de code te inspecteren.

Om stap voor stap doorheen de code te bewegen tijdens de uitvoering ervan, kan je *Step Over* gebruiken. Klik op het icoontje (🔍), druk op **F10** of ga naar **Debug** → **Step Over** om deze navigatie te doen. Je kom nu terecht op de instructie binnen de *if*-clausule, wat wil zeggen dat de *ActivePlayer* dus niet *Nothing* is zoals eerst gedacht.

Step over zal de instructie uitvoeren op de regel van de huidige uitvoeringscursor. Daarna zal het de uitvoeringscursor naar de volgende instructie in de functie brengen. Op deze manier kan je instructie per instructie doorheen de broncode 'lopen' en zien wat er wordt uitgevoerd en (via het *Locals*-scherm) wat de waarde van de variabelen zijn. Omdat de instructies ook echt uitgevoerd worden, kan het ook zijn dat dit zichtbaar wordt in het formulier (als de aangedane instructies hier een wijziging toebrengen).

9.2 Step into

De uitvoeringscursor staat nu op de instructie `ActivePlayer.PickFromUI(control.Tile.Position)`. We kunnen manueel deze procedure opzoeken, maar het zou makkelijker zijn als we niet weer een breakpoint hiervoor hoeven te zetten. Hiervoor bestaat er het **Step Into**-commando.

We kunnen nu in de `PickFromUI()`-procedure stappen door op de knop (🔍), **F11** of **Debug** → **Step Into** te drukken⁷. Je komt nu in een andere oproep terecht, namelijk `HumanPlayer.PickFromUI()`. Omdat hier niets anders gebeurt⁸ dan een andere oproep, kan je opnieuw in de procedure stappen.

Visual Studio houdt een lijst bij van welke procedures en functies doorlopen werden om op de huidige locatie te komen in de **call stack**. Open deze door op **Alt + 7** te drukken of in het menu voor **Debug** → **Windows** → **Call Stack** te kiezen. Je ziet hier initieel enkel een gele pijl voor de procedure of functie waar de uitvoeringscursor zich bevindt. Indien je op één van de andere elementen uit de lijst dubbelklikt, word je automatisch op de instructie gewezen die de uitvoeringscursor naar de bovenste functie bracht. Het bovenste element in de lijst is altijd de laatst opgeroepen functie. Dubbelklik nu op een andere functie en je merkt dat de groene pijl je huidige locatie aangeeft. Dit maakt navigeren in de broncode makkelijker.

Stap nu binnen `Game.PickTile()` nog eens in de oproep `_state.Value.PickTile(pos)` en daarin over het *if*-construct om te kijken of deze wordt uitgevoerd. Je merkt dat de cursor erover springt naar het einde van de procedure. Er is dus wellicht iets mis met de clausule van het *if*-construct. Dit gaan we controleren in de volgende sectie.

Een kleine opmerking bij *Step Into* is dat het niet in procedures of functies zal stappen die afkomstig zijn van de softwarebibliotheek van Visual Studio. Dit merkte je misschien al toen je een *Step Into* wou uitvoeren op een `Debug.Assert`-operatie en er gewoon werd overgestapt. Omdat de broncode van deze functies niet beschikbaar is kan je er niet zomaar instappen om er fouten in te zoeken. Je moet er dus van uitgaan dat deze foutloos werken, wat ook bijna altijd het geval is⁹.

9.3 Step out

Met *Step Into* kan je makkelijker de code overlopen. Indien de oproep van een bepaalde procedure of functie mogelijk een probleem vormt, kan je hiermee ook deze code bekijken. Wil je dit echter niet, dan kan je met een *Step Over* op het zelfde uitvoeringsniveau blijven.

Wanneer je met *Step Into* binnen een functie of procedure terecht komt, en je merkt (eventueel na een deel van de code overlopen te hebben) dat deze functie geen fout zal vertonen, dan kan je er onmiddellijk uitstappen met behulp van **Step Out** (🔍).

⁷Het kan zijn dat er een melding verschijnt die de vraag of er een melding bij een automatische step-over moet worden gegeven. Klik hier op 'No'.

⁸De `Debug.Assert`-instructie wordt hier automatisch overgeslaan. Hierover later meer.

⁹In deze oefenzitting zou je alvast geen last moeten hebben van bugs binnen de softwarebibliotheek, maar ook voor je practicum is dit zeer onwaarschijnlijk!

Met *Step Out* zal je terug worden gebracht naar de vorige functie binnen de call stack, op de positie juist na de oproep van de functie of procedure waarin je je nu bevindt.

10 De plaatjes omdraaien

In Sectie 9.2 waren we terecht gekomen in de procedure `WaitingForSecondPickState.PickTile()`. Er leek iets mis te gaan met het `if`-construct. We gaan dus controleren of er aan de clause kan voldaan worden.

Voor je dit doet, is een klein beetje uitleg over de broncode misschien wel handig. Intern worden de plaatjes bewaard door instanties van de klasse `GameTile`. Elke `GameTile` heeft een `Shown` eigenschap die weergeeft of het plaatje momenteel wordt getoond, een `Paired` eigenschap die weergeeft of dit plaatje en zijn tweelingsbroer al werden gevonden (en dus ook worden getoond) en `Contents` kent een getal aan het plaatje toe zodat de paren intern geïdentificeerd herkend worden. Alle plaatjes worden op een bord geplaatst dat wordt bijgehouden door de `GameGrid`-klasse en waarbij de `Position`-eigenschap van de plaatjes de positie aangeeft.

We willen weten wat de zogenaamde `Value`-waarden zijn van de `Shown`-eigenschap die er hier worden gevraagd. De `if`-clause gaat na of de waarde van deze eigenschap `True` is¹⁰. Plaats daarom nu een `watch` op `_parent` binnen deze procedure. Je merkt misschien dat dit niet zomaar mogelijk is op de eigenschappen die ervan worden opgeroepen. Weet je ook waarom?

Inspecteer nu de waarden van enkele plaatjes. Doe dit door de `_grid` instantievariabele drie keer te expanderen binnen `_parent`¹¹. Je komt op de `GameTile`-instanties terecht, die worden weergegeven als `(0,0)`, `(0,1)`, `(0,2)`, enzovoort. Dit zijn de objecten van de plaatjes die we willen bekijken. Expandeer er een paar en kijk naar welke `Value` de `Shown`-eigenschap heeft. Je merkt op dat deze allemaal op `False` staan, op het plaatje dat je al hebt aangeklikt na.

De code initialiseert dus de plaatjes als met de `Shown`-eigenschap op `False`. Het `if`-construct in de procedure `WaitingForSecondPickState.PickTile()` vereist echter dat de waarde op `True` staat voordat hij ze omdraait. Er is dus een fout geslopen in de logica van het `if`-construct. Zorg ervoor dat deze fout gecorrigeerd wordt.

Wanneer je nu het programma opnieuw uitvoert, zal je het kunnen spelen. Er is echter nog een probleem wanneer je het uitspeelt. Dit zal je zelf moeten achterhalen en oplossen. In Sectie 12 staan hiervoor enkele tips.

Extra opdracht: Je hebt in deze sectie kennism gemaakt met de eigenschappen van de klasse `GameTile`. Kan je deze nu ook gebruiken om vals te spelen met het spel? Probeer dit uit! Maak hiervoor gebruik van een `watch`.

11 Andere handige debugging-werktuigen

Visual Studio biedt ook nog andere werktuigen aan die helpen bij het debuggen.

11.1 Debug.WriteLine

Wanneer je op een bepaald ogenblik in de broncode een melding wil doorgeven kan je gebruik maken van de instructie `Debug.WriteLine`. Dit zorgt dat er een bericht wordt weergegeven dat enkel zichtbaar is in debug modus.

¹⁰Indien jullie dit nog niet in een ander programmeervak hebben gezien, de `If _parent.Grid(secondPick).Shown.Value = True Then` en `If _parent.Grid(secondPick).Shown.Value Then` zijn equivalent.

¹¹Je vindt het misschien raar dat je drie keren dezelfde variabele moet expanderen. Geen zorgen: dit komt door het ontwerp van de code. Waarom dit zo gedaan werd, zal je later in je studentenloopbaan ontdekken.

Plaats nu de instructie `Debug.WriteLine("Debugging bericht")` ergens in de broncode waar het zeker zal uitgevoerd worden (bijvoorbeeld in de `New` procedure van `GameForm`). Start daarna opnieuw het programma.

In Visual Studio zal er onmiddellijk wanneer de `WriteLine`-instructie wordt uitgevoerd de focus op het onderste scherm veranderen naar een console, het zogenaamde **Output Window**. Hierin worden alle berichten die met debugging te maken hebben samengebracht.

Een variant op `Debug.WriteLine` is de instructie `Debug.WriteLineIf`, die een bericht enkel weergeeft wanneer er aan een opgegeven clause (dit is het eerste argument dat je eraan meegeeft) wordt voldaan.

11.2 Beweringen

De instructie `Debug.Assert` kan je het best vergelijken met een (blokkerende) `if`-clause in functie van het debuggen. We controleren hierin de waarde van een variabele. Voldoet de variabele niet aan de clause die er in de bewering staat, dan zal Visual Studio een melding geven die aangeeft dat aan de bewering niet werd voldaan (**Assertion Failed**).

Hierbij wordt ook een *call stack* gegeven die aangeeft welke functies en procedures er werden opgeroepen om tot deze bewering te komen. Je kan kiezen om het programma toch uit te voeren, opnieuw te starten of te stoppen.

Assertions worden vaak gebruikt om precondities of postcondities van functies en procedures te controleren. Voor het debuggen kunnen ze handig zijn om aan te geven dat iets wat verwacht werd aan een voorwaarde te voldoen (zoals de waarde van een variabele), hier wel of niet aan voldoet.

11.3 Option Strict

De compiler van Visual Studio kan geconfigureerd worden om strenger te zijn op impliciete conversies van de waarden van variabelen naar een ander type. Neem het volgende kleine voorbeeldje:

```
Dim kommaGetal As Double = 3.1415926536
Dim gewoonGetal As Integer
gewoonGetal = 3 * kommaGetal
MessageBox.Show(gewoonGetal.ToString)
```

Zie je hier een probleem mee? Waar gaat het verkeerd?

Visual Studio kan dit type problemen voorkomen door middel van de optie **Strict**. Standaard staat deze controle uit, maar het kan belangrijke¹² softwarefouten voorkomen en het is dus aangeraden deze controle in te schakelen.

Om te voorkomen dat er informatie verloren gaat bij het plaatsen van een kommagetal in de variabele van een integer, pas je de projectopties best aan. Dit doe je door naar **Tools** → **Options** te gaan en hier onder **Projects and Solutions** → **VB Defaults** de optie **Option Strict** op **On** te zetten. Dit zal het risico op een foutieve afronding omdat de waarde aan de variabele van een 'nauwer' type wordt toegekend tegengaan.

Standaard verwacht Visual Studio wel dat een variabele wordt gedeclareerd voor hij kan worden gebruikt. Dit is de zogenaamde **Option Explicit** die je misschien in de instellingen zag staan. Zou dit uitstaan, dan was er het risico dat je met typefouten een 'nieuwe' variabele zou aanspreken en hierdoor een softwarefout introduceren omdat een eerste verwijzing van een variabele dan een impliciete declaratie zou teweegbrengen en met deze variabele ook ineens de instructies worden uitgevoerd. Laat deze dus zeker aan staan!

¹²Om een idee te krijgen hoe fataal de gevolgen kunnen zijn van een slechte omzetting, kan je het artikel lezen op <http://www.ima.umn.edu/~arnold/disasters/ariane.html>

12 Opdracht

Je hebt in deze oefenzitting kennisgemaakt met enkele hulpmiddelen om fouten uit een programma te halen. Het spel werkt echter nog niet volledig. Er wordt normaal een melding gegeven wanneer alle plaatjes zijn omgedraaid. Ga op zoek waar het verkeerd gaat, en probeer het spel helemaal werkende te krijgen. Begin hierbij bij de procedure `WaitingForFirstPickState.PickTile()`.

Tip 1: Om dit probleem te vinden is een verdere uitleg van de broncode nodig. De `Game`-klasse bevat de logica voor het spel in goede banen te loodsen. Een `Game` heeft een zogenaamde `state`. Je hebt vijf statussen: de `InitializationState`, de `WaitingForFirstPickState`, de `WaitingForSecondPickState`, de `LingeringState` en de `EndedState`. Een status bepaalt welke acties er in de huidige fase van het spel mogen worden uitgevoerd en zal zorgen dat de nodige instructies worden uitgevoerd om het spel verder te laten lopen. Zo zal er in de `WaitingForFirstPickState` worden toegelaten dat de gebruiker op een niet-weergegeven plaatje klikt en dit omdraaien. Wanneer de gewenste acties in een bepaalde status zijn uitgevoerd, dan zal het spel overgaan naar een volgende status.

Tip 2: Je dient de code maar op één locatie aan te passen.

Succes!

Yolande Berbers
Raoul Starckx
Thomas Winant
Job Noorman
Jasper Bogaerts