Inżynieria oprogramowania
Mini-projekt

Wprowadzenie

W ramach zadania studenci otrzymują zestaw diagramów UML składający się na projekt prostej aplikacji typu program graficzny oraz szkielet interfejsu użytkownika.
Celem zajęć jest implementacja otrzymanego projektu do w pełni działającego programu. Musi on umożliwiać rysowanie figur, a także zapisywanie i wczytywanie już wykonanych rysunków.

Szkielet kodu jest przygotowany w języku C# na platformę .NET Framework i udostępniony w postaci projektu Visual Studio .NET 2010.
Zalecany format plików z zapisywanymi obrazkami to XML.

Opis zadania

Celem jest napisanie aplikacji graficznej umożliwiającej wykonywanie prostych rysunków. Rysunki składają się z figur, które mogą być jednym z trzech rodzajów figur:

- linie proste,
- prostokąty,
- okręgi.

Rysowanie na ekranie polega na wciśnięciu przycisku myszki w punkcie początkowym i puszczeniu w punkcie końcowym. W przypadku okręgów punkt początkowy jest interpretowany jako środek okręgu, a punkt końcowy wyznacza jego promień.
Wykonany rysunek można zapisać do pliku, a rysunek zapisany można wczytać. Wymagane jest użycie formatu XML do zapisu plików.
Program działa pod Windows z pełnym interfejsem graficznym.

Projekt

Diagram przypadków użycia

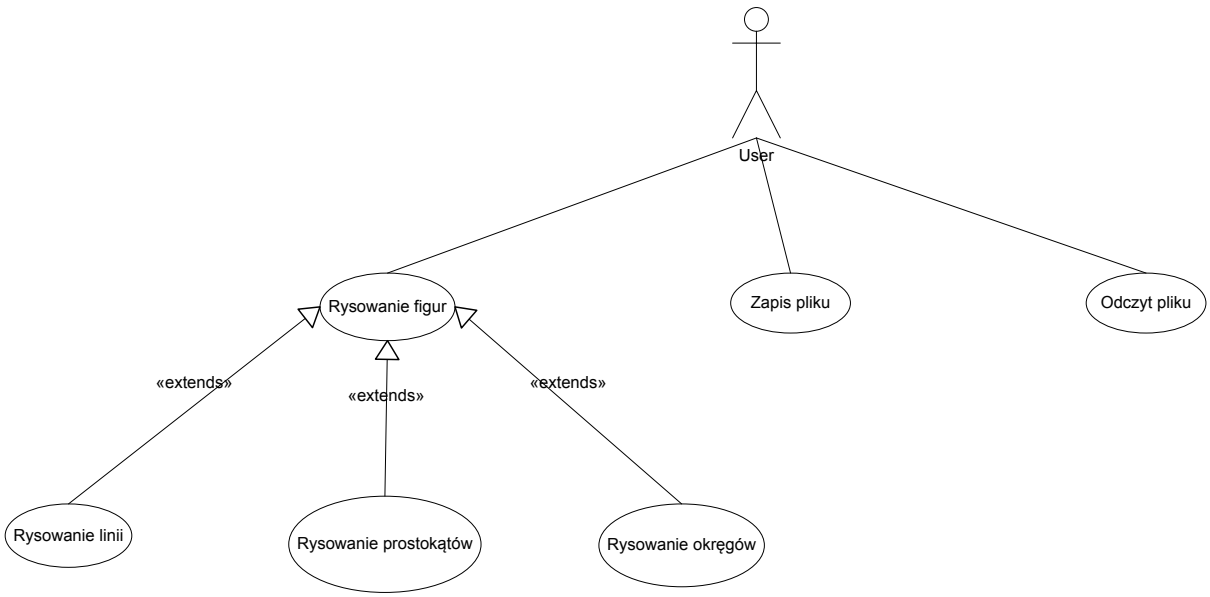


Diagram klas

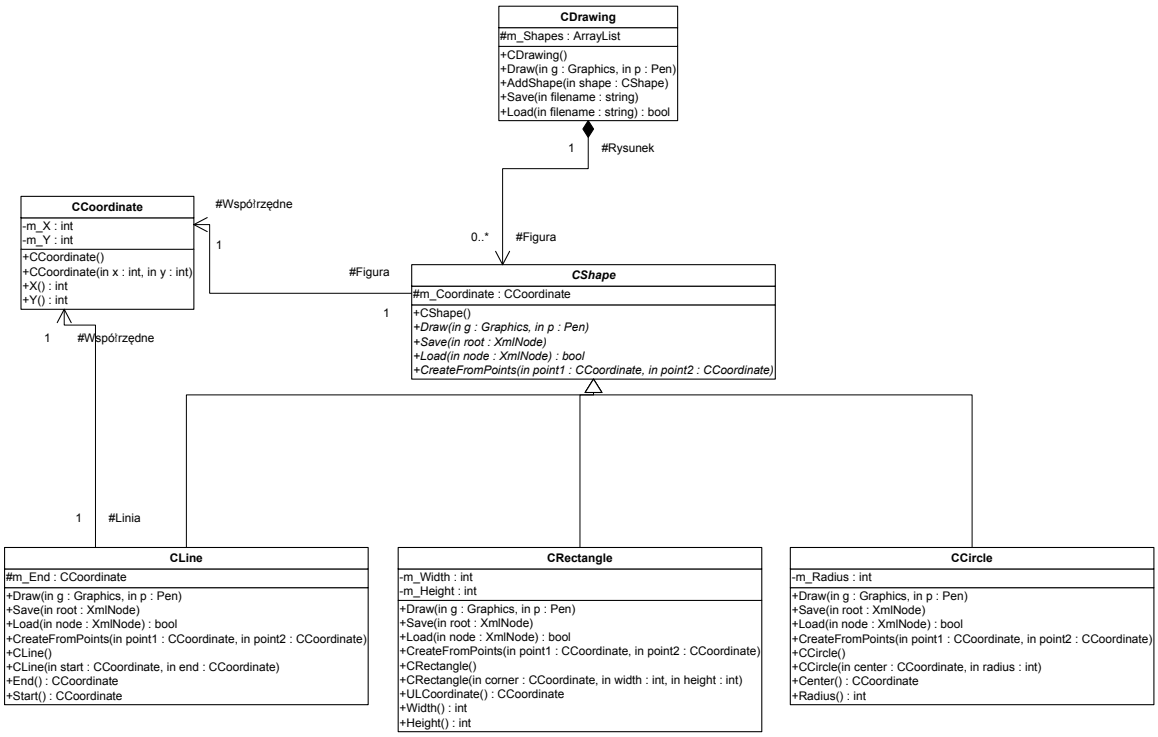


Diagram sekwencji dla rysowania figur

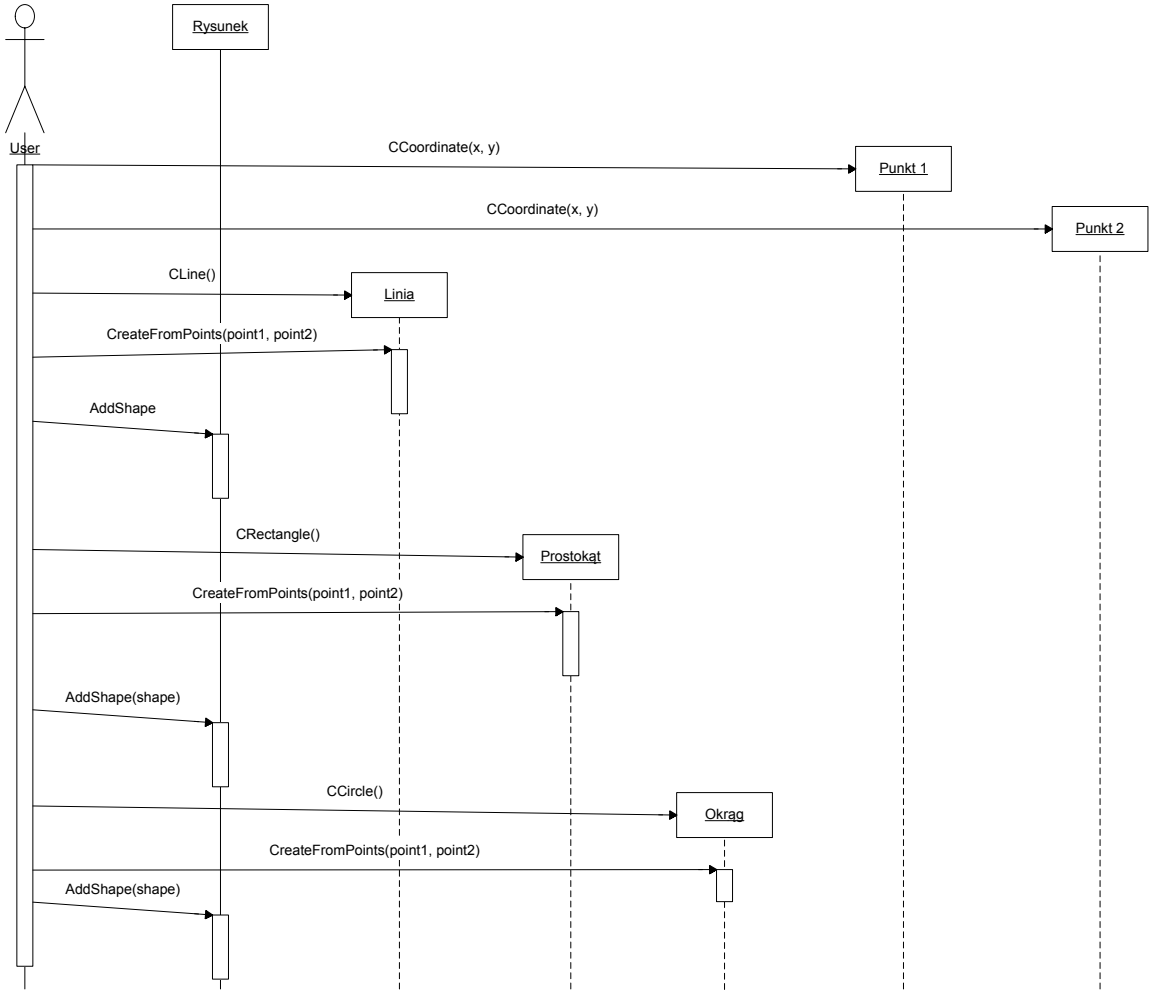


Diagram sekwencji dla wyświetlania obrazka

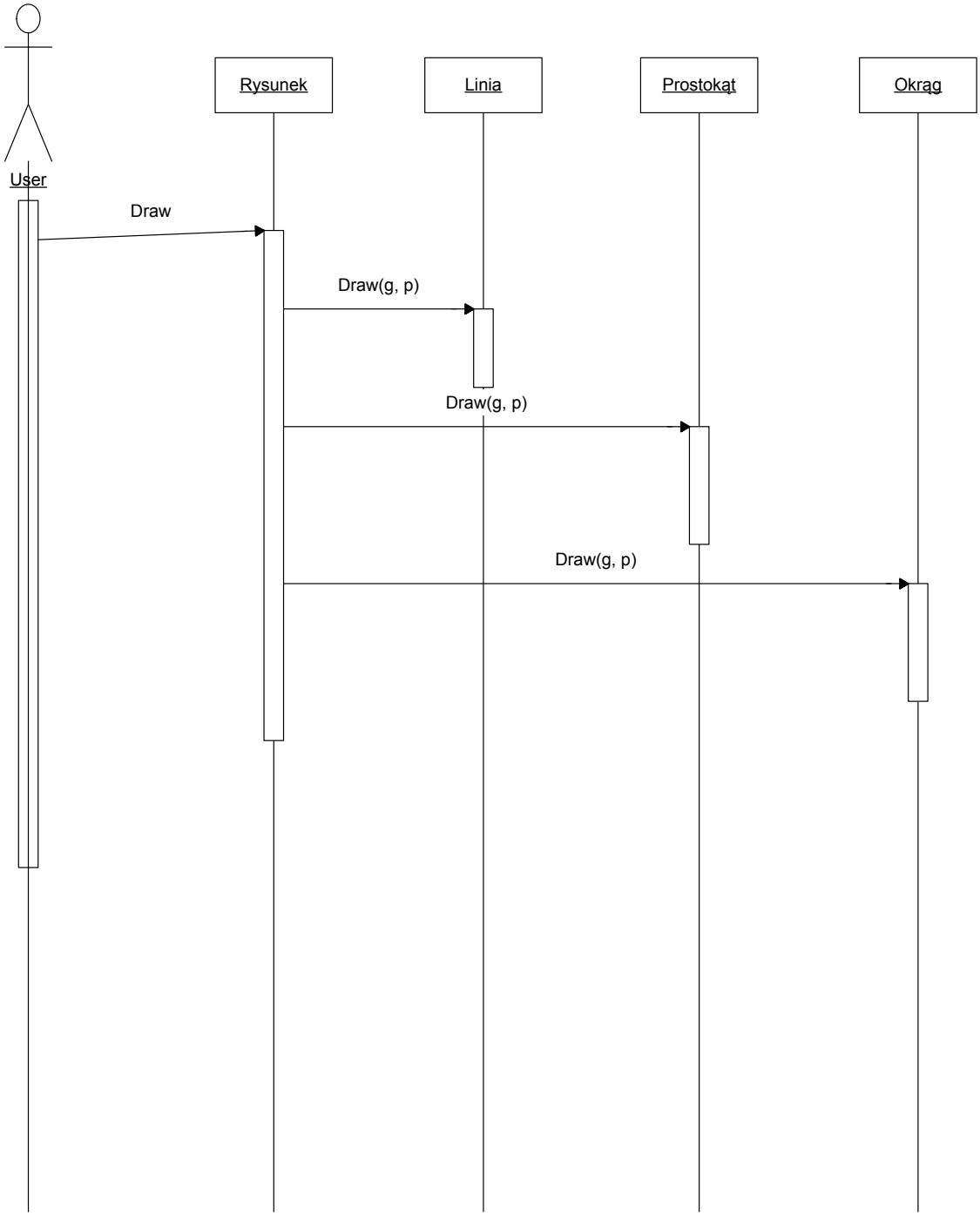


Diagram sekwencji dla zapisu do pliku

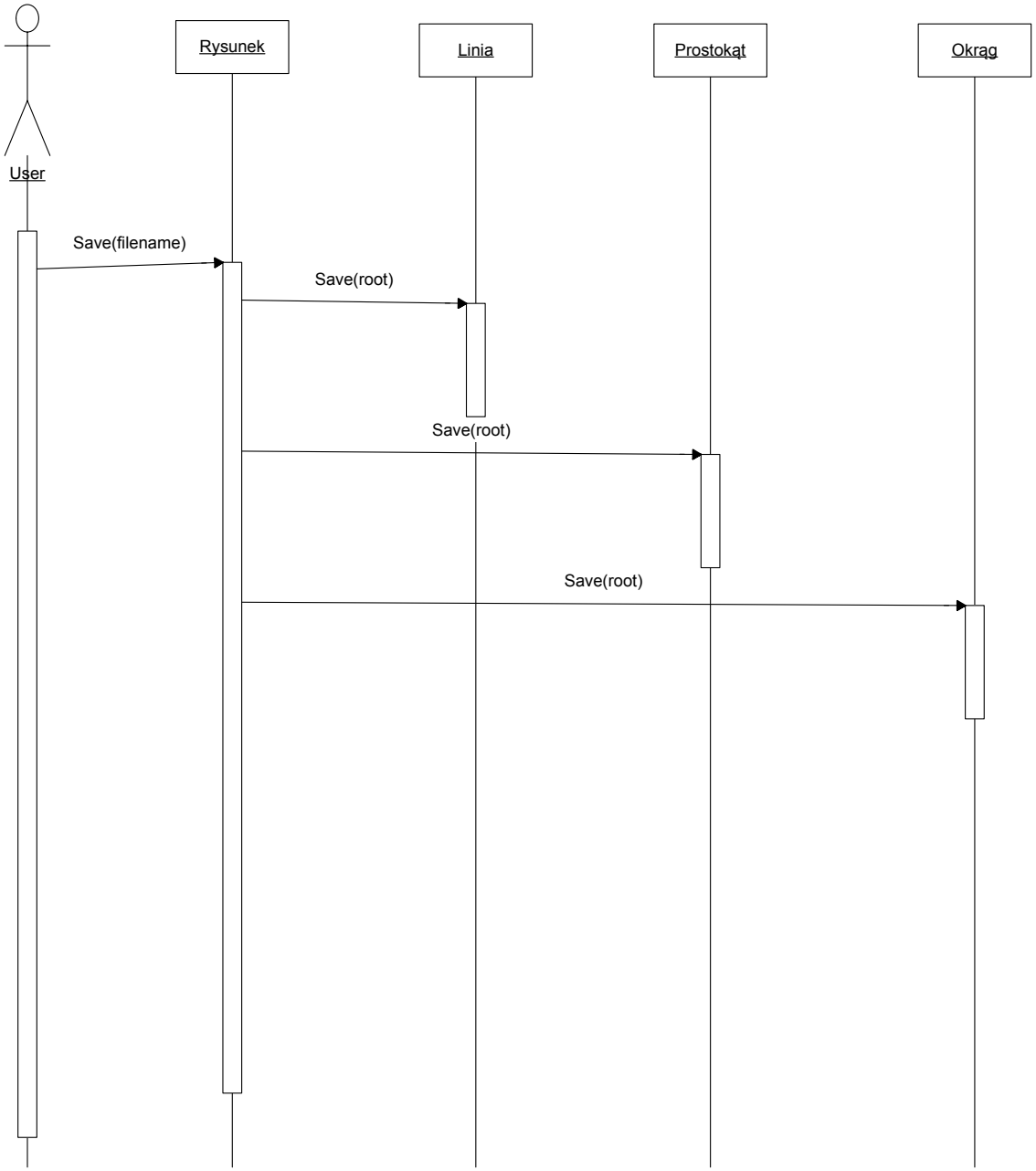
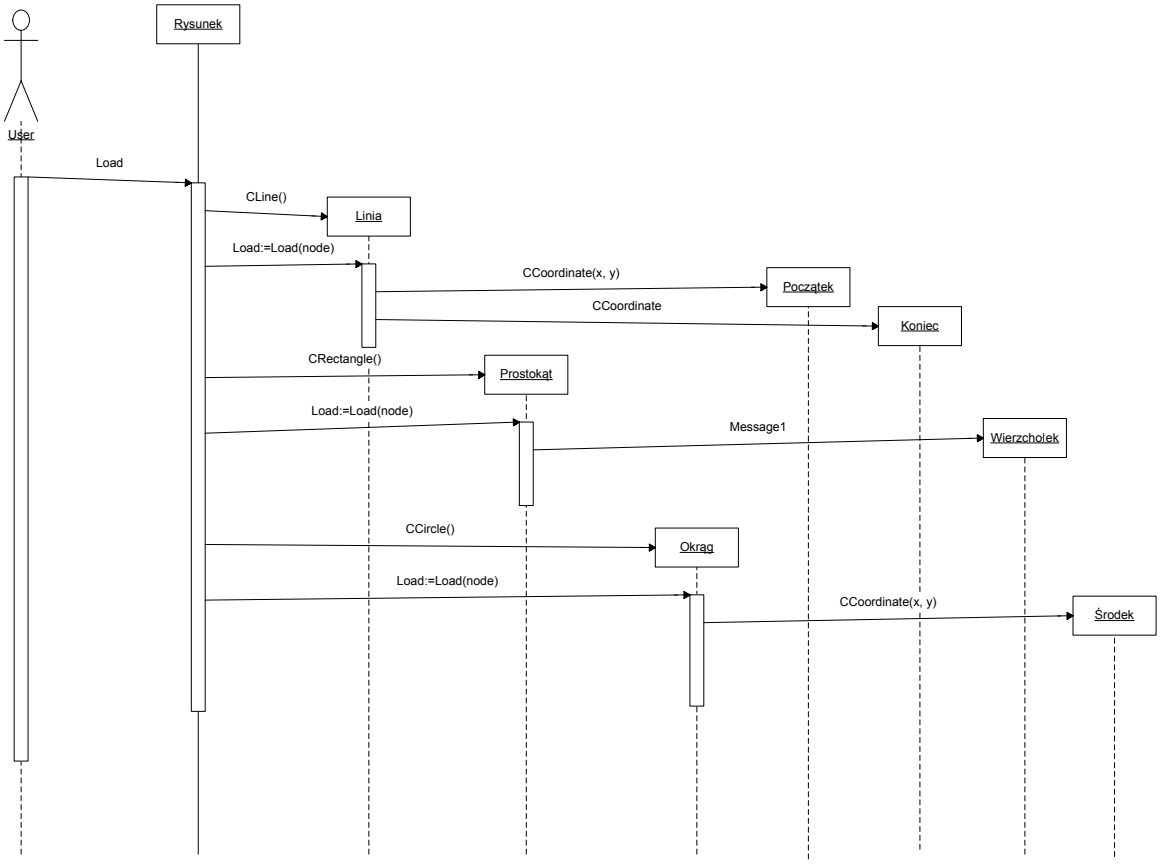


Diagram sekwencji dla zapisu do pliku



Uwagi pomocne przy realizacji

Język C#

Język C# jest językiem pośrednim pomiędzy C++ a Javą.
Wszystkie typy danych w C# są obiektami. Nie ma samodzielnych funkcji, każda funkcja jest metodą jednej z klas. Nie używa się wskaźników i nie jest konieczne zwalnianie pamięci.
Składnia języka jest niemal identyczna z C++.
Do realizacji związków wielokrotnych w przypadku języka C# warto jest wykorzystać wbudowany typ `List<>`, będący dynamiczną tablicą zdolną do przechowywania dowolnych obiektów podanego typu.

Obiekty typu List musi zostać zainicjalizowany:

```
List<Linia> linie=new List<Linia>();
```

Następnie można się nim posługiwać podobnie jak tablicą. Aby dodać obiekt do kolekcji używa się metody `Add`:

```
kolekcja.Add(linia);
```

Obiekty można indeksować korzystając z nawiasów `[]`, albo wykorzystać instrukcję `foreach`:

```
foreach (Linia l in linie)
{
      l.Draw();
}
```

Taka instrukcja spowoduje wywołanie metody `Draw` dla każdego obiektu umieszczonego w kolekcji należącego do klasy `Linia`.
Dostęp do pól klasy realizuje się za pomocą tzw. właściwości (ang. *Properties*). Pole deklaruje się jako prywatne lub chronione, np.:

```
private int m_Pole;
```

a następnie dodaje się specjalną konstrukcję (w najprostszym przypadku):

```
public int Pole
{
	get
	{
		return m_Pole;
	}
	set
	{
		m_Pole=value;
	}
}
```

Można wtedy odwoływać się do właściwości `Pole` klasy, tak jak do pola publicznego jednak wewnątrz kodu klasa może sprawdzać czy przypisywane wartości spełniają jej wymagania. Jeżeli pominiemy instrukcję `get` pole będzie można tylko zapisywać, a jeżeli `set` to tylko odczytywać.

Przy tworzeniu oprogramowania w języku C# zakłada się właściwe komentowanie elementów. Zdefiniowane są w środowisku specjalne komentarze zaczynające się od `///` i zawierające określone elementy. Takie komentarze powinno się stosować do wszystkich publicznych elementów klas (pól, właściwości, metod), co nie znaczy, że nie można ich stosować do elementów prywatnych. Taki komentarz dla metody wygląda np.:

```
/// <summary>
/// Tworzy linię z dwóch punktów.
/// </summary>
/// <param name="point1">Punkt pierwszy.</param>
/// <param name="point2">Punkt drugi.</param>
public override void CreateFromPoints(CCoordinate point1, CCoordinate
point2)
{
}
```

Środowisko Visual Studio w pełni wspomaga tworzenie takich komentarzy. Wystarczy przed metodą wpisać tylko `///`, a edytor doda szkielet komentarza automatycznie.

Dziedziczenie zapisuje się podobnie jak w C++, np.

```
public class CDyrektor : CPracownik
{
}
```

Natomiast jeżeli chcemy przeciążyć metodę z klasy, po której dziedziczymy konieczne jest dodanie słowa kluczowego `override`, jak na przykładzie z komentarzem. W celu uzyskania polimorfizmu metody muszą być oznaczone jako `virtual`.

Jeżeli klasa ma być abstrakcyjna, to dodaje się w jej deklaracji słowo kluczowe `abstract`, np.:

```
public abstract class Klasa
{
}
```

Dodatkowo, jeżeli abstrakcyjna jest metoda, jej deklarację też poprzedza się tym słowem kluczowym.

```
public abstract void Rysuj();
```

Rysowanie

W przypadku gdy w definiowanej klasie chcemy się posługiwać procedurami rysowania na ekranie w systemie Windows do każdego pliku należy dodać na początku deklarację:

```
using System.Drawing;
```

Do rysowania linii należy wykorzystać metodę `DrawLine` klasy `Graphics` w następujący sposób:

```
g.DrawLine(p,x1,y1,x2,y2);
```

gdzie `g` jest obiektem klasy `Graphics`, `p` jest obiektem klasy `Pen`, a `Xn` i `Yn` współrzędnymi na ekranie.

Do rysowania prostokątu służy podobna metoda `DrawRectangle`:

```
g.DrawRectangle(p,x,y,width,height);
```

gdzie `width` i `height` to szerokość i wysokość prostokąta.

Do rysowania okręgu należy użyć metody DrawEllipse:

```
g.DrawEllipse(p,x,y,width,height);
```

gdzie parametry metody wyznaczają prostokąt (w szczególności kwadrat) w który wpisany jest okrąg.

Obsługa zdarzeń

Do rysownia wykorzystamy dwa zdarzenia: MouseDown i MouseUp. Pierwsze występuje w momencie wciśnięcia klawisza mysz, drugie gdy klawisz myszy zostanie puszczony. W kodzie szkieletu interfejsu umieszczone są już metody obsługujące te zdarzenia:

```
private void picObrazek_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
private void picObrazek_MouseUp(object sender,
System.Windows.Forms.MouseEventArgs e)
```

XML

W modułach, w których chcemy korzystać z obsługi plików XML należy na początku pliku dodać deklarację:

```
using System.Xml;
```

Aby utworzyć plik XML w pamięci konieczne jest wykonanie następujących instrukcji:

```
XmlDocument doc=new XmlDocument();
XmlDeclaration dec=doc.CreateXmlDeclaration("1.0", "", "yes");
doc.PrependChild(dec);
XmlElement rootNode=doc.CreateElement("Drawing");
doc.AppendChild(rootNode);
```

W pierwszej linii tworzymy nowy obiekt klasy `XmlDocument`. Następnie konieczne jest dodanie do pliku deklaracji XML, czemu służą dwie następne linie. W czwartej linii tworzymy nowy węzeł drzewa XML o nazwie „`Drawing`”, a następnie dodajemy go do dokumentu. Węzeł ten będzie stanowił korzeń dokumentu.

Aby ostatecznie zapisać plik na dysk konieczne jest wydanie polecenia:

```
doc.Save(nazwa_pliku);
```

gdzie `nazwa_pliku` jest łańcuchem znaków (typ `string`);

W celu utworzenia węzłów w dokumencie odpowiadających poszczególnym figurom i zakładając, że znamy węzeł-korzeń `root`, kod będzie wyglądał mniej więcej tak:

```
XmlDocument doc=root.OwnerDocument;
XmlElement shapeNode=doc.CreateElement("Line");
shapeNode.SetAttribute("StartX",m_Coordinate.X.ToString());
…
root.AppendChild(shapeNode);
```

W pierwszej linii musimy pobrać odnośnik do dokumentu XML. Następnie tworzymy nowy element XML o nazwie „`Line`”. Dla innych figur nazwa powinna być inna. W dalszej kolejności do utworzonego węzła przypisujemy atrybuty opisujące własności figury, np. tutaj współrzędne początku figury. Na koniec dodajemy utworzony węzeł jako potomka węzła-korzenia.

W przypadku wczytywania dokumentu XML musimy najpierw utworzyć obiekt odpowiadający dokumentowi i wczytać go z pliku:

```
XmlDocument doc=new XmlDocument();
XmlNode rootNode;
doc.Load(filename);
rootNode=doc.ChildNodes[1];
```

Przy okazji zapamiętamy sobie wskazanie na węzeł-korzeń. Następnie możemy sprawdzić, czy wczytano dokument właściwego typu:

```
if (rootNode.Name!="Drawing")
      return false;
```

Jeżeli nazwa węzła-korzenia jest inna niż „Drawing” oznacza to, że plik jest niewłaściwy i możemy zakończyć proces odczytu.
Aby wczytać poszczególne figury można posłużyć się instrukcją foreach:

```
foreach (XmlNode shape in rootNode.ChildNodes)
{
      shape. …
}
```

Aby odczytać wartości zapisane w atrybutach węzła XML należy skorzystać z polecenia:

```
int X=int.Parse(node.Attributes["StartX"].Value);
```

gdzie `StartX` jest nazwą atrybutu. Konieczna przy tym jest konwersja z typu `string` na `int` (`int.Parse`).

Po zapisaniu przykładowy plik powinien wyglądać następująco:

```
<?xml version="1.0" standalone="yes"?>
<Drawing>
  <Line StartX="110" StartY="79" EndX="225" EndY="174" />
  <Line StartX="62" StartY="193" EndX="243" EndY="102" />
  <Rectangle X="75" Y="113" Width="101" Height="73" />
  <Rectangle X="163" Y="206" Width="61" Height="42" />
  <Rectangle X="286" Y="111" Width="36" Height="43" />
  <Circle CenterX="46" CenterY="27" Radius="102" />
  <Circle CenterX="212" CenterY="126" Radius="29" />
  <Circle CenterX="260" CenterY="204" Radius="18" />
  <Line StartX="48" StartY="173" EndX="233" EndY="181" />
  <Line StartX="247" StartY="210" EndX="301" EndY="172" />
  <Line StartX="313" StartY="99" EndX="312" EndY="179" />
  <Line StartX="146" StartY="224" EndX="243" EndY="236" />
  <Rectangle X="46" Y="215" Width="60" Height="32" />
  <Rectangle X="110" Y="178" Width="111" Height="90" />
  <Rectangle X="217" Y="50" Width="104" Height="103" />
  <Rectangle X="113" Y="57" Width="103" Height="64" />
  <Rectangle X="329" Y="45" Width="43" Height="51" />
  <Rectangle X="248" Y="128" Width="90" Height="21" />
  <Circle CenterX="84" CenterY="132" Radius="65" />
  <Circle CenterX="241" CenterY="203" Radius="32" />
  <Circle CenterX="258" CenterY="252" Radius="28" />
</Drawing>
```