Deel 18

Interfaces

Dit deel introduceert een nieuwe taal element van C# : interfaces.

Er wordt veel tekst en uitleg besteed aan voorbeelden die duidelijk maken waaróm we interfaces nodig hebben in een programmeertaal als C#.

Elk voorbeeld is een programma dat uit meerdere klassen bestaat, het is dus telkens een gans project.

Bij onze uitleg worden UML klassendiagrammen gebruikt om de structuur van zo'n project visueel voor te stellen. Zo'n diagram toont welke klassen er zijn en wat elke klasse bevat qua datavelden, properties en methods.

De uitleg gaat o.a. over het software ontwerp van deze projecten (zie software ontwikkelingsproces). Je zult merken dat we veel zinvols over de goeie en slechte aspecten van een project kunnen vertellen, zonder dat we de eigenlijk code (i.e. de implementatie) nodig hebben.

Dat is precies waarom 'ontwerp' zo belangrijk is in het ontwikkelingsproces : je kunt alternatieven afwegen (en beslissingen nemen) zonder telkens de code uit te moeten schrijven of wijzigen!

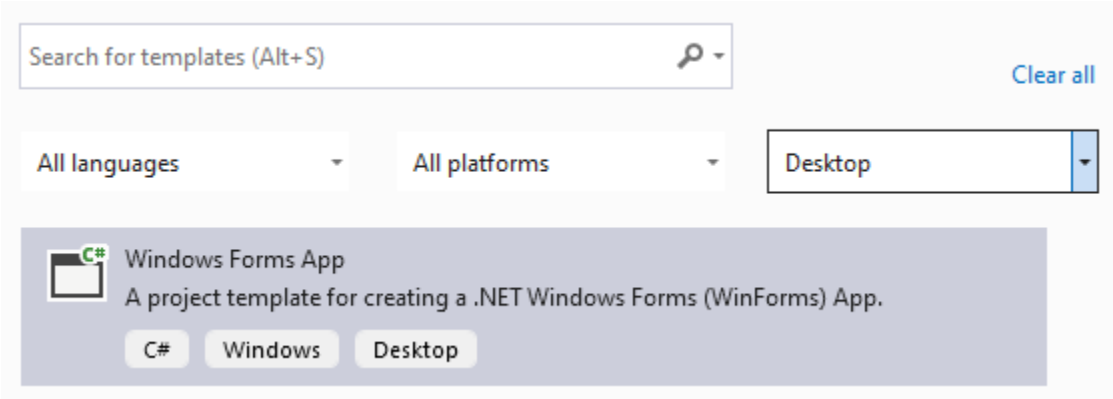
Dus beschouw deze "lange uitleg om tot interfaces te komen", als een kleine introductie over wat software ontwerp is. Je zult merken dat het zich op een "hoger niveau" afspeelt dan de eigenlijke code regels die we tot nu toe geschreven hebben.

Op een bepaald moment moet er natuurlijk ook concrete C# code geschreven worden op basis van het ontwerp! Je kunt bij het ontwerpen zoveel luchtkastelen bouwen als je wil, maar enkel de implementatie (i.e. de broncode) levert iets tastbaars op voor de opdrachtgever : namelijk een uitvoerbaar programma.

Bij dit deel horen verschillende projecten. De broncode van elk project staat in een .zip file die je (net als bij delen GFX1 en GFX2) telkens in een leeg(!) "Windows Forms App" project moet plaatsen.

Voor de duidelijkheid, nog eens de instructies hiervoor :

Maak een nieuw project aan van het type



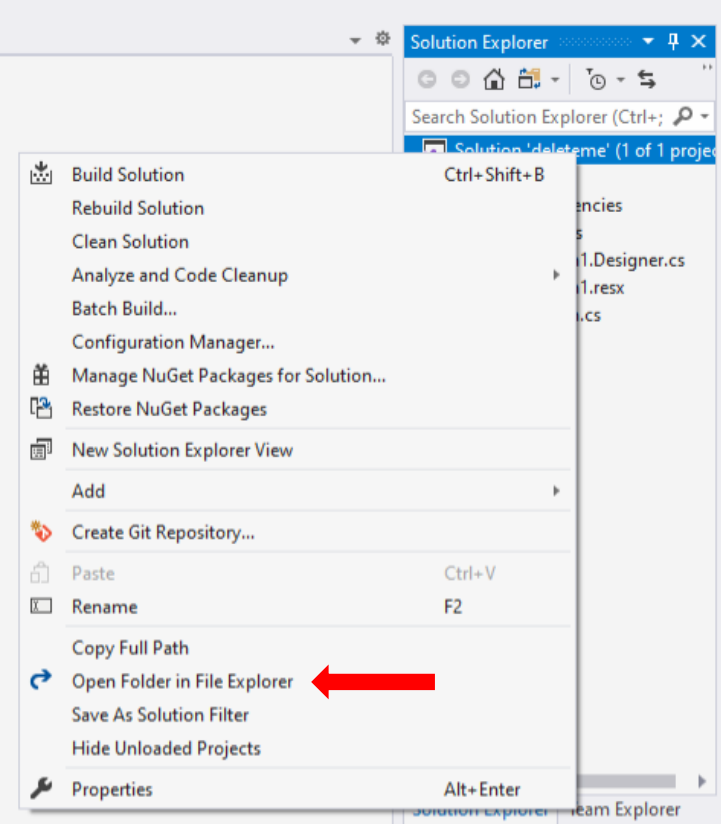
In de Solution Explorer verwijder je nu de files voor Form1.cs en Program.cs, we zullen deze niet nodig hebben.

Selecteer ze en rechtsklik erop, in de popup vind je de optie om ze te verwijderen.

Unzip het .zip bestand met de broncode en stop de .cs files in het project.

Je kunt dit doen door de files via de Windows Verkenner naar de project folder te kopiëren.

Mocht je niet weten waar de folders voor de solution en het project staan : rechtsklik op de solution in de Solution Explorer en kies in de popup voor 'Open Folder in File Explorer'.



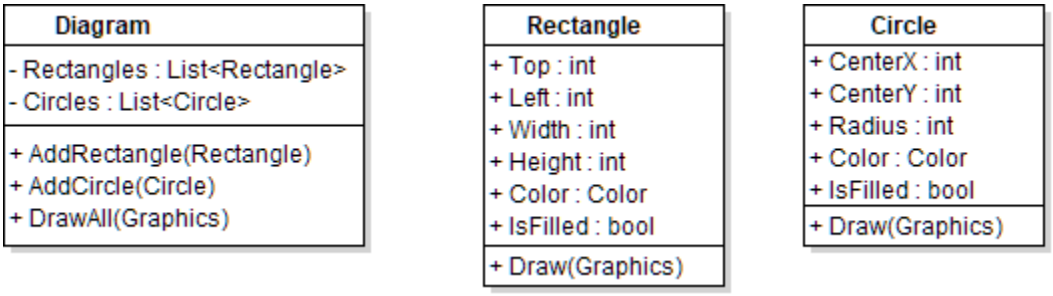
De Program klasse bevat steeds de Main method, stel deze in als het Startup Object van het project.

Download het bestand '**deel-18-demo1-apart**.zip', dit is hetzelfde programma als 'oplossing GFX2.04'. Stop de broncode in een leeg(!) 'Windows Form App (.NET Core)' project en bekijk nog eens de klassen.

Intermezzo : klassendiagrammen

Als we willen redeneren of communiceren over de grotere structuren in een programma, is het soms handig om dit visueel te doen. Een mogelijke voorstellingswijze is het UML klassendiagram.

Het klassendiagram van het huidige project '**deel-18-demo1-apart**' ziet er als volgt uit :



Je ziet dat er per klasse een blokje voorzien wordt met drie onderverdelingen :

naam van de klasse
de datavelden/properties van de klasse
de methods van de klasse

Merk op dat, in tegenstelling tot C# code, de types van datavelden en teruggeeftypes van methods achteraan staan.

Bijvoorbeeld in C# schrijven we

```
private int _height;
public bool IsFilled { get; set; }
public bool Contains(int x, int y) { … }
```

maar op een klassendiagram wordt dit

```
- _height : int
+ IsFilled : bool
+ Contains(int, int) : bool
```

De public of private visibility wordt resp. met een + en een - symbool aangeduid.

De constructoren worden meestal niet getoond : ze zijn zelden interessant genoeg en bovendien hebben ze dikwijls veel parameters waardoor ze teveel plaats zouden innemen op het diagram.

Stel je nu eens voor dat we een spelletje of een grafische editor willen bouwen waarbij meer soorten figuren gebruikt worden dan alleen maar cirkels en rechthoeken. Bijvoorbeeld driehoeken, ruiten, afbeeldingen, enz.

We zullen dan een aantal nieuwe klassen moeten toevoegen voor die figuren :

```
class Triangle
class Diamond
class Image
```

Elke klasse zal een Draw method krijgen, zodat we haar objecten kunnen opdragen zichzelf te tekenen :

```
public void Draw(Graphics g) { … }
```

Bovendien zal in een nuttiger programma, elke soort figuur meer moeten kunnen dan alleen maar zichzelf te tekenen via de Draw method.

Bv. in een spelletje zullen figuren moeten **bewegen** naarmate de tijd verstrijkt. We zouden voor elke figuur een Simulate method kunnen voorzien waaraan we het aantal verstreken milliseconden meegeven (zodat de figuur weet in welke mate ze haar positie moet aanpassen) :

```
public void Simulate(long milliseconds) { … }
```

Bv. in een grafische editor moeten we figuren kunnen **selecteren** door erop te klikken. Om na te gaan of een bepaald punt waarop geklikt werd binnen de figuur valt, zouden we een soort Contains method kunnen voorzien (met x en y parameters voor de coördinaten van het punt) :

```
public bool Contains(int x, int y) { … }
```

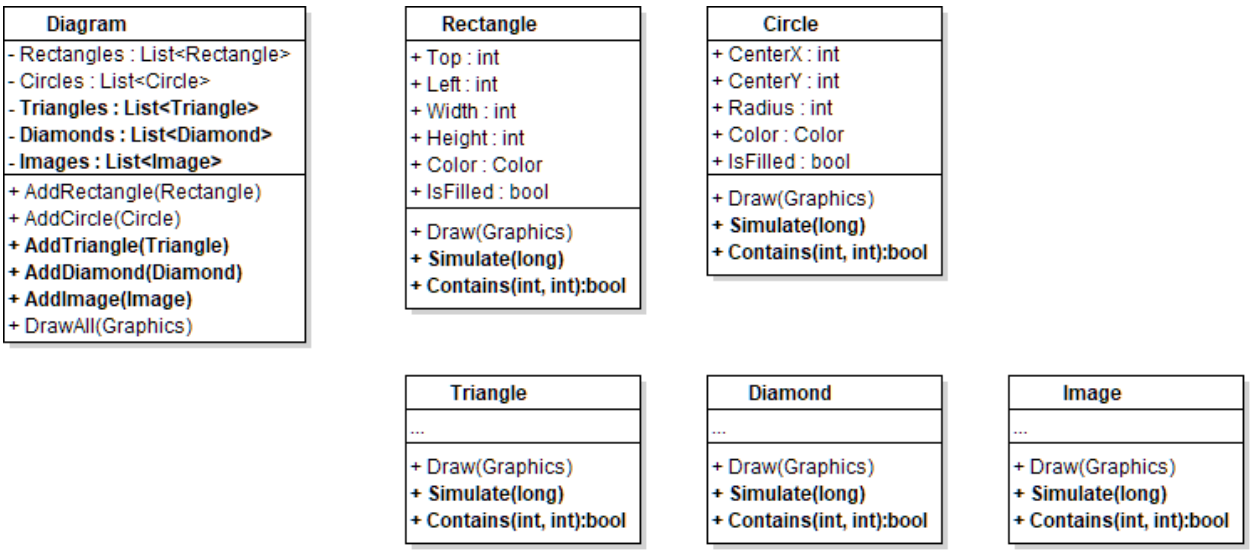
We bekomen dus een programma waarin de verschillende "figuurachtige" klassen (bv. Circle, Rectangle, Triangle, Diamond, Image) deels dezelfde methods bevatten.

De implementatie van zo'n method (bv Draw, Simulate of Contains) zal in elke klasse weliswaar anders zijn, maar elke klasse heeft zo'n method met precies dezelfde naam en precies dezelfde parameters.

Merk op dat er voor de compiler geen verband bestaat tussen al die klassen en hun gelijkaardige methods, die ziet gewoon klassen met elk hun eigen methods en merkt geen gelijkenissen op.

Veronderstel een fictief programma met **bewegende selecteerbare** figuren. Probeer je eens in te beelden hoe zo'n programma er dan ongeveer zou uitzien, qua klassen en methods.

Voor het bewegen en selecteren van de figuren kunnen we de methods Simulate en Contains voorzien. Het klassendiagram met de relevante klassen ziet er dan zo uit :



(alles wat erbij gekomen is staat **dikgedrukt**)

Deze oplossing is eigenlijk niet zo goed.

Ze heeft wel als **pluspunt** dat alles wat met een bepaalde soort figuur te maken heeft, netjes gebundeld is in één klasse. Elke klasse heeft een mooi afgelijnde focus en de onderdelen van een klasse vertonen een grote samenhang. Dit heet trouwens ***cohesie*** en is een belangrijke eigenschap van goed ontworpen klassen (hoge cohesie is beter).

Een zeer groot **minpunt** is, dat de klasse Diagram afhankelijk is van heel veel andere klassen. Telkens we een nieuw soort figuur willen ondersteunen, moet de klasse Diagram aangepast worden.

De ***mate van afhankelijkheid van andere klassen***, is een ander zeer belangrijk kwaliteitskenmerk van goed ontworpen klassen. Voor alle duidelijkheid : *minder afhankelijk* is beter 😉

Als we eens zouden nadenken over de verschillende Add mogelijkheden in klasse Diagram, zouden we vaststellen dat deze methods er allen hetzelfde uitzien :

```
private List<Circle> Circles {get; set; }          // Circle als voorbeeld maar we hebben dit ook
public void AddCircle(Circle c) {                  // Rectangle, Triangle, Diamond en Image.
    this.Circles.Add( c );
}
```

Als we een manier hadden om alle soorten figuren op eenzelfde manier te behandelen, dan zouden we al deze gelijklopende Add varianten kunnen vervangen door één enkele Add method. *Helaas, zucht..*

De DrawAll method in klasse Diagram van dit fictief programma zal er ongeveer zo uitzien :

```
public void DrawAll(Graphics gfx) {
    // Merk op dat de volgorde van de loops bepaalt dat je bv. nooit een rechthoek bovenop een cirkel zult zien!
    foreach (Rectangle r in this.Rectangles) {
        r.Draw(gfx);
    }

    foreach (Circle c in this.Circles) {
        c.Draw(gfx);
    }

    foreach (Triangle t in this.Triangles) {
        t.Draw(gfx);
    }
    enz…
}
```

Weerom, mochten we een manier hebben *om alle soorten figuren op eenzelfde manier te behandelen*, dan zouden we ze samen in 1 lijst stoppen en zou er maar één simpele foreach-loop nodig zijn.

De reden waarom we in klasse Diagram voor elk soort figuur een aparte Add method en een aparte List nodig hebben is : elke soort wordt voorgesteld door een apart type en er is niks gemeenschappelijks.

Intermezzo : static typing

In het verleden is gebleken dat veel fouten in een programma veroorzaakt worden doordat het programma soms op een ongepaste manier met data omgaat. Deze uitvoeringsfouten worden veroorzaakt door programmeerfouten : een programmeur gebruikte in de code een bepaalde waarde of object op een niet-toegelaten manier.

Dit gebeurt doorgaans per vergissing, de programmeur veronderstelt dat er op een bepaalde plaats in het geheugen (bv. de opslagplaats van een variabele) een waarde van soort X zal staan maar tijdens de uitvoering staat daar eigenlijk een waarde van soort Y.
Vermits zowel X als Y waarden in het geheugen niet meer zijn dan een klompje bits, zal het programma de bits die een Y-waarde voorstellen, verkeerdelijk als een X-waarde interpreteren. Het programma probeert dus een X-bewerking op een Y-waarde toe te passen wat ofwel tot een crash leidt ofwel een onzinnige waarde produceert die later in de uitvoering problemen geeft.

Bijvoorbeeld, we gebruiken een int waarde om een leeftijd bij te houden en stoppen die in een variabele. Tijdens de uitvoering staan op die geheugenlocatie echter plots de bits van (een stuk van) een String. Dan krijg je het soort fouten waarbij iemand volgens het programma -287483 jaar oud is.

Merk op dat in zo'n geval, een programma crash eigenlijk beter is dan verder doen met een verkeerde waarde : bij een crash merken we tenminste dat er iets verkeerd is.

In deze context spreken we trouwens niet over het *soort* waarde, maar over het *type* van de waarde.

Om dit soort type fouten te vermijden voorziet men in vrijwel alle programmeertalen een controle mechanisme : ***type checking***. Dit betekent dat men overal waar met waarden of objecten wordt gewerkt, nagaat of dit op een toegelaten manier gebeurt.

Naargelang de programmeertaal gebeurt deze controle

- tijdens de uitvoering, dit heet ***dynamic type checking*** (bv. in Javascript)
- vóór de uitvoering, dit heet ***static type checking*** (bv. in C#)

Bij static type checking is dit vaak tijdens de compilatiefase, het is dus een extra controle tijdens de omzetting van de source code naar uitvoerbare binaire code.

Static type checking is op een belangrijke manier beter : je hebt de garantie dat type fouten niet zullen voorkomen tijdens de uitvoering. Bij dynamic type checking heb je geen garanties : het kan bij de ene uitvoering goed gaan en bij de volgende uitvoering (met andere data) verkeerd lopen.

C# is een statisch getypeerde programmeertaal (d.w.z. we moeten o.a. bij variabelen steeds hun type vermelden) en gebruikt static type checking. Dit controle mechanisme verifieert dus, dat alle waarden en objecten in ons programma op een correcte manier gebruikt worden.

Bv. dat we niet proberen om een int door een boolean te delen, of een Student object bij een Button object op te tellen, of een onbestaande GetLeeftijd method oproepen op een Adres object. Er wordt o.a. ook nagegaan of we het juiste aantal parameters meegeven bij een method oproep en of die parameters wel degelijk van de juiste soort zijn.

Een belangrijk begrip dat we eerder al tegenkwamen is ‘een expressie’.

Een expressie is een zinnetje in een programmeertaal dat een waarde aanduidt. Enkel voorbeelden :

- de naam van een variabele, dataveld, property of parameter zoals
  persoon, this._geboorteDatum, this.Circles, gfx

- een method oproep (indien de method een waarde teruggeeft) zoals
  persoon.GetAdres(), c.GetRadius(), persoon.GetNaam().ToLower()

Een zinnetje als persoon.SetNaam(“Jan”) is geen expressie omdat de SetNaam method geen waarde teruggeeft (het teruggeeftype is void).

Welnu, bij static type checking moet het mogelijk zijn om te achterhalen wat het type is van elke expressie in de source code!

Voor eenvoudige expressies zoals de naam van een variabele is dit makkelijk, kijk gewoon naar het gedeclareerde type van die variabele. Bv. een declaratie als ‘int i’ leert ons dat het type van i int is.

Voor complexere expressies moeten we gaan kijken naar teruggeeftypes van methods. Bv. het type van de expressie **persoon.GetNaam().Length** is int! Wij (en de compiler) kunnen dit als volgt afleiden :

1. het type van **persoon** is klasse **Persoon**
2. in klasse **Persoon** heeft GetNaam() een **String** teruggeeftype
3. in klasse **String** zien we tenslotte dat de Length property van type **int** is.

Static type checking is dus **de reden** waarom we in klasse Diagram zoveel varianten nodig hebben :

- aan een List<Circle> object kunnen we geen Rectangle objecten toevoegen
- aan een AddCircle method oproep kunnen we geen Rectangle parameter meegeven

De compiler zal dit telkens verbieden.

Vandaar dat we allemaal aparte Add methods en aparte List properties moesten voorzien!

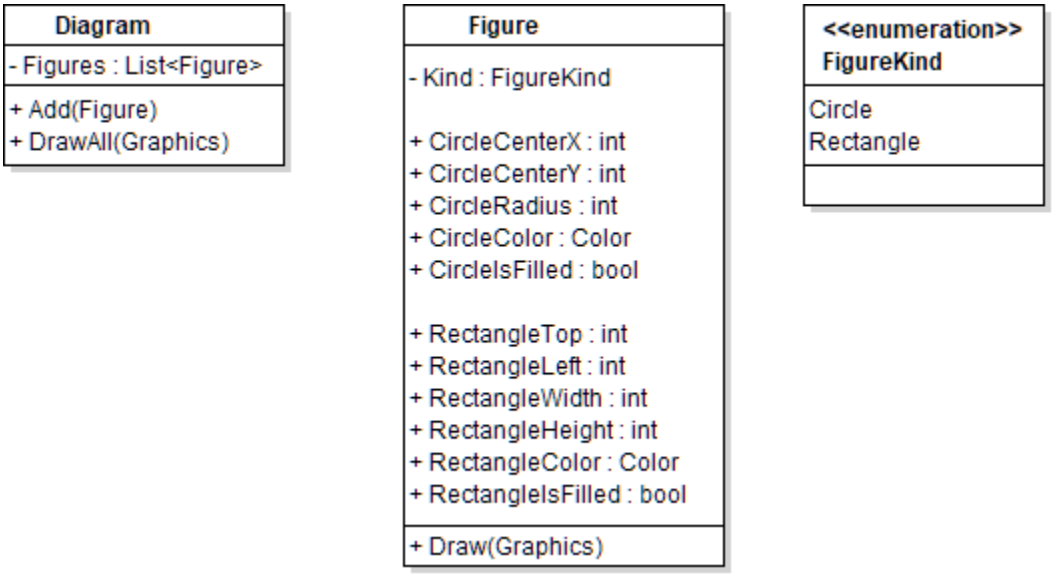
Nogmaals, het project 'deel-18-demo1-apart' heeft dus

- als pluspunt : elke klasse vertoont een hoge cohesie
- als minpunt : de klasse Diagram heeft teveel afhankelijkheden naar andere klassen

Download nu het bestand '**deel-18-demo2-samen**.zip' en stop de broncode in een leeg(!) 'Windows Form App (.NET Core)' project.

In project '**deel-18-demo2-samen**' werd de volledige functionaliteit van rechthoeken en cirkels in één enkele klasse Figure geplaatst.

Het klassendiagram met de relevante klassen ziet er zo uit :



Terzijde : "kind" in het Engels betekent "soort", het heeft dus niks met kinderen te maken 😉

Merk op dat elk Figure object een property '**Kind**' heeft die bepaalt of die Figure een cirkel dan wel een rechthoek voorstelt. Indien bv. Kind == FigureKind.Rectangle dan stelt het een rechthoek voor.

Als je in klasse Figure de code van de Draw method bekijkt, zie je dat deze uiteenvalt in twee delen :

```
public void Draw(Graphics gfx) {
    if ( this.kind == FigureKind.Circle ) {
        // Teken een cirkel
        ...
    } else if ( this.kind == FigureKind.Rectangle ) {
        // Teken een rechthoek
        ...
    }
}
```

Deze klasse Figure heeft een **zeer lage cohesie**. Ze mengt twee totaal verschillende concepten en naargelang de waarde van 'Kind' is de helft van de overige properties niet eens van toepassing!

Als we een Figure object hebben dat een rechthoek voorstelt, is er trouwens niks dat ons belet om toch cirkel-specifieke properties in te stellen. Dat kan tot fouten leiden en is dus nog een bijkomend nadeel.

Stel je eens voor dat we op deze manier ons fictief programma met **bewegende selecteerbare** figuren zouden schrijven! De klasse Figure zou bijkomende properties voor driehoek, ruit en afbeelding krijgen plus de Simulate en Contains methods. Dan wordt die Figuur klasse wel heel erg ingewikkeld!

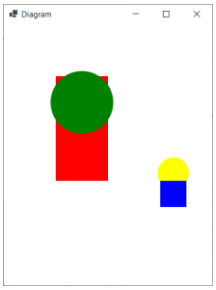
Dat is het (zéér) grote nadeel van dit ontwerp : de extreem lage cohesie van klasse Figure, de klasse lijkt wel schizofreen.

Langs de andere kant, er is ook een pluspunt : de klasse Diagram heeft dit keer weinig afhankelijkheden naar andere klassen! Als we een nieuw soort figuur willen ondersteunen, moeten we enkel de Figure klasse aanpassen, aan Diagram hoeft er niks te wijzigen.

De klasse Diagram is ook veel eenvoudiger geworden : er is nog maar één lijst van figuren nodig en er is ook maar één bijbehorende Add method. Ook de code van method DrawAll is veel eenvoudiger :

```
public void DrawAll(Graphics gfx) {
    foreach ( Figure f in this.Figures ) {
        f.Draw(gfx);
    }
}
```

Merk op dat we nu ook de mogelijkheid hebben om rechthoeken en cirkels elkaar willekeurig te laten overlappen, i.p.v. steeds alle cirkels bovenop de rechthoeken (of omgekeerd) te tekenen :



Het project '**deel-18-demo2-samen**' heeft dus

- als pluspunt : klasse Diagram heeft weinig afhankelijkheden naar andere klassen
    - (en figuren kunnen elkaar willekeurig overlappen)
- als minpunt : de klasse Figure heeft een extreem lage cohesie
    - (en een grotere kans op fouten door verkeerd gebruik v.d. properties van Figure)

Ter herhaling, project '**deel-18-demo1-apart**' had

- als pluspunt : elke klasse vertoont een hoge cohesie
- als minpunt : de klasse Diagram heeft teveel afhankelijkheden naar andere klassen

Belangrijk :

We zouden graag de voordelen willen van beide demo's zonder de nadelen. We willen :

1. voor elke soort figuur een aparte klasse
2. in klasse Diagram slechts één lijst en één Add method voor alle soorten figuren tesamen

Uit puntje 1 blijkt dat we verschillende klassen willen voor de figuren, maar uit puntje 2 volgt dat we voor klasse Diagram maar één enkel type willen.

We willen een gemeenschappelijk type voor alle soorten figuren en tegelijkertijd voor elke soort figuur een aparte klasse gebruiken.

Met hetgeen we tot nu toe gezien hebben in C# is dit niet mogelijk :

Elke klasse introduceert een eigen type. Als we meerdere klassen voor de verschillende soorten figuren voorzien, dan krijgen we ook meerdere types.

Omgekeerd, elk type hoort bij exact één klasse (als we types als int en bool eventjes negeren). Als we één gemeenschappelijk type willen voor Diagram, dan mogen we ook maar één klasse voorzien.

We hebben dus iets nieuws nodig in onze programmeertaal : **interfaces**!

Interfaces

Download nu het bestand '**deel-18-demo3-interface**.zip' en stop de broncode in een leeg(!) 'Windows Form App (.NET Core)' project.

Kijk eerst eens naar het bestand Domain/Figure.cs, daarin wordt een interface Figure gedefinieerd:

```
public interface Figure {
    public void Draw(Graphics gfx);
}
```

Op het eerste zicht lijkt dit op een klassendefinitie, maar als je goed kijkt zie je dat de Draw method helemaal geen code heeft! Iets technischer uitgedrukt : de Draw method '*is niet geïmplementeerd*' of '*heeft geen implementatie*'. Trouwens, een interface mag gerust meerdere van die methods bevatten.

Een interface is een programmeertaal element dat uitsluitend bedoeld is om een type te introduceren in het static type checking mechanisme.

Het is echter geen klasse, dus je kunt van dit nieuwe type niet zomaar objecten aanmaken. Wat voor nut heeft dit nieuwe type dan? Wel, we kunnen ervoor zorgen dat objecten van bepaalde klassen als objecten van dit nieuwe type beschouwd worden! Dat wil dus zeggen,

- in het algemeen : sommige objecten kunnen "van meerdere types zijn"
    - het type van hun klasse, plus één of meerdere (interface) types
- specifiek voor deze interface Figure : sommige objecten kunnen "***Figure-compatibel*** " zijn
    - bovenop hun eigen klasse zijn ze ook van het Figure type

De implicaties hiervan zijn wellicht niet meteen duidelijk. Bedenk echter dat als we

- Circle objecten "Figure-compatibel" kunnen maken, dan zijn ze van het Figure type
- Rectangle objecten "Figure-compatibel" kunnen maken, dan zijn van het Figure type

Als Circle en Rectangle object beiden van het Figure type zijn, dan is ons probleem opgelost : we hebben nu een gemeenschappelijk type voor al onze figuren!

Hoe leggen we vast welke objecten "Figure-compatibel" zijn?

Simpel, we schrijven dit in hun klasse.

Bv. bekijk de broncode van klassen Circle en Rectangle in dit project :

```
public class Circle : Figure {                  // Circle is "Figure-compatibel"
    ...
    public void Draw ( Graphics gfx ) {         // Circle implementeert de Draw method
        // code die een cirkel tekent
    }
    …
}

public class Rectangle : Figure {               // Rectangle is "Figure-compatibel"
    ...
    public void Draw ( Graphics gfx ) {         // Rectangle implementeert de Draw method
        // code die een cirkel tekent
    }
    …
}
```

Merk op dat beide klassen declareren dat ze "Figure-compatibel" zijn én een implementatie voor de Draw method uit de Figure interface bevatten!

Wat betekent dit "Figure-compatibel zijn"?

Dit betekent dat telkens de compiler een expressie van type Figure verwacht, een expressie van type Rectangle of Circle ook acceptabel is.

Bv. bekijk de method Add in klasse Diagram, deze heeft een parameter van type Figure :

```
public void Add ( Figure f ) { … }
```

en kijk nu naar dit stukje code in de Main method van klasse Program :

```
Diagram diagram = new Diagram();
Rectangle r1 = new Rectangle(100, 100, 100, 200, System.Drawing.Color.Red, true);
diagram.Add( r1 );
Circle c1 = new Circle(150, 150, 60, System.Drawing.Color.Green, true);
diagram.Add( c1 );
```

Normaliter worden voor die Add method enkel parameters van type Figure aanvaard, maar expressies als **r1** (van type Rectangle) en **c1** (van type Circle) zijn nu ook ok!

Bv. kijk nu eens naar dit stukje code, weerom in de Main method van klasse Program :

```
Figure c2 = new Circle (325, 285, 30, System.Drawing.Color.Yellow, true);
Figure r2 = new Rectangle (300, 300, 50, 50, System.Drawing.Color.Blue, true);
```

Normaliter zouden we aan Figure variabelen c2 en r2 enkel Figure verwijzingen mogen toekennen, maar Circle en Rectangle verwijzingen zijn nu ook ok!

Op zich is deze formulering niet erg nuttig, je kan bv. net zo goed **Circle** c2 = **new Circle**(...) schrijven, maar het illustreert wel dat "Figure-compatibel" ook voor toekenningen geldt.

Terzijde : bij wijze van experiment, verwijder eens het stukje " : Figure" bovenaan klasse Rectangle, waardoor die niet meer Figure-compatibel is. De compiler ziet meteen twee errors :

```
Diagram diagram = new Diagram();
Rectangle r1 = new Rectangle(100, 100, 100, 200, System.Drawing.Color.Red, true);
diagram.Add(r1);
Circle c1 = new Circle(150, 150, 60, System.Drawing.Color.Green, true);
diagram.Add(c1);
Figure c2 = new Circle(325, 285, 30, System.Drawing.Color.Yellow, true);
Figure r2 = new Rectangle(300, 300, 50, 50, System.Drawing.Color.Blue, true);
```

In beide gevallen gaat de foutmelding over iets als "cannot convert Rectangle to Figure".

Allemaal goed en wel, maar wat schieten we daar nu eigenlijk mee op?

Het laat ons toe om klasse Diagram volledig te schrijven op basis van het (interface) type Figure.

We schrijven daar code voor "Figure-compatibele" objecten, zonder ons te moeten vastpinnen op specifieke klassen!

Bv. een List property op basis van Figure

```
private List<Figure> Figures { get; set; }
```

Bv. de code in de DrawAll method

```
foreach (Figure f in Figures) {
   f.Draw(gfx);
}
```

Merk op dat nergens in de klasse Diagram, ook maar iets staat over klasse Rectangle of Circle. We gebruiken enkel wat beschikbaar is in de Figure interface : de Draw method.

Klasse Diagram is in dit project dus niet meer afhankelijk van klassen Rectangle of Circle, enkel nog van die ene interface Figure!

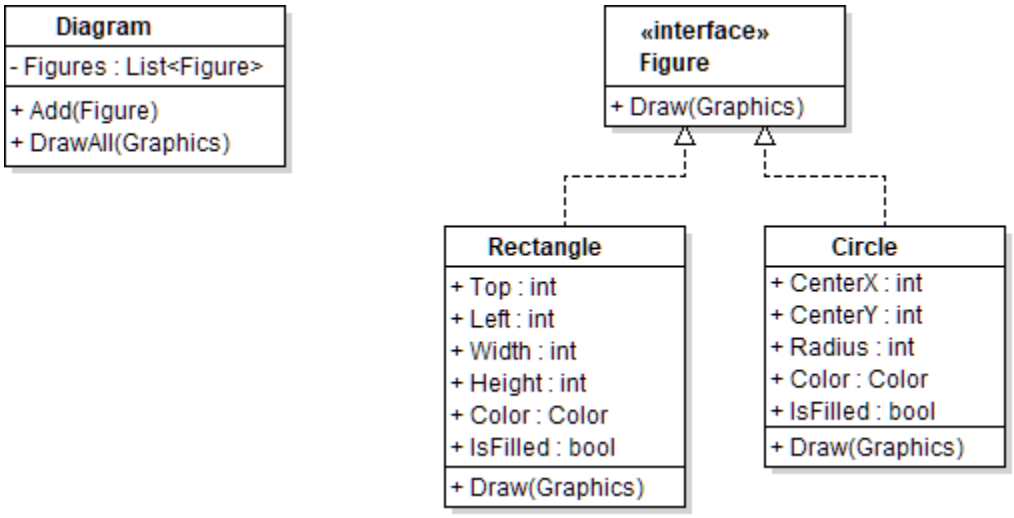
Verliezen we nu niet de voordelen van het static type checking mechanisme? Het voelt een beetje aan alsof we de compiler om de tuin leiden.

Het antwoord op die vraag is een luide "NEEN!". Het static type checking mechanisme

- controleert of types compatibel zijn, bv. bij toekenning en parameters van method oproepen
- laat enkel Draw method oproepen toe als we werken met Figure expressies
- verplicht ons een Draw method met code te voorzien in elke klasse die Figure-compatibel is
    - de klassen Rectangle en Circle hebben beiden een eigen Draw method met code

Deze twee puntjes samen, zorgen ervoor dat het mechanisme nog steeds kan garanderen dat er tijdens de uitvoering nooit een type fout kan voorkomen waarbij een object een method niet ondersteunt.

Op een klassendiagram ziet dit project er als volgt uit :



Let erop hoe de interface wordt voorgesteld, alsook de 'implementeert' relatie tussen de klassen en de Figure interface.

Dankzij de interface Figure hebben we alle voordelen bekomen van de vorige twee projecten :

1. een aparte klasse per soort figuur (Circle, Rectangle), die elk een hoge **cohesie** hebben
2. slechts één lijst en één Add method in klasse Diagram, die **weinig afhankelijkheden** heeft

Over het algemeen schrijft men trouwens niet dat een klasse "*Figure-compatibel*" is, maar wel dat de klasse "*de interface Figure implementeert*". Dit benadrukt dat de klasse code voorziet voor de methods uit de interface.

Doorgaans krijgen interfaces trouwens een hoofdletter i vooraan hun naam, dat maakt het makkelijker om ze van klassen te onderscheiden. Onze interface Figure zouden we dus normaliter IFigure noemen.

Nog eens herhalen dat een interface gerust meerdere methods mag bevatten.

Als we bv. ons fictief programma met **bewegende selecteerbare** figuren op deze manier zouden schrijven, moeten we de Simulate en Contains methods (zonder implementatie) aan de Figure interface toevoegen. Deze interface zal dan 3 methods bevatten!

De klassen Circle, Rectangle, Triangle, Diamond en Image zullen allen die interface (en z'n methods) implementeren zodat we alle soorten figuren aan onze Diagram objecten kunnen toevoegen.

Terzijde : het is perfect mogelijk dat een klasse meerdere interfaces implementeert, je moet dan wel de namen van de interfaces scheiden met komma's :

```
public class X : A, B, C {                    // klasse X implementeert interfaces A, B en C
    …
}
```

De klasse moet dan natuurlijk elke method van elke ondersteunde interface implementeren.

Het ontwerp perspectief :

Interfaces (als programmeertaal concept) zorgen voor flexibiliteit in het anders zeer rigide static type checking mechanisme, zonder de voordelen van dit mechanisme op te offeren.

Dankzij interfaces kunnen we op een veilige manier methods van objecten oproepen, zonder ons vast te pinnen op de specifieke klassen van die objecten.

Mind Blown?

Iets anders geformuleerd, het laat toe om in een klasse X bepaalde objecten te gebruiken zonder dat klasse X afhankelijk wordt van de klasse van die objecten. Bijvoorbeeld, Diagram kan met Circle objecten werken zonder dat klasse Diagram afhankelijk is van klasse Circle.

Interfaces zijn dus een manier om flexibeler om te kunnen gaan met de afhankelijkheden van klassen.

Een belangrijke vaststelling : omdat de klasse Diagram niet meer afhankelijk is van Rectangle of Circle, kan ze gecheckt en gecompileerd worden *zonder dat we over de klassen Circle of Rectangle* beschikken. De compiler heeft enkel de Figure interface nodig om klasse Diagram te verwerken.

We kunnen dus een gecompileerde versie van klasse Diagram aan andere programmeurs geven en deze zal qua typing perfect passen bij hun klassen die de Figure interface implementeren. En dit zonder dat we hun klassen ooit gezien hebben die misschien pas jaren later worden geschreven!

Dit lijkt een stukje compiler trivia, maar dit tijdsaspect is eigenlijk heel belangrijk! Het is een ontwerp techniek die heel vaak gebruikt wordt in libraries, b.v. de collectieklassen die meegeleverd worden met de .NET omgeving.

Op basis van interfaces uit die library kunnen we (jaren later) klassen schrijven die perfect aansluiten bij de klasse uit die library. Strak volgt een toepassing hiervan : de IComparer interface.

IComparer<T>

De klasse List<T> bevat Sort methods die de lijst voor ons zullen sorteren. Bij dit sorteren moeten elementen met elkaar vergeleken worden om hun onderlinge volgorde te bepalen.

Voor ingebouwde types zoals int, string en DateTime is er een 'natuurlijke grootte' die de volgorde bepaalt, maar wat als we elementen van onze eigen types willen sorteren? Bijvoorbeeld,

- een List<Student> sorteren op basis van hun leeftijd (oplopend)
    - vergelijk de familienamen van twee Student objecten
- een List<Afspraak> sorteren op basis van hun datum (chronologisch)
    - vergelijk de datums van twee Afspraak objecten
- een List<Cirkel> sorteren op basis van hun straal (groot naar klein)
    - vergelijk de stralen van twee Cirkel objecten

Je moet je realiseren dat die Sort method vele jaren geleden geschreven werd en dus onmogelijk details kan bevatten over onze Persoon, Afspraak of Cirkel objecten.

Hoe weet die oude Sort method hoe ze onze elementen moet vergelijken? In de documentatie van de Sort method (die met een IComparer<T> parameter) vind je de details. Er wordt gesorteerd van 'klein' naar 'groot' op basis van een meegegeven *comparer* object.

Aha! We moeten een *comparer* object meegeven dat telkens twee elementen met elkaar vergelijkt en kan bepalen wat hun onderlinge verhouding is (kleiner/gelijk/groter).

Het *comparer* object is op maat gemaakt voor de specifieke combinatie van het soort element (Persoon/Afspraak/Cirkel) en het criterium (oplopende leeftijd/chronologisch datum/straal van groot naar klein).

Voor elke combinatie die we willen, zullen we dus een aparte *comparer* klasse moeten schrijven die de IComparer interface implementeert. Dit soort klasse is gelukkig vrij eenvoudig 😉

Handig! Men heeft een List<T> klasse voorzien die lijsten van onze objecten kan sorteren op basis van criteria die wij voorzien, en dit reeds vele jaren voordat wij de klassen voor deze objecten schrijven.

De interface **IComparer<T>** zit in namespace System.Collections.Generic en ziet er zo uit :

```
public interface IComparer < T > {
    public int Compare(T x, T y);
}
```

De documentatie vind je op

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1

De Compare method moet twee objecten x en y (van type T) met elkaar vergelijken en via de return value aangeven wat het verband is ertussen :

- indien x < y moet de return value < 0 zijn
- indien x == y moet de return value == 0 zijn
- indien x > y moet de return value > 0 zijn

De Sort method uit List<T> zal bij het sorteren de Compare method van het *comparer* object gebruiken, om te achterhalen welk van twee elementen 'kleiner' of 'minder' is en eerder moet komen.

Wat 'kleiner' of 'minder' precies betekent, hangt van af de context : we hebben een bepaalde volgorde voor ogen (op leeftijd, alfabetisch op naam, op lengte, op prijs, etc.) en dan interpreteren we 'kleiner' of 'minder' gewoon als 'eerder' in die volgorde.

Om een List<Persoon> te sorteren op basis van de leeftijd van de personen (jong naar oud), moeten we dus een klasse PersoonLeeftijdComparer schrijven die de ICompare<Persoon> interface implementeert :

```
public class PersoonLeeftijdComparer : IComparer<Persoon> {
    public int Compare(Persoon x, Persoon y) {
        return x.Leeftijd – y.Leeftijd;
    }
}
```

Merk op dat we hier voor de eenvoud geen rekening houden met mogelijke null parameters!

Vergewis je ervan dat de return value telkens klopt (bv. indien x jonger is zal de return value < 0 zijn).

Als we nu een List<Persoon> willen sorteren op basis van hun leeftijd, schrijven we :

```
List<Persoon> personen = …
…
PersoonLeeftijdComparer comparer = new PersoonLeeftijdComparer();
personen.Sort( comparer );
```

Al bij al een zeer simpele klasse PersoonLeeftijdComparer en het gebruik ervan is ook heel eenvoudig.

De meeste ingebouwde types met een 'natuurlijke grootte' zoals int, string en DateTime hebben een eigen CompareTo method die goed van pas komt bij het schrijven van een comparer klasse.

```
int i1 = 5;
int i2 = 10;
Console.WriteLine( i1.CompareTo( i2 ) );                    // output : -1

string s1 = "Hello";
string s2 = "World";
Console.WriteLine( s1.CompareTo( s2 ) );                    // output : -1

DateTime dt1 = DateTime.Now;
DateTime dt2 = dt1.AddMinutes(10);
Console.WriteLine( dt1.CompareTo( dt2 ) );                  // output : -1
```

We zouden klasse PersoonLeeftijdComparer duidelijker kunnen schrijven met

```
public class PersoonLeeftijdComparer : IComparer<Persoon> {
    public int Compare(Persoon x, Persoon y) {
        return x.Leeftijd.CompareTo( y.Leeftijd );
    }
}
```

Let op : als de List<T> ook null waarden kan bevatten, zul je daar in je comparer klasse rekening mee moeten houden (lees : checken met een 'if' statement). Volgens de documentatie van Compare moet een null als kleiner gezien worden dan elke andere (niet null) waarde. Bovendien worden twee null waarden als 'even groot' beschouwd, dus in dat geval moet de return value == 0 zijn.

Veronderstel dat we een List<Persoon> willen sorteren op basis van de .Naam property van de Persoon objecten en dat deze lijst ook null waarden kan bevatten.

Eerst schrijven we een klasse PersoonNaamComparer die rekening houdt met null waarden in de lijst :

```
public class PersoonNaamComparer : IComparer<Persoon> {
    public int Compare(Persoon x, Persoon y) {
        if (x == null && y == null ) {
                return 0;
        } else if (x == null) {
                return -1;
        } else if (y == null) {
                return 1;
        } else {
                return x.Naam.CompareTo( y.Naam );
        }
    }
}
```

Daarna kunnen we de lijst sorteren met :

```
List<Persoon> personen = …
…
PersoonNaamComparer comparer = new PersoonNaamComparer();
personen.Sort( comparer );
```

Het ontwerp perspectief

Om het sorteren van eender welk soort elementen mogelijk te maken, hebben de auteurs van List<T> een interface IComparer<T> gedefinieerd. Onze eigen *comparer* klassen implementeren deze interface, waardoor onze code om elementen te vergelijken netjes integreert met de Sort code van List<T>.

Dit demonstreert nogmaals de het nut van interfaces in een statisch getypeerde programmeertaal :

> ***om code te kunnen schrijven die methods oproept van objecten***
> ***wiens precieze klasse we niet kunnen (of willen) kennen.***

Een interface legt dus een soort protocol vast om met objecten te kunnen communiceren, zonder dat we de precieze klasse van die objecten moeten kennen.

De code die geschreven wordt op basis van een interface, bv.

- List<T>.Sort() op basis van IComparer<T>
- Diagram.DrawAll(Graphics) op basis van Figure

kan dus met eender welk object samenwerken op voorwaarde dat het dit protocol ondersteunt.

Dit maakt het mogelijk om de nodige flexibiliteit in onze code in te bouwen die latere aanpassingen en uitbreidingen kan vergemakkelijken.