

GEM — projekt architektury systemu

wersja 1.2

Maria Donten
Marek Grabowski
Piotr Hofman
Kuba Pochrybniak

Spis treści

1. Wprowadzenie	3
1.1. Cel	3
1.2. Zakres	3
1.3. Definicje	3
1.4. Załączniki	3
1.5. Omówienie reszty dokumentu	3
2. Prezentacja architektury systemu	3
2.1. Widoki	4
3. Założenia i zależności	5
3.1. Założenia	5
3.2. Użycie darmowych narzędzi	6
3.3. Bezpieczeństwo	6
3.4. Niezawodność	6
4. Przegląd przypadków użycia	6
4.1. Szybki podgląd	6
4.2. Rysowanie krzywych	7
4.3. Dodanie etykiety	7
5. Dekompozycja logiczna systemu	8
5.1. Omówienie	8
5.2. Najważniejsze komponenty	8
6. Dekompozycja na procesy	9
7. Instalacja systemu	10
8. Implementacja systemu	10
8.1. Omówienie	10
8.2. Warstwy	11
8.3. Diagramy klas programu	28
9. Przechowywane dane	30
9.1. Przechowywanie rysunków	30
9.2. Pliki konfiguracyjne programu	30
10. Wydajność systemu	31
11. Jakość	31
11.1. Przenośność	31
11.2. Możliwości rozbudowywania systemu	31
11.3. Bezpieczeństwo	32
12. Historia zmian	32

1. Wprowadzenie

1.1. Cel

Celem niniejszego dokumentu jest przedstawienie projektu architektury programu GEM.

1.2. Zakres

Dokument przedstawia kilka różnych aspektów zarysu architektury programu GEM. Program jest przeznaczony do tworzenia grafiki do prac naukowych z zakresu nauk ścisłych. Wobec tego istotną rzeczą jest analiza reprezentacji obiektów graficznych w systemie.

1.3. Definicje

Podane w załączniku „Słownik projektu GEM”, uaktualnianym na bieżąco, aby zawierał definicje wymagane przez powstające kolejno dokumenty.

1.4. Załączniki

Dokument „Słownik projektu GEM”, wersja 4.0 (od poprzedniego dokumentu nie przybyły żadne nowe definicje).

1.5. Omówienie reszty dokumentu

W rozdziale 2 i 3 zostały przedstawione podstawowe założenia i ogólny schemat architektury powstającego systemu. W następnych rozdziałach opisane zostały następujące aspekty architektury:

- zależności między komponentami z punktu widzenia kilku przykładowych przypadków użycia,
- podział systemu na warstwy,
- widok procesów w systemie,
- przegląd klas implementacyjnych programu,
- przechowywane dane.

Ponadto omówiona została instalacja systemu, a także wydajność i problemy związane z jakością powstającego programu.

2. Prezentacja architektury systemu

System zbudowany jest, jak wynika z poprzednich dokumentów, z następujących składowych:

Edytory graficzny

- Pozwala na wygodne wprowadzanie danych do programu przez użytkownika.
- Przetrzymuje wszystkie dane często wykorzystywane przez użytkownika.
- Zamienia dane wprowadzone przez użytkownika w trybie graficznym na kod META-POST.

Edytor tekstowy

- Pozwala na wprowadzanie danych przez zaawansowanych użytkowników.
- Wspomaga szukanie błędów w języku METAPOST przez kolorowanie składni (opcjonalnie).

Kompilator i konwerter

- Kompiluje kod METAPOST do gotowego rysunku.
- Generuje rysunek w jednym z kilku formatów wektorowych lub rastrowych.

Aplikacja sieciowa

- Pozwala na komunikację między dwoma komputerami podłączonymi do internetu.

2.1. Widoki

W dalszej części dokumentu architektura systemu przedstawiona jest w kilku widokach opisanych poniżej:

2.1.1. Perspektywa przypadków użycia

Opisuje kilka fundamentalnych przypadków użycia, wybranych przez zespół spośród wszystkich opisanych w dokumencie Techniczne przypadki użycia. Są to:

1. Szybki podgląd,
2. rysowanie krzywych,
3. dodanie etykiety.

2.1.2. Dekompozycja logiczna systemu

Opisuje podział systemu na podsystemy wraz z przydzieleniem ich do różnych warstw systemu. Podsystemy wyróżnione w systemie to:

1. Interfejs,
2. edytor graficzny,
3. edytor tekstowy,
4. kompilator z konwerterem,
5. aplikacja sieciowa.

2.1.3. Perspektywa procesów

Opisuje podział systemu na procesy, czyli niezależne przebiegi sterowania. W wypadku projektu GEM można wyróżnić następujące procesy:

Edytor z interfejsem główny proces systemu odpowiedzialny za komunikację z użytkownikiem.

Kompilator zewnętrzny program kompilujący kod METAPOST do pliku pdf.

Konwerter zewnętrzny program zmieniający formaty plików.

Serwer XML-RPC serwer Apache XML-RPC działający na maszynie i odbierający zewnętrzne polecenia.

2.1.4. Implementacja

Podaje szczegółowy podział systemu na klasy implementacyjne, razem z interfejsami. Podstawowy podział podsystemów na klasy wygląda następująco:

Interfejs

- MVC.

Edytor graficzny

- Klasy obiektów graficznych,
- klasy grup obiektów graficznych,
- klasa edytora.

Edytor tekstowy

- Klasa słów kluczowych (opcjonalnie),
- klasa reprezentująca otwarte pliki,
- klasa edytora.

Przeglądarka

- Klasa przeglądarki.

Kompilator i konwerter

- Klasa odpowiedzialna za kompilację kodu METAPOST,
- klasy odpowiedzialne za konwersję jednego formatu danych na inny.

Aplikacja sieciowa

- Klasa klienta,
- klasa serwera.

2.1.5. Widok warstw

W związku z podziałem na podsystemy można wyróżnić następujące warstwy systemu:

Warstwa sieciowa do komunikacji między programami uruchomionymi na różnych komputerach.

Warstwa logiki systemu do przechowywania danych i manipulacji nimi (dodawanie nowych obiektów, dbanie o stos wycofań, konwersja i kompilacja danych itp). W skład tej warstwy wchodzi zarówno kompilator, konwerter jak i modele edytorów.

Warstwa interfejsu do komunikacji między programem, a użytkownikiem. Dbą o poprawną interpretację poleceń użytkownika.

2.1.6. Instalacja

Opisuje proces instalacji systemu na komputerze. System GEM, jako system praktycznie jednoużytkownikowy, jest stosunkowo prosty do instalacji.

3. Założenia i zależności

3.1. Założenia

Zespół zakłada, że użytkownik zna język \LaTeX i umie się nim w miarę sprawnie posługiwać. Ponadto użytkownik musi mieć zainstalowany kompilator języka \LaTeX i META-

POST (dystrybucja T_EXLive 2003 lub nowsza; ze starszymi dystrybucjami program może nie pracować w pełni poprawnie).

Ponadto zespół założył, że system będzie wykorzystywany przez matematyków w celu tworzenia rysunków matematycznych. System został zaprojektowany tak, aby umożliwiał tworzenie właśnie taki rodzaj grafiki. Używanie w innych celach może być utrudnione przez przyjęte rozwiązania.

3.2. Użycie darmowych narzędzi

W projekt zostaną włączone następujące zewnętrzne narzędzia:

- Kompilator języka METAPOST,
- kompilator języka L^AT_EX,
- biblioteka Apache XML-RPC,
- biblioteki do konwersji formatów graficznych,
- standardowe biblioteki JSE.

3.3. Bezpieczeństwo

Kwestia bezpieczeństwa jest bardzo ważną sprawą, ale ponieważ system jest przeznaczony do używania na komputerze osobistym użytkownika, nie przewiduje się w tej kwestii większych problemów.

W przypadku komunikacji sieciowej trzeba zadbać, aby z jednym serwerem mógł się połączyć jeden klient. Zapewniane jest to przez ustalenie identyfikatora sesji przy próbie połączenia. Komunikacja sieciowa nie jest szyfrowana, gdyż nie pojawiają się w niej żadne dane wymagające ochrony.

3.4. Niezawodność

Projekt jest przeznaczony do używania na komputerze osobistym przez pojedynczych użytkowników, więc niezawodność schodzi na drugi plan wobec wygody użytkownika.

4. Przegląd przypadków użycia

W niniejszym dokumencie prezentujemy kilka przypadków użycia, które najlepiej oddają zależności między różnymi częściami programu GEM. Wszystkie przypadki użycia są wymienione w dokumencie „Przypadki użycia”.

4.1. Szybki podgląd

4.1.1. Krótki opis

Możliwość uzyskania szybkiego podglądu bardzo ułatwia pracę nad szczegółami rysunku. Ten przypadek jest ważny ze względu na współpracę kilku modułów ze sobą.

4.1.2. Czynności

1. Użytkownik wybiera opcję z menu „Podgląd”.
2. Klasy implementujące interfejs informują edytor graficzny o zarejestrowanym zdarzeniu, wywołując metodę `do_podglądu`.

3. Wewnątrz metody `do_podglądu` obiekt `edytor_graficzny` wywołuje odpowiednią metodę klasy `Grafika2MP`, tłumaczącą wewnętrzny format edytora na kod METAPOST. W efekcie edytor graficzny zna nazwę pliku tymczasowego, w którym jest zapisany przetłumaczony kod.
 4. Obiekt `edytor_graficzny` przekazuje przeglądarce parametry pliku z kodem języka METAPOST, wywołując metodę `podgląd_mp(ścieżka_pliku)`.
 5. Przeglądarka wywołuje metodę kompilatora `kompiluj_mp(ścieżka_pliku)`.
 6. Obiekt `kompilator` na podstawie pliku tekstowego tworzy plik graficzny i oddaje jego nazwę przeglądarce.
 7. Rysunek jest wyświetlany w oknie przeglądarki.
-

4.2. Rysowanie krzywych

4.2.1. Krótki opis

Krzywe są podstawą każdego rysunku — najczęściej operujemy na znanych nam figurach geometrycznych. Do rysowania krzywych specjalnego kształtu (prostokątów, elips etc.) będą specjalne narzędzia, ułatwiające tę czynność; tutaj przedstawiamy działanie bardziej skomplikowanego narzędzia, pozwalającego na uzyskanie krzywej dowolnego kształtu.

4.2.2. Czynności

1. Użytkownik wybiera narzędzie do rysowania krzywych. Obiekt reprezentujący aktualny rysunek rejestruje informację o wyborze narzędzia (ze względu na planowaną możliwość edycji wielu rysunków, informacje o narzędziu i tymczasowe informacje o właśnie tworzonych przez użytkownika obiektach trzymane są wewnątrz obiektu reprezentującego rysunek).
 2. Użytkownik przeciąga myszą. Tworzony jest obiekt klasy `krzywa`, składający się z jednego węzła. Obiekt jest dodawany do struktur rysunku. Parametry pierwszego wektora sterującego są tymczasowo zapamiętane.
 3. Użytkownik (być może wielokrotnie) przeciąga myszą. Za każdym razem tworzony jest kolejny węzeł, kolejny łuk krzywej, oba są dodawane do krzywej; wektor sterujący powstały przez ostatnie przeciągnięcie myszą jest zapamiętywany, zaś wektor sterujący w węźle zamykającym ostatni łuk krzywej jest do niego przeciwny. Po dodaniu nowego łuku krzywej obiekt reprezentujący rysunek generuje zdarzenie oznaczające konieczność odświeżenia obrazu.
-

4.3. Dodanie etykiety

4.3.1. Krótki opis

Program umożliwia dodanie do rysunku etykiety z tekstem, który może zawierać symbole matematyczne. Tekst etykiety jest edytowany tak, jak w systemie \LaTeX : w trybie zwykłym lub matematycznym, otwieranym i zamykanym znakiem „\$”. Ten przypadek użycia rozpatruje wyłącznie czynności podstawowe dla dodania nowej etykiety.

4.3.2. Czynności

Jeśli więcej niż jeden obiekt jest zaznaczony, etykieta zostanie dodana tylko do jednego z nich. System ustala pewną kolejność obiektów. Jest to potrzebne między innymi do obsługi **menedżera obiektów**.

1. Użytkownik wybiera z graficznego paska zadań opcję dodania etykiety.
2. Obiekt reprezentujący aktualny rysunek zapamiętuje narzędzie i generuje zdarzenie oznaczające konieczność wyświetlenia pól do edycji etykiety.
3. **edytor_graf** sprawdza, czy są zaznaczone obiekty. Jeśli tak, są wyświetlone wszystkie punkty specjalne tych obiektów i wyróżniony jest domyślnie wybrany punkt specjalny pierwszego obiektu na liście zaznaczonych.
4. Jeśli użytkownik kliknie w pole rysunku, współrzędne kliknięcia są interpretowane następująco: dla wszystkich zaznaczonych obiektów wywoływana jest metoda sprawdzająca, czy w bliskim sąsiedztwie współrzędnych kliknięcia położony jest któryś z punktów specjalnych tego obiektu. Jeśli tak, to wybrany punkt specjalny staje się wyróżnionym, względem którego będzie pozycjonowana etykieta. Jeśli użytkownik kliknie daleko od punktów specjalnych zaznaczonych obiektów, etykieta będzie pozycjonowana bezwzględnie.
5. Użytkownik wpisuje tekst i naciska przycisk OK.
6. Obiekt **rysunek** tworzy obiekt klasy **Etykieta** na podstawie podanych parametrów. Obiekt jest dodawany do struktur rysunku.
7. Przy pozycjonowaniu względnym obiekt, do którego dowiązana jest etykieta, dodaje ją do swoich struktur. Generowane jest zdarzenie oznaczające konieczność przerysowania obrazu.

5. Dekompozycja logiczna systemu

5.1. Omówienie

Program GEM składa się z następujących komponentów:

Edytor graficzny to główna część programu. W nim tworzony jest rysunek. Komunikuje się z przeglądarką.

Edytor tekstowy to dodatkowa część programu. Pozwala na tworzenie rysunków w trybie tekstowym, bezpośrednio w kodzie METAPOST.

Przeglądarka wyświetla gotowy, skompilowany przez METAPOST rysunek. Komunikuje się z kompilatorem, mogąc mu zlecić kompilację danego rysunku.

Kompilator i konwerter to zespół zewnętrznych programów, które generują z kodu METAPOST gotowy rysunek w zadanym formacie wektorowym lub rastrowym.

Moduł sieciowy służy do przesyłania bieżących zmian na rysunku przez sieć i wyświetlania ich u drugiego użytkownika.

5.2. Najważniejsze komponenty

5.2.1. Edytor graficzny

Warstwa Logika systemowa.

Odpowiedzialność

1. Trzymanie informacji o edytowanym rysunku.
2. Komunikacja z użytkownikiem, umożliwianie edycji rysunku.
3. Komunikacja z przeglądarką.
4. Konwersja z wewnętrznego formatu edytora na kod METAPOST.
5. Komunikacja z kompilatorem.

Uwagi Edytor grafiki będzie dysponował podstawowym pakietem narzędzi do rysowania grafiki oraz pakietem narzędzi wspomagającym rysowanie grafów. W przyszłości prawdopodobnie zostaną dołączone inne pakiety, np. do konstrukcji geometrycznych czy wykresów.

5.2.2. Edytor tekstowy

Warstwa Logika systemowa.

Odpowiedzialność

1. Komunikacja z użytkownikiem, umożliwianie edycji kodu METAPOST.
2. Komunikacja z przeglądarką.
3. Komunikacja z kompilatorem.

5.2.3. Przeglądarka

Warstwa Logika systemowa.

Odpowiedzialność

1. Wyświetlanie rysunku.
2. Komunikacja z edytorem graficznym.
3. Komunikacja z kompilatorem.

5.2.4. Moduł sieciowy

Warstwa Warstwa sieciowa.

Odpowiedzialność

1. Komunikacja z edytorem graficznym.
2. Komunikacja z serwerem sieciowym.
3. Przesyłanie bieżących zmian na rysunku z użytkownika-serwera do użytkownika-klienta.

5.2.5. Kompilator i konwerter

6. Dekompozycja na procesy

Poniższy diagram prezentuje zależności między procesami, które będą działać w systemie.



Edytor to główny proces systemu odpowiedzialny za komunikację z użytkownikiem. W jego skład wchodzi sam edytor graficzny/tekstowy, przeglądarka etc. Ten proces działa przez cały czas.

Kompilator to zewnętrzny program kompilujący kod METAPOST. Działa tylko na polecenie edytora. Jest uruchamiany dosyć często — wymaga tego każde odświeżenie podglądu w przeglądarce.

Konwerter to zewnętrzny program konwertujący gotowy rysunek do różnych formatów wektorowych i rastrowych. Jest uruchamiany stosunkowo rzadko.

Serwer XML-RPC to serwer Apache XML-RPC działający na maszynie i odbierający zewnętrzne polecenia. Działa przez cały czas, kiedy włączona jest komunikacja sieciowa.

7. Instalacja systemu

Projekt GEM jest dostarczany w postaci źródeł z załączonymi skryptami do kompilacji i uruchamiania skompilowanego programu. Aby źródła dało się skompilować, potrzebny jest zainstalowany na komputerze kompilator języka Java razem z następującymi bibliotekami:

1. Apache XML-RPC,
2. Java Swing.

Do poprawnego działania programu użytkownik musi także mieć poprawnie zainstalowane i działające kompilatory języka \LaTeX i METAPOST. Wszystkie niezbędne narzędzia są darmowe i dostępne w sieci z poniższych adresów:

- <http://ws.apache.org/xmlrpc/>
- <http://www.tug.org/texlive>
- <http://java.sun.com/>

Po skompilowaniu zostanie uruchomiony moduł do wstępnej konfiguracji programu. Po poprawnym ustawieniu wszystkich wymaganych parametrów będzie można już normalnie używać programu, uruchamiając go odpowiednim skryptem. Wpisaną konfigurację można zmienić w dowolnym momencie, wybierając odpowiednią opcję z menu programu.

Aby działała sieciowa wersja programu, należy mieć dwie poprawnie zainstalowane kopie na komputerach połączonych siecią obsługującą protokół HTTP.

8. Implementacja systemu

8.1. Omówienie

Opis klas implementacyjnych przedstawia dość dokładnie projekt implementacji programu GEM, ale niektóre szczegóły zostały pominięte. Nie są podawane metody służące do modyfikacji atrybutów. Nie są wymienione standardowe klasy, z których Zespół będzie korzystał podczas implementacji systemu — w szczególności nie zostały opisane szczegóły budowy interfejsu systemu, złożonego w dużej części ze standardowych komponentów.

Zespół skoncentrował się na przedstawieniu najważniejszych klas programu, wraz z opisem istotnych metod i atrybutów. Ponadto w wielu punktach zostały podane uzasadnienia przyjęcia konkretnych rozwiązań, a także perspektywy dalszego rozwoju projektu w kontekście podejmowanych obecnie decyzji projektowych.

Podane poniżej nazwy klas i metod mogą jeszcze ulec drobnym zmianom. Przede wszystkim w programie nie będzie w nich polskich znaków, które w dokumencie są używane dla ułatwienia czytania. Ponadto niektóre nazwy mogą zostać skrócone.

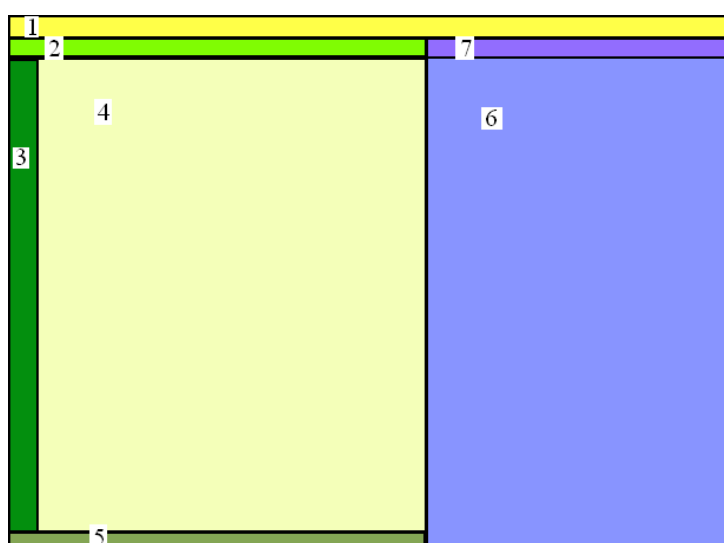
8.2. Warstwy

Poniżej zostały opisane poszczególne warstwy programu, wraz z omówieniem ich części składowych.

8.2.1. Interfejs

Interfejs programu GEM zostanie zaimplementowany z wykorzystaniem biblioteki Java SWING. Nie podajemy opisu klas odpowiedzialnych za wyświetlanie grafiki oraz przechwytywanie akcji użytkownika. Będą one wykorzystywały standardowe komponenty graficzne.

Poniżej przedstawiony został schematyczny wygląd interfejsu programu, podczas gdy uruchomiony jest edytor graficzny i widoczne jest pole przeglądarki. Przy uruchomionym edytorze tekstowym onko programu wygląda bardzo podobnie — różnice są takie, że edytor tekstowy ma jeden pasek narzędzi.



Opis poszczególnych części okna:

1. Pasek menu programu: zawiera standardowe opcje (włączanie/wyłączanie, minimalizacja itp.), obsługę plików oraz zmiany konfiguracji, a także opcje dostępne przez paski narzędzi.
2. Główny pasek narzędzi edytora graficznego: zawiera podstawowe opcje edytora: obsługę plików, przejście do kompilacji i tłumaczenia rysunku na kod języka METAPOST.
3. Boczny pasek narzędzi edytora graficznego: zawiera narzędzia graficzne; możliwość indywidualnej konfiguracji tego paska przez użytkownika jest funkcjonalnością opcjonalną programu. Pasek może być przemieszczany wewnątrz okna edytora.
4. Pole graficzne: na nim widoczny jest rysunek. Jeśli obszar przeglądarki zostanie zmniejszony, pole graficzne będzie zajmowało większy obszar i rysowanie może być w ten sposób wygodniejsze.
5. Pasek stanu edytora graficznego: zawiera kilka (bardzo niewiele) opcji edytora i możliwość ich szybkiego przełączania. W wersji podstawowej będzie to przede wszystkim możliwość włączenia/wyłączenia przyciągania do siatki. W edytorze tekstowym w pasku stanu znajdzie się opcja podświetlania składni.
6. Pole przeglądarki: w nim wyświetla się przeglądany rysunek w formacie wektorowym. Pole przeglądarki może być zwijane do prawej strony ekranu, aby zwiększyć obszar rysowania.

7. Menu przeglądarki: podstawowe opcje przeglądarki, takie jak skalowanie rysunku. Może być wyłączone, żeby nie zmniejszać pola podglądu — wówczas wszystkie konieczne opcje będą dostępne po kliknięciu prawym przyciskiem myszy w oknie przeglądarki. Ponadto podczas kompilacji lub tłumaczenia rysunku na kod METAPOST na dole okna pojawia się pasek z informacjami o przebiegu kompilacji.

8.2.2. Logika systemu — główna klasa nadzorująca działanie programu

Klasa	Model_GEM
Krótki opis	Jest główną klasą programu GEM, łączącą warstwy interfejsu i logiki systemowej. Zna obiekty odpowiedzialne za działanie poszczególnych uruchomionych części działającego programu. Obiekt głównej klasy interfejsu programu zna obiekt klasy <i>Model_GEM</i> i przekazuje do niego lub do obiektów będących jego atrybutami akcje użytkownika. Obiekt klasy <i>Model_GEM</i> służy w programie głównie do wykonania inicjalizacji i zamknięcia całego programu oraz do umożliwienia w łatwy sposób komunikacji między modułami: główne części programu znają nadzorujący je obiekt tej klasy i przez jego atrybuty mogą się odwoływać do innych części programu. Takie rozwiązanie zostało wybrane, ponieważ różne części programu (edytory, klasy sieciowe) mogą wielokrotnie powstawać i być likwidowane podczas działania programu, a wtedy przydaje się obiekt, który istnieje przez cały czas działania programu, nadzoruje włączanie i wyłączanie poszczególnych części oraz istotnie ułatwia aktualizację połączeń pomiędzy modułami programu.
Atrybuty	<p>Przede wszystkim istnieją atrybuty odpowiadające obiektom klas odpowiedzialnych za działanie głównych części programu:</p> <ul style="list-style-type: none"> • edytor — obiekt jednej z podklas klasy <i>Edytor</i> (czyli <i>Edytor_Graficzny</i> lub <i>Edytor_Tekstowy</i>), w zależności od tego, który aktualnie jest otwarty (może być tylko jeden); • kompilator — obiekt klasy <i>Komp_Konw</i>, odpowiedzialny za kompilację kodu METAPOST oraz konwersję formatu eps na inne formaty; • graftomp — obiekt klasy <i>Grafika2MP</i>, odpowiedzialny za zamianę wewnętrznego formatu plików edytora graficznego na język METAPOST; • przeglądarka — obiekt klasy <i>Przeglądarka</i>, odpowiedzialny za działanie szybkiego podglądu rysunku; • serwer — obiekt klasy <i>Serwer</i>, jeśli aktualnie jest potrzebny; • klient — obiekt klasy <i>Klient</i>, jeśli aktualnie jest potrzebny. <p>Ponadto istnieje</p> <ul style="list-style-type: none"> • konfiguracja — obiekt klasy <i>Konfig</i>, przechowujący dane ustawień wykonywanych podczas uruchamiania programu; te ustawienia są wczytywane z pliku konfiguracyjnego podczas uruchamiania programu i zapisywane do niego przy każdej zmianie konfiguracji dokonywanej przez użytkownika.

Metody	<ul style="list-style-type: none"> • init() — otwiera program: wczytuje konfigurację z pliku tekstowego, tworzy obiekty odpowiedzialne za poszczególne części działającego programu zgodnie z wczytaną konfiguracją (np. użytkownik może ustawić, czy domyślnie ma się otwierać edytor tekstowy, czy graficzny). <p>Ponadto istnieje kilka metod włączających poszczególne części programu, np. uruchamiające edytor tekstowy lub serwer, ale nie ma potrzeby opisywać ich szczegółowo.</p>
---------------	---

Klasa	Konfig
Krótki opis	Jest klasą służącą do odczytywania i zapisu plików konfiguracyjnych różnych części programu.
Atrybuty	<ul style="list-style-type: none"> • ściezka — ścieżka do pliku konfiguracyjnego. • info — słownik parametrów konfiguracyjnych, w którym trzymane są wartości przeczytane z pliku konfiguracyjnego lub wstawiane są domyślne, jeśli w pliku nie wszystko zostało podane.
Metody	<ul style="list-style-type: none"> • init(ściezka) — próbuje czytać dane z pliku o podanej ścieżce; jeśli nie może go znaleźć lub format jest nieprawidłowy, ustawia domyślne wartości parametrów; zapamiętuje również ścieżkę pliku konfiguracyjnego; • zapisz() — zapisuje do pliku o zapamiętanej ścieżce obecną konfigurację.

Parametry konfiguracyjne obiektu klasy *Model_GEM*:

- **rozmiar_okna** — wymiary okna programu;
- **edytor** — czy przy włączaniu programu ma być uruchamiany edytor tekstowy, czy graficzny;
- **rozmiar_przeglądarki** — jak dużą część okna ma zajmować przeglądarka efektów;
- **katalog** — domyślna ścieżka do katalogu z projektami.

8.2.3. Logika systemu — edytor graficzny

Klasa	Edytor
Krótki opis	Jest nadklasą klas <i>Edytor_Graficzny</i> i <i>Edytor_Tekstowy</i> (opisanej w punkcie 8.2.5) — występuje, ponieważ oba edytory mają wspólne podstawowe atrybuty i część metod. Poniższe atrybuty są opisane również przy podklasach klasy <i>Edytor</i> , dokładniej, z wyszczególnieniem pewnych różnic między edytorem tekstowym i graficznym. Tutaj jest opisane tylko ogólne przeznaczenie.

Atrybuty	<ul style="list-style-type: none"> • konfiguracja — obiekt klasy <i>Konfig</i> przechowujący dane konfiguracyjne; • obrazy — lista otwartych rysunków: obiektów klasy <i>Rysunek</i> w przypadku edytora graficznego lub obiektów zawierających informacje o plikach tekstowych w przypadku edytora tekstowego); jest to przyszłościowe ujęcie problemu — w pierwszej wersji program ma zapewniać możliwość edycji jednego rysunku, otwieranie wielu plików naraz jest funkcjonalnością opcjonalną; • aktualny — aktualnie edytowany rysunek.
Metody	<ul style="list-style-type: none"> • init(ścieżka_pliku) — inicjuje pracę edytora: wczytuje dane konfiguracyjne; tworzy nowy obiekt klasy <i>Rysunek</i> lub <i>Plik_Tekst</i>, zależnie od edytora lub, jeśli została podana ścieżka pliku, a nie pusty napis, to wczytuje elementy rysunku lub tekst z pliku; • close() — zamyka edytor pytając, czy zapisać plik (jeśli jakiś jest edytowany); • podgląd() — wywołuje jedną z metod generowania podglądu w przeglądarce i dostarcza odpowiednie dane, w zależności od rodzaju edytora; • otwórz_plik(ścieżka_pliku) — otwiera plik do edycji i wczytuje jego zawartość; • zapisz_plik(ścieżka_pliku) — zapisuje dane aktualnie edytowanego rysunku do pliku o podanej ścieżce; • do_kompilacji — przekazuje aktualnie edytowany plik modułowi kompilatora do procesu kompilacji, w edytorze graficznym po przetłumaczeniu na język METAPOST. <p>Powyższe metody są definiowane w podklasach.</p>

Klasa	Edytor >> Edytor_Graficzny
Krótki opis	Stanowi podklasę klasy <i>Edytor</i> . Jest klasą, której obiekt nadzoruje pracę edytora graficznego w programie GEM. Przechowuje wszystkie informacje o edytowanych rysunkach oraz dane konfiguracyjne edytora graficznego. Podczas działania programu istnieje co najwyżej jeden obiekt tej klasy — jeśli użytkownik wyłączy edytor graficzny (np. uruchamiając tekstowy), to nie ma potrzeby, żeby istniał.
Atrybuty	<p>Dokładniejszy opis parametrów zdefiniowanych w nadklasie:</p> <ul style="list-style-type: none"> • konfiguracja — obiekt klasy <i>Konfig</i>, zawierający informacje o startowych ustawieniach edytora graficznego; • obrazy — lista obiektów klasy <i>Rysunek</i>, przechowujących i modyfikujących zgodnie z życzeniami użytkownika wszystkie dane dotyczące rysunków (w początkowej wersji będzie możliwe utworzenie tylko jednego rysunku); • aktualny — aktualnie edytowany rysunek (obiekt klasy <i>Rysunek</i>);
Metody	<p>Przeddefiniowuje metody nadklasy. Ponadto istnieje metoda:</p> <ul style="list-style-type: none"> • tłumacz_do_MP(ścieżka_pliku_wynikowego) — korzystając z modułu tłumaczącego plik graficzny na kod języka METAPOST tworzy plik tekstowy tego języka z opisem rysunku.

Parametry konfiguracyjne obiektu klasy *Edytor_Graf* — ogólne:

- **kolor_tła** — kolor tła pola do rysowania;
- **czcionka** — domyślnie używana w polu rysunkowym czcionka;
- **rozmiar_czcionki** — domyślna wielkość czcionki;
- **kolor_czcionki** — domyślny kolor czcionki;
- **paski** — informacje o włączanych przy uruchamianiu programu paskach narzędzi (różnicowanie pasków narzędzi).

Dane konfiguracyjne obiektu klasy *Edytor_Graf* dotyczące etykiet i połączeń grafowych:

- **pozycja_etykiety** — domyślny kierunek przesunięcia etykiety dowiązanej do obiektu względem punktu specjalnego;
- **tło_etykiety** — domyślny kolor tła etykiety;
- **punkt_specjalny** — dane precyzujące, który punkt specjalny obiektu graficznego ma być wybierany domyślnie;
- **kształt_połączenia** — domyślnie wybierany przebieg połączenia grafowego.

Klasa	Rysunek
Krótki opis	Zawiera wszystkie dane potrzebne do obsługi pojedynczego rysunku (otwartego do edycji pliku graficznego). Implementuje ogół operacji, które użytkownik może wykonywać na rysunku.

Atrybuty	<ul style="list-style-type: none"> • ścieżka — ścieżka do pliku, z którego rysunek został wczytany (jeśli nie jest to nowy rysunek); • skala — parametr określający skalowanie wyświetlanego rysunku; • centrum — położenie na rysunku punktu, który jest wyświetlany w środku pola graficznego; • wyświetlaj_jak_przeglądarka — informacja, czy zmiany wielkości i położenia widocznego na ekranie fragmentu rysunku mają być dostosowywane do przeglądarki. <p>Przechowywanie obiektów graficznych:</p> <ul style="list-style-type: none"> • obiekty — lista wszystkich obiektów na rysunku; • zaznaczone — lista obiektów zaznaczonych na rysunku (tych, na których odbywa się jakaś operacja); • zero — obiekt klasy <i>Środek_układu</i>, reprezentujący początek układu współrzędnych rysunku — względem niego pozycjonowane są obiekty, nie jest on wyświetlany na ekranie (to na razie tylko udogodnienie implementacyjne, ale może podczas dalszego rozwoju programu pojawiać się inne zastosowania tego punktu); • stosy — obiekt klasy <i>Stosy</i>, która implementuje cofanie wykonanych operacji: składa się ze stosu operacji oraz stosu wycofań. • menedżer — obiekt klasy <i>Menedżer</i>, indywidualny dla rysunku menedżer obiektów; mógłby być jeden dla całego edytora, ale ponieważ program ma umożliwiać szybkie przełączanie pomiędzy kilkoma otwartymi rysunkami, to inicjalizowanie menedżera za każdym razem byłoby kłopotliwe. • siatka — obiekt klasy <i>Siatka</i> przechowujący dane siatki używanej aktualnie na rysunku. <p>Atrybuty potrzebne do reagowania na akcje użytkownika:</p> <ul style="list-style-type: none"> • wsp — ostatnio zapamiętana para współrzędnych kursora, zazwyczaj współrzędne kliknięcia lub (podczas przybliżania trasy kursora krzywą) pozycja kursora sprawdzana co określoną jednostkę czasu); • narzędzie — identyfikator aktualnie wybranego narzędzia (istnieje również możliwość umieszczenia znacznika wybranego narzędzia bezpośrednio w edytorze graficznym, ale Zespół uznał, że lepiej wybierać narzędzie indywidualnie dla każdego rysunku;) Będą również potrzebne pola do przechowywania wiadomości na temat tworzonych w odpowiedzi na akcje użytkownika obiektów graficznych — często trzeba czekać na podanie przez użytkownika kilku kolejnych danych, zanim można stworzyć nowy obiekt i dodać go do rysunku. Powstanie pole: • tymczasowe — pozwala trzymać tworzony obiekt, zanim będzie można go włączyć do struktur rysunku; <p>ale być może w tym celu zostaną dodane również inne pola.</p>
----------	--

Metody	<p>Operacje ogólne:</p> <ul style="list-style-type: none"> • init() — ustawia początkowe wartości swoich atrybutów dla pustego rysunku; • zapisz_do_pliku(nazwa_pliku) — zapisuje do pliku aktualny stan rysunku (po prostu zrzucając do niego cały obiekt); • zmień_siatkę(rodzaj) — zmienia rodzaj siatki na inny, tworzy nowy obiekt podklasy klasy <i>Siatka</i> odpowiadającej podanemu rodzajowi, z domyślnymi wymiarami pól; zmiany wymiarów siatki będą się odbywały poprzez wywołanie metod klasy <i>Siatka</i>; • ustal_wyświetlanie(skala, centrum) — ustawia atrybuty <i>skala</i> i <i>centrum</i> na odpowiednie wartości (służy między innymi do ujednolicania wyświetlanego obszaru w polu graficznym edytora i w przeglądarce); • interpretuj_współrzędne — wykonywana, gdy edytor dostanie współrzędne kursora jako reakcję na jakieś zdarzenie, np. użytkownik kliknie wewnątrz pola graficznego; wówczas, w zależności od aktualnie wybranego narzędzia, zostanie wykonana odpowiednia akcja. Wewnątrz tej metody będą odbywały się główne operacje graficzne, takie jak rysowanie nowych figur i ich modyfikacje. Wobec tego ta metoda będzie korzystała z wielu (zazwyczaj niewielkich) metod pomocniczych, które jednak są na tyle szczegółowe, że Zespół postanowił nie opisywać ich w tym miejscu. <p>Główne metody służące do opisu zmian wprowadzanych na rysunku to:</p> <ul style="list-style-type: none"> • usuń() — usuwa zaznaczone obiekty z rysunku; • dodaj(obiekt_graficzny) — dodaje do struktur rysunku dany obiekt graficzny (zazwyczaj nowo utworzony w wyniku interpretacji akcji użytkownika); • wyszukaj(współrzędne_punktu) — wywołując metodę <i>czy_należy(współrzędne_punktu)</i> w obiektach z listy, wyszukuje obiekt, do którego należy punkt o podanych współrzędnych (zazwyczaj odczytanych współrzędnych kursora; chodzi o punkt dany jako współrzędne w polu graficznym, a nie obiekt klasy <i>Punkt</i>); • zaznacz(obiekt_graficzny) — zmienia atrybut obiektu określający zaznaczenie i dodaje go do listy obiektów zaznaczonych. <p>Zespół nie podaje opisu wszystkich metod, wymienione są wyłącznie najważniejsze i najbardziej ogólne — wiele reakcji na akcje użytkownika będzie się odbywało po prostu poprzez wywołanie metody pewnego obiektu graficznego (lub grupy obiektów) z odpowiednimi parametrami. Część opisanych metod będzie również modyfikowała menedżer obiektów.</p>
---------------	--

Klasa	Menedżer
Krótki opis	Klasa implementująca menedżer obiektów. Zajmuje się formatowaniem danych do wyświetlenia w oknie menedżera, wyszukiwaniem obiektów graficznych wybieranych przez użytkownika poprzez okno menedżera oraz przekazywaniem edytorowi graficznemu poleceń użytkownika po wyszukaniu obiektów potrzebnych do ich wykonania.

Atrybuty	<ul style="list-style-type: none"> • obiekty — struktura danych zawierająca obiekty graficzne rysunku oraz dane potrzebne menedżerowi do ich wyświetlenia, czyli przede wszystkim identyfikator obiektu (przypisana mu nazwa wyświetlana w oknie) oraz informacja, czy obiekt jest zaznaczony; tę strukturę Zespół chciałby zaimplementować jako słownik indeksowany kolejnością wyświetlania obiektów graficznych — będzie to bardzo przydatne w przyszłości, jeśli Zespół zdecyduje się rozwijać program;
Metody	<ul style="list-style-type: none"> • init() — inicjalizuje menedżer obiektów na podstawie danych rysunku, do którego jest przypisany; • zaznacz(identyfikator) — wyszukuje obiekt o danym identyfikatorze i wywołuje metodę zaznaczania w obiekcie rysunku, do którego jest przypisany; • aktualizuj() — odświeża informacje o rysunku. <p>Więcej opcji menedżer będzie udostępniał podczas dalszego rozwoju programu.</p>

Klasa	Stosy
Krótki opis	Klasa implementująca jednoczesną obsługę stosu czynności wykonanych i stosu wycofań.
Atrybuty	<ul style="list-style-type: none"> • stos_wykonań — stos (ograniczonej wielkości), na którym są odkładane po kolei czynności wykonane na danym rysunku; • stos_wycofań — stos, na którym podczas wycofywania ostatnio wprowadzonych zmian odkładane są elementy zdjęte ze stosu wykonań.
Metody	<ul style="list-style-type: none"> • cofnij() — implementuje operację cofania wykonanej czynności; • powtórz() — implementuje operację odwrotną do poprzedniej operacji.

8.2.4. Logika systemu — grafika

Klasa	Środek_układu
Krótki opis	Nie jest podklasą klasy <i>Obiekt_Graficzny</i> , ponieważ ma istotnie inne właściwości i mniejsze możliwości niż obiekty graficzne. Na każdym otwartym przez edytor graficzny rysunku istnieje dokładnie jeden obiekt tej klasy. Istnieje on po to, aby określać względem niego położenie obiektów, które nie mają być pozycjonowane względem żadnego innego obiektu rysunku. Nie jest wyświetlany; użytkownik nie wie o jego istnieniu.
Atrybuty	<ul style="list-style-type: none"> • współrzędne — współrzędne na ekranie względem jego środka.
Metody	Ma wyłącznie standardowe metody obsługujące modyfikacje atrybutów.

Klasa	Obiekt_graficzny
Krótki opis	Jest nadklasą wszystkich obiektów graficznych, opisanych poniżej.

Atrybuty	<ul style="list-style-type: none"> • kolejność_wyświetlania — w przyszłości zespół chciałby dodać do programu możliwość kontroli kolejności wyświetlania obiektów; ten atrybut będzie aktualizowany podczas działania programu, ale w pierwszej wersji prawdopodobnie nie zostanie udostępniona możliwość jego zmiany przez użytkownika; • zaznaczony — pamięta, czy obiekt jest aktualnie zaznaczony, czy nie; ta informacja wystarczałaby do obsługi zaznaczania obiektów, ale obiekt klasy <i>Rysunek</i> trzyma również listę zaznaczonych obiektów, aby nie trzeba było zbyt często przeglądać listy wszystkich obiektów graficznych; • położenie_względem — pamięta obiekt, względem którego należy obliczać jego położenie; • współrzędne — w prawie wszystkich podklasach ten atrybut oznacza wektor przesunięcia (w umownych jednostkach programu) względem obiektu, według którego dany obiekt jest pozycjonowany (dość zaawansowana opcja, w pierwszej wersji przeznaczona głównie dla etykiet, inne obiekty będą pozycjonowane względem specjalnego obiektu — środka układu współrzędnych; ta opcja będzie w przyszłości rozwijana); • kolor — kolor rysunku obiektu; • grubość — grubość linii rysunku obiektu; • etykiety — lista etykiet dowiązanych do tego obiektu; niektóre klasy definiują ją tak, aby nie pozwalać dodać do siebie etykiety; • widoczność — czy obiekt jest widoczny na rysunku, czy nie (zazwyczaj jest).
----------	--

Metody	<ul style="list-style-type: none"> • czy_należy(współrzędne_punktu) — sprawdza, czy dany punkt (zazwyczaj współrzędne kliknięcia w pole rysunkowe) należy do danego obiektu graficznego (czyli czy leży odpowiednio blisko tego obiektu); tę metodę przeddefiniowuje się we wszystkich podklasach; • czy_specjalny(współrzędne_punktu) — sprawdza, czy dany punkt jest punktem specjalnym danego obiektu graficznego; tę metodę przeddefiniowuje się we wszystkich podklasach; • narysuj(płótno) — przelicza przechowywane dane na dane stanowiące podstawę do narysowania obiektu (np. z końców i wektorów sterujących wylicza parametryzację krzywej) oraz rysuje swój obraz na podanym płótnie; • wyświetl_punkty_specjalne — zaznacza na rysunku punkty specjalne danego obiektu. W pierwotnej wersji programu, w której większość obiektów graficznych to krzywe (ze względu na planowane możliwości edytora, również prostokąty i elipsy będą rysowane jako krzywe, z możliwością dokonywania dowolnych zmian po narysowaniu), punkty specjalne będą powiązane z węzłami i punktami parametryzowanymi na krzywej, a także z punktami niewidocznymi w pewnych szczególnych przypadkach. Ta metoda jest przeddefiniowywana przez wszystkie podklasy. <p>Oprócz tego istnieją standardowe metody pozwalające między innymi na modyfikację koloru, grubości linii, dodanie, wyszukanie lub usunięcie etykiety z listy.</p>
---------------	--

Klasa	Obiekt_graficzny >> Punkt
Krótki opis	Abstrakcyjna nadklasa opisanych poniżej klas <i>Węzeł</i> oraz <i>Punkt_parametryzowany</i>
Atrybuty	Nie potrzebuje atrybutów innych, niż w nadklasie.
Metody	Przeddefiniowuje metodę <i>czy_należy(współrzędne_punktu)</i> ; nie jest ona już zmieniana w podklasach.

Klasa	Obiekt_graficzny >> Punkt >> Węzeł
Krótki opis	Jest klasą, której obiekty tworzą węzły krzywych, czyli końce łuków krzywych. Klasa <i>Węzeł</i> może mieć zdefiniowane podklasy opisujące różne typy węzłów w krzywych Béziera (gładkie i niegładkie). Nie jest to jednak konieczne w pierwotnej wersji programu i nie zostało tutaj dokładnie opisane.
Atrybuty	<ul style="list-style-type: none"> • lista_łuków — lista łuków krzywych, których wierzchołkiem jest dany węzeł. Jeśli nawet w pierwotnej wersji programu ten atrybut będzie miał niewielkie zastosowanie, to przyda się do rozwijania opcjonalnych funkcjonalności — operacji na krzywych, które mają szansę powstać do czerwca.
Metody	Przeddefiniowuje metodę rysującą.

Klasa	Obiekt_graficzny >> Punkt >> Punkt_parametryzowany
Krótki opis	Klasa opisująca punkty, które należą do łuków krzywych i ich położenie na łuku jest określone poprzez informację, ile procent długości łuku ma się znajdować przed tym punktem.
Atrybuty	<ul style="list-style-type: none"> • przesunięcie — procent długości łuku krzywej określający miejsce, gdzie ten punkt się znajduje; • łuk — łuk krzywej, do którego dany punkt należy. Byłoby dobrze, żeby taki punkt mógł należeć do kilku łuków naraz — wtedy jego przesunięcie na jednym z łuków modyfikowałoby pozostałe łuki. Tego jednak Zespół na razie nie zamierza zawrzeć w programie.
Metody	Przeddefiniowuje metodę rysującą, definiując w tym celu bardzo ważną metodę pomocniczą wyliczającą faktyczne (nie względne) położenie punktu.

Klasa	Obiekt_graficzny >> Punkt >> Środek_cieźkości
Krótki opis	Klasa, która jest potrzebna do szczególnych rozwiązań dotyczących głównie etykiet. Konkretnie, byłoby dobrze zapewnić możliwość umieszczenia etykiety lub doprowadzenia połączenia grafowego np. do środka symetrii prostokąta lub środka okręgu, a ogólnie — do środka ciężkości węzłów krzywej (prostokąty i elipsy będą tworzone jako krzywe). Wobec tego z krzywą można związać punkt klasy <i>Środek_cieźkości</i> , do którego będzie można dowiązać etykiety i połączenia.
Atrybuty	• krzywa — krzywa, z którą jest związany dany punkt niewidoczny.
Metody	Przeddefiniowuje metody z nadklasy.

Klasa	Obiekt_graficzny >> Łuk_krzywej
Krótki opis	Jest to klasa, której obiekty modelują fragmenty krzywych, łączące dwa sąsiednie węzły.
Atrybuty	<ul style="list-style-type: none"> • początek — węzeł będący początkiem łuku; • koniec — węzeł będący końcem łuku (kolejność jest istotna dla parametryzowania punktów); • wektor_pocz — wektor sterujący, określający przebieg danego łuku (krzywej Béziera) w pobliżu początku; • wektor_kon — wektor sterujący, określający przebieg danego łuku (krzywej Béziera) w pobliżu końca; • punkty_parametryzowane — lista punktów parametryzowanych danego łuku krzywej.
Metody	Przeddefiniowuje metody z nadklasy; szczególnie metoda rysująca jest dość techniczna i może być skomplikowana.

Klasa	Obiekt_graficzny >> Krzywa
Krótki opis	Klasa, której obiekty opisują krzywe, złożone z łuków.

Atrybuty	<ul style="list-style-type: none"> • łuki — lista łuków danej krzywej; • węzły — lista wszystkich węzłów krzywej; • środek — środek ciężkości krzywej.
Metody	Przeddefiniowuje metody z nadklasy, wykorzystując do tego głównie metody (rysującą i sprawdzającą należenie punktu o danych współrzędnych) klasy <i>Łuk_krzywej</i> .

Klasa	Obiekt_graficzny >> Etykieta
Krótki opis	Klasa opisująca etykiety dodawane do rysunku.
Atrybuty	<ul style="list-style-type: none"> • tekst — treść etykiety; • gdzie_tekst — przesunięcie tekstu w stosunku do końca wektora przesunięcia etykiety względem obiektu, do którego jest dowiązana; • obiekt_bazowy — obiekt, do którego dana etykieta jest dowiązana; • ramka — ewentualnie istniejąca ramka otaczająca tekst.
Metody	Przeddefiniowuje metody z nadklasy.

Klasa	Obiekt_graficzny >> Ramka
Krótki opis	Klasa opisująca ramkę, która może otaczać tekst etykiety. Być może będzie miała kilka podklas, w zależności od zamierzonego wyglądu (np. czy rogi mają być zaokrąglone, czy nie). Nie zostało to tutaj dokładnie opisane, ponieważ jest to szczegół nie mający dużego znaczenia dla całości programu. Poza tym dobra implementacja tej funkcjonalności może się okazać dość trudna, ponieważ automatyczne narysowanie takiej ramki wymaga uzyskania od kompilatora języka METAPOST informacji o wymiarach tekstu. W pierwotnej wersji można jednak pozwolić użytkownikowi na podanie własnych wymiarów ramki i dopasowanie ich do tekstu etykiety. Ogólnie, jak wynika z oglądanych przez Zespół prac naukowych, rysowanie ramek wokół etykiet nie jest ważną funkcjonalnością programu.
Atrybuty	<ul style="list-style-type: none"> • wymiary_tekstu — szerokość i wysokość tekstu znajdującego się w etykiecie; • tło — kolor tła etykiety wewnątrz ramki.
Metody	Przeddefiniowuje metody z nadklasy.

Klasa	Obiekt_graficzny >> Krawędź
Krótki opis	Klasa, której obiekty są połączeniami grafowymi, prowadzącymi od jednego punktu do drugiego, w ustalonej kolejności, według określonego schematu przebiegu.

Atrybuty	<ul style="list-style-type: none"> • skąd — para: obiekt graficzny i jego punkt specjalny, z którego wychodzi połączenie; • dokąd — para: obiekt graficzny i jego punkt specjalny, do którego biegnie połączenie; • przebieg — obiekt klasy <i>Kształt_połączenia</i> lub <i>Łuk krzywej</i>; program ma udostępniać połączenia mogące być modyfikowane jak krzywe Béziera.
Metody	Przeddefiniowuje metody z nadklasy.

Klasa	Obiekt_graficzny >> Kształt_połączenia
Krótki opis	Klasa, której obiekty przechowują parametry połączeń grafowych (krawędzi) korzystających ze standardowych szablonów. Ma kilka podklas odpowiadających różnym szablonom połączeń, ale nie opisujemy ich tu dokładnie — różnią się one głównie sposobem definiowania metody rysującej.
Atrybuty	<p>Parametry mogą się różnić dla różnych podklas. W początkowej wersji programu będą przynajmniej dwie podklasy, tworzące połączenia biegnące równolegle do osi układu współrzędnych, o co najwyżej dwóch punktach zgięcia, różniące się tym, czy będą gładkie w punktach zgięcia. Wówczas potrzebne są następujące parametry:</p> <ul style="list-style-type: none"> • odc[i] — $i \in \{1, 2, 3\}$; są to pary złożone z identyfikatora kierunku, w którym idzie odcinek połączenia (N,E,S,W) oraz długości kolejnego odcinka; od funkcji rysującej zależy, czy zaokrągli połączenia kolejnych odcinków.
Metody	Przeddefiniowuje metody z nadklasy.

Klasa	Obiekt_graficzny >> Siatka
Krótki opis	Nadklasa klas odpowiadających wyspecjalizowanym siatkom — trójkątnym, prostokątnym lub jeszcze innym, które być może powstaną podczas rozwoju programu. Nie są tu opisane dokładnie, ich opis nie zawiera istotnych dla architektury systemu szczegółów.
Atrybuty	<ul style="list-style-type: none"> • wierzchołki — lista wektorów określających przesunięcie kolejnych wierzchołków podstawowego obszaru siatki względem środka układu współrzędnych.
Metody	Przeddefiniowuje metody z nadklasy. W szczególności metoda <i>czy_należy(współrzędne_punktu)</i> jest bardzo istotna dla opcji przyciągania do siatki.

8.2.5. Logika systemu — edytor tekstowy

Klasa	Edytor >> Edytor_Tekstowy
--------------	--

Krótki opis	Stanowi podklasę klasy <i>Edytor</i> . Obiekt tej klasy nadzoruje pracę edytora tekstowego. Przechowuje potrzebne informacje o otwartych do edycji plikach tekstowych oraz dane konfiguracyjne specyficzne dla edytora tekstowego, które nie są trzymane razem z konfiguracją całego programu.
Atrybuty	Dokładniejszy opis parametrów zdefiniowanych w nadklasie: <ul style="list-style-type: none"> • pliki — lista obiektów klasy <i>Plik_Tekst</i>, obsługujących pracę nad tekstem (kodem języka METAPOST); • aktualny — aktualnie edytowany plik tekstowy (obiekt klasy <i>Plik_Tekst</i>); • konfiguracja — obiekt klasy <i>Konfig</i>, zawierający informacje o startowych ustawieniach edytora tekstowego, np. używanej czcionce i kolorze tła.
Metody	Przeddefiniowuje metody z nadklasy. Z charakterystycznych dla naszego programu metod edytora tekstowego istnieją: <ul style="list-style-type: none"> • analizuj() — analizuje podany tekst, wyszukując słów kluczowych języka METAPOST i tworzy dane o tekście pozwalające na wypisanie go z zaznaczeniem składni.

Klasa	Plik_Tekst
Krótki opis	Klasa opisująca otwarty przez edytora tekstowy plik z kodem języka METAPOST.
Atrybuty	<ul style="list-style-type: none"> • ścieżka — ścieżka do pliku, z którego tekst został wczytany (jeśli nie jest to nowy plik); • treść — aktualny tekst edytowanego pliku; • stosy — obiekt klasy <i>Stosy</i>, która implementuje cofanie wykonanych operacji: składa się ze stosu operacji oraz stosu wycofań.
Metody	<ul style="list-style-type: none"> • init() — ustawia początkowe wartości swoich atrybutów dla pustego pliku tekstowego; • zapisz_do_pliku(nazwa_pliku) — zapisuje do pliku aktualny tekst.

Parametry konfiguracyjne obiektu klasy *Edytor_Tekst*:

- **czcionka** — domyślnie używana czcionka;
- **rozmiar_czcionki** — domyślna wielkość czcionki;
- **kolor_czcionki** — domyślny kolor czcionki;
- **składnia** — czy składnia języka METAPOST ma być rozpoznawana i podkreślana.

8.2.6. Logika systemu — przeglądarka efektów

Klasa	Przeglądarka
Krótki opis	Główna klasa zajmująca się obsługą szybkiego wyświetlania efektów pracy. Na polecenie edytora, korzystając z przekazanych parametrów rysunku, który należy wyświetlić, ewentualnie zleca jego tłumaczenie na kod METAPOST i kompilację przeznaczonym do tego modułem, a następnie wyświetla rysunek w formacie wektorowym.

Atrybuty	<ul style="list-style-type: none"> • konfiguracja — obiekt klasy <i>Konfig</i>, zawierający dane konfiguracyjne przeglądarki; • aktualny — wyświetlany w danej chwili plik; może być potrzebny do odświeżania rysunku; • skala — parametr określający skalowanie wyświetlanego rysunku; • centrum — położenie na rysunku punktu, który jest wyświetlany w środku pola graficznego przeglądarki; • wyświetlaj_jak_edytor — informacja, czy zmiany wielkości i położenia widocznego na ekranie fragmentu rysunku mają być dostosowywane do edytora graficznego.
Metody	<ul style="list-style-type: none"> • init() — ustawia początkowe wartości parametrów po utworzeniu; • ustal_wyświetlanie(skala, centrum) — ustawia atrybuty <i>skala</i> i <i>centrum</i> na odpowiednie wartości (służy między innymi do ujednolicania wyświetlanego obszaru w polu graficznym edytora i w przeglądarce); • wyświetl(gdzie) — rysuje w podanym miejscu odpowiedni obszar rysunku, według pamiętanych wartości <i>centrum</i> i <i>skala</i>; • podgląd_mp(ścieżka_pliku) — wyświetla obraz dany jako obiekt klasy <i>Rysunek</i>, po kompilacji podanego pliku; • podgląd_wekt(ścieżka_pliku) — wyświetla dany plik w formacie wektorowym, jeśli format jest odpowiedni.

Parametry konfiguracyjne obiektu klasy *Przeglądarka*:

— **tmp** — ścieżka do katalogu, w którym przeglądarka może tworzyć pliki tymczasowe.

8.2.7. Logika systemu — kompilator i konwersje formatów

Klasa	Komp_Konw
Krótki opis	Jest to klasa obsługująca kompilację plików tekstowych z kodem METAPOST do plików graficznych w formacie wektorowym powstającym w czasie kompilacji języka METAPOST, bardzo zbliżonym do eps. Ponadto udostępnia również metody konwersji pewnych formatów wektorowych na inne. Implementacja tych metod będzie polegała na wywołaniu zewnętrznych programów z odpowiednimi parametrami. Zespół uznał, że jedna klasa może kontrolować te czynności.
Atrybuty	<ul style="list-style-type: none"> • pokaż_logi — czy logi kompilacji mają być wyświetlane; • konfiguracja — obiekt klasy <i>Konfig</i>, konfiguracja kompilatora i konwertera, zawierająca głównie ścieżki do wywoływanych zewnętrznych programów.
Metody	<ul style="list-style-type: none"> • kompiluj_mp(ścieżka_pliku) — wywołuje kompilator języka METAPOST, próbując znaleźć go poprzez ścieżkę zapisaną w konfiguracji, dla danego pliku; • konwertuj_do(nazwa, ścieżka_pliku) — wywołuje zewnętrzny program konwertujący podany plik odpowiedniego formatu; • kompiluj_do(nazwa, ścieżka_pliku) — kompiluje podany plik i od razu konwertuje go do podanego formatu.

Parametry konfiguracyjne obiektu klasy *Komp_Konw*:

- **tmp** — ścieżka do katalogu, w którym można tworzyć pliki tymczasowe kompilacji i konwersji;
- **ścieżka_komp** — ścieżka do kompilatora języka METAPOST; jeśli podczas próby kompilacji okaże się, że jest nieaktualna, trzeba poprosić użytkownika o podanie nowej wartości;
- **konwertery** — lista (lub słownik) ścieżek do plików wykonywalnych konwerterów.

8.2.8. Logika systemu — tłumacz rysunku na język METAPOST

Klasa	Grafika2MP
Krótki opis	Klasa pełniąca funkcję tłumacza obiektowego formatu rysunku na kod języka METAPOST. Ma prosty schemat, ale metoda tłumacząca może wymagać wielu metod pomocniczych, które nie zostaną tu wymienione.
Atrybuty	<ul style="list-style-type: none"> • konfiguracja — obiekt klasy <i>Konfig</i>, przechowujący dane konfiguracyjne, na razie nieliczne.
Metody	<ul style="list-style-type: none"> • tłumacz(rysunek) — dostaje rysunek w postaci obiektowej (czyli obiekt klasy <i>Rysunek</i>) i przeprowadza proces konstrukcji kodu METAPOST odpowiadającego temu rysunkowi.

Parametry konfiguracyjne obiektu klasy *Grafika2MP*:

- **tmp** — ścieżka do katalogu, w którym można tworzyć pliki tymczasowe z wynikiem tłumaczenia, zazwyczaj przetwarzane później przez inne moduły programu.

8.2.9. Warstwa sieciowa — aplikacje klienta i serwer

Klasa	Serwer
Krótki opis	Ta klasa służy do tworzenia połączenia internetowego i odbierania oraz interpretacji informacji o zmianach dokonywanych na rysunku przez użytkownika po drugiej stronie połączenia.
Atrybuty	Atrybutami są dane połączenia.
Metody	<ul style="list-style-type: none"> • init() — rozpoczyna działanie serwera, wykonuje czynności związane z tworzeniem połączenia, między innymi blokuje możliwość lokalnego rysowania; • wywołaj(przesłane_dane) — interpretuje otrzymane dane i wywołuje odpowiednią metodę edytora graficznego.

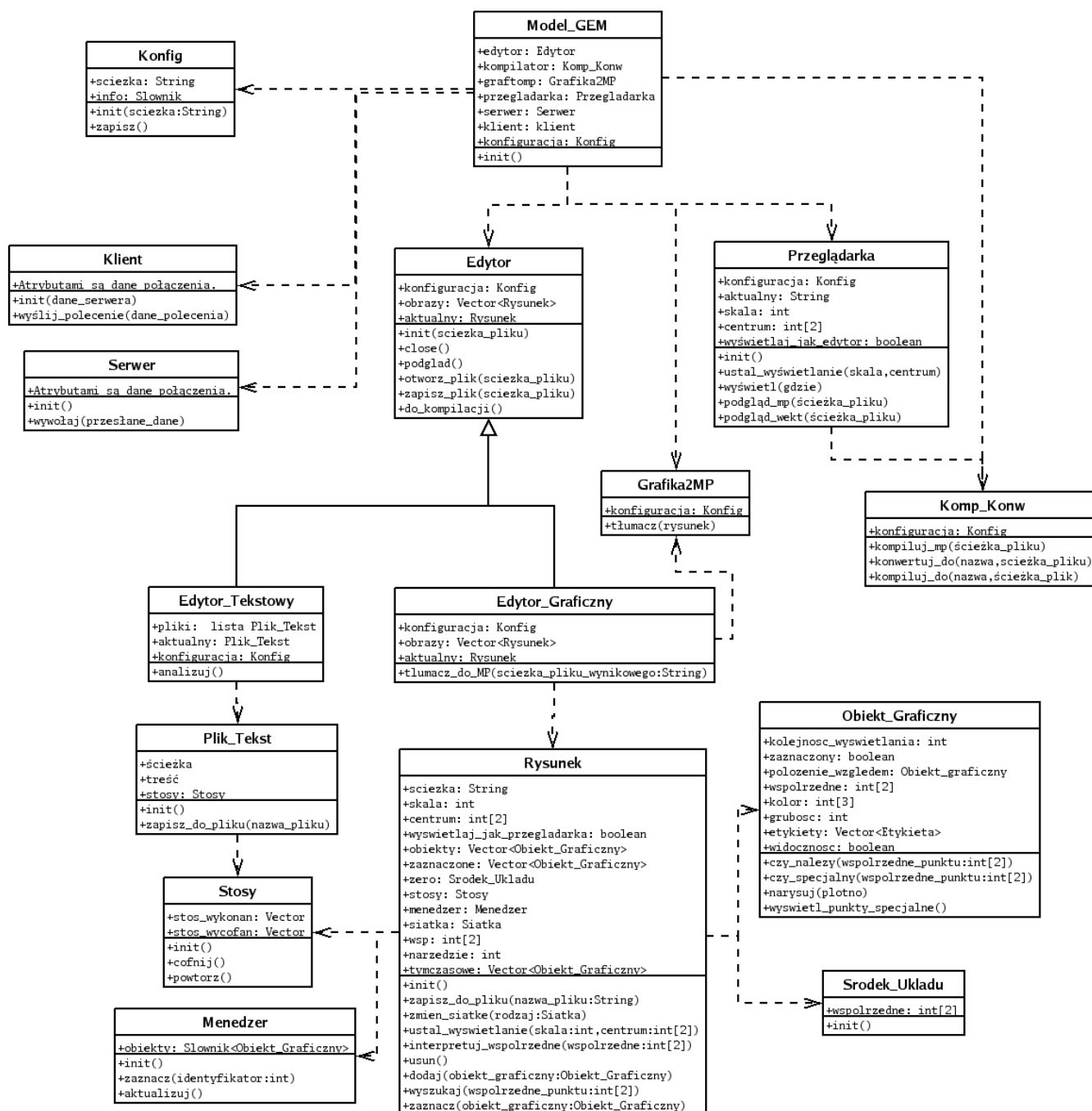
Klasa	Klient
Krótki opis	Obiekt tej klasy, po przyłączeniu się do połączenia internetowego, wysyła informacje o zmianach dokonywanych na tworzonym rysunku.
Atrybuty	Atrybutami są dane połączenia.

Metody	<ul style="list-style-type: none"> • init(dane_serwera) — podłącza się do serwera o podanych parametrach, aby odbierać informacje o zmianach w rysunku; • wyślij_polecenie(dane_polecenia) — wysyła przez sieć dane dokonywanych na rysunku modyfikacji.
---------------	--

Parametrami konfiguracyjnymi obiektów klas *Serwer* i *Klient* są domyślne dane połączenia.

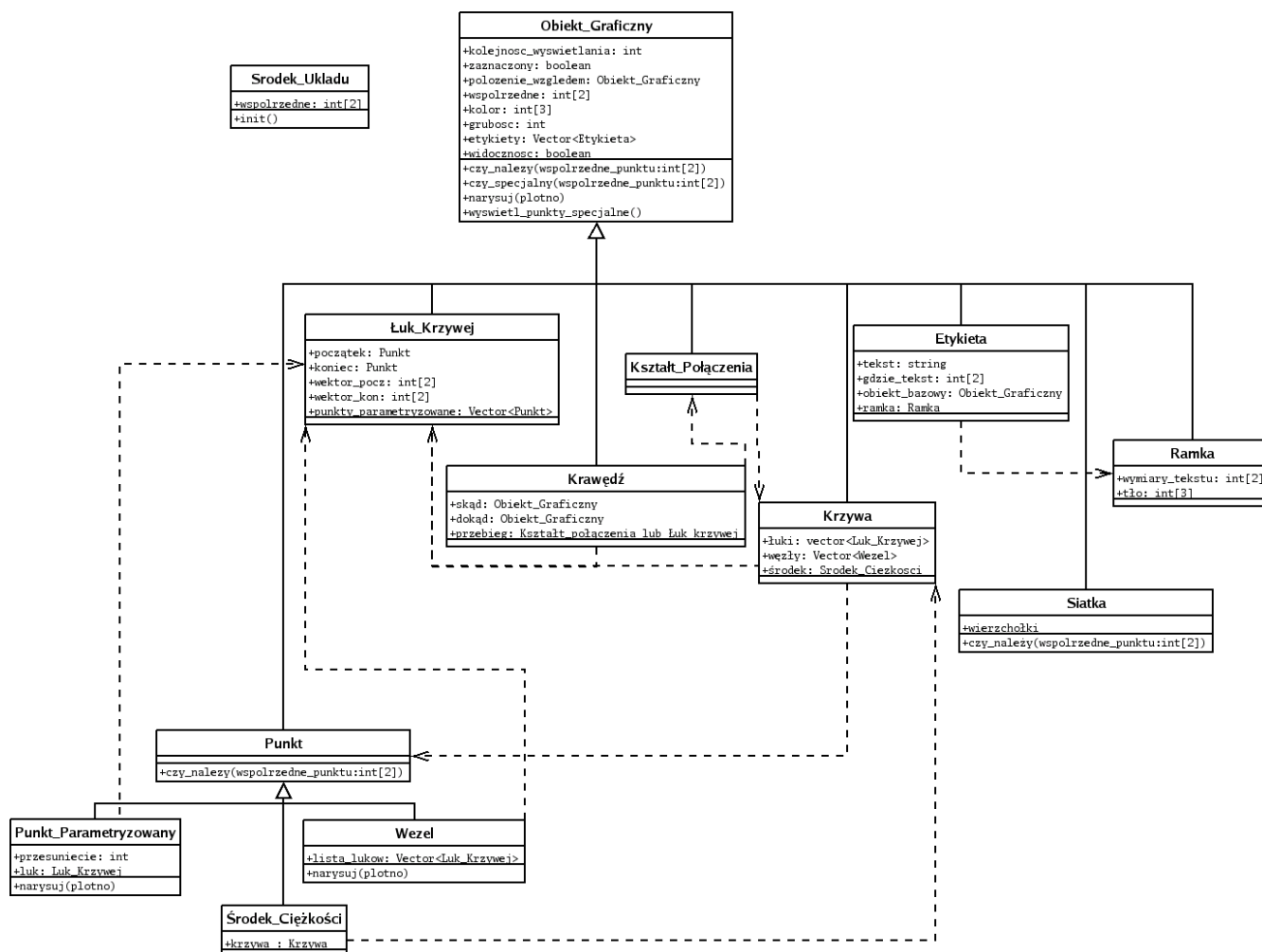
8.3.1. Diagram klas sterujących programem

8.3.1. Diagram klas sterujących programem



8.3.2. Diagram klas obsługi grafiki

Poniższy diagram przedstawia zależności pomiędzy klasami reprezentującymi obiekty graficzne rysunku.



9. Przechowywane dane

9.1. Przechowywanie rysunków

Aby użytkownik mógł pracować z edytorem graficznym, konieczne jest umożliwienie zapisania efektów pracy do pliku, wczytywania ich i kontynuacji pracy na tych danych. Zapisywanie danych do pliku będzie się odbywać poprzez serializację obiektów danego rysunku — przechowywanych przez atrybut *obiekty* obiektu klasy *Rysunek* (będący listą obiektów graficznych). Wówczas wczytanie danych z pliku będzie tworzyło od razu listę obiektów na rysunku, a większość pozostałych elementów rysunku da się łatwo odtworzyć z tej listy. Stosy wycofań nie będą przechowywane.

Jeśli serializacja obiektów będzie stwarzać problemy podczas implementacji systemu, Zespół opracuje własny format zapisu danych rysunku do pliku.

9.2. Pliki konfiguracyjne programu

Pomiędzy kolejnymi uruchomieniami programu trzeba przechowywać dane konfiguracyjne całości programu oraz jego poszczególnych części. Dla każdej części programu, która wymaga pliku konfiguracyjnego oraz dla całości programu będą istniały oddzielne pliki konfiguracyjne, umieszczane w odpowiednim miejscu tak, aby program mógł je odnaleźć (szczegóły zależą od używanego systemu operacyjnego).

Pliki konfiguracyjne będą miały prosty i czytelny format tekstowy, którego pełna dokumentacja zostanie zawarta w dokumentacji programu (aczkolwiek użytkownik nie będzie musiał go znać — wszystkie dane będzie można modyfikować bezpośrednio w programie). Tutaj podane są jedynie ogólne reguły konstruowania tych plików:

9.2.1. Reguły tworzenia

1. Wiersz rozpoczyna się od identyfikatora — ustalonej nazwy danego parametru, bez białych znaków. Pełen słownik identyfikatorów i odpowiadających im sposobów zapisu wartości zostanie podany w dokumentacji.
2. Po identyfikatorze pola następuje ciąg białych znaków (najlepiej jedna spacja).
3. Dalej, w zależności od rodzaju pola, którego wartość ma być podana w danym wierszu, wczytywane są kolejne słowa, liczby lub pojedyncze znaki, zawsze oddzielone białymi znakami. Stąd wynika, że białe znaki nie mogą być częścią słowa podawanego jako parametr konfiguracji.
4. Jeśli program wczyta z danego wiersza wszystkie potrzebne mu parametry (i nie będzie ich za dużo), przechodzi do następnego wiersza. Jeśli natomiast w wierszu jest zła liczba parametrów, albo ich wartości nie pochodzą z zakresu obsługiwanego przez program (w to wliczany jest również przypadek, kiedy zamiast liczby pojawia się słowo), program nie uwzględnia odczytanych z tego wiersza przed błędem wartości i odpowiadający wierszowi parametr ustawia na pewną wartość domyślną.
5. Jeśli dany identyfikator pojawia się w pliku więcej niż raz, parametr jest wczytywany za każdym jego pojawieniem się, ma więc wartość pochodzącą z ostatniego wiersza z tym identyfikatorem.
6. Jeśli wiersz nie rozpoczyna się od poprawnego identyfikatora, program nie czyta go dalej i przechodzi do kolejnego wiersza.
7. Znak „%” zostaje uznany za znak rozpoczynający komentarz ciągnący się do końca wiersza, od miejsca pojawienia się znaku. Wobec tego znak „%” nie może występować jako część słowa podawanego jako parametr konfiguracji.

9.2.2. Przykład

Poniżej podany jest przykładowy plik konfiguracyjny edytora graficznego (aczkolwiek zazwyczaj będzie w nim więcej danych, ponieważ edytor w tworzonych przez siebie plikach konfiguracyjnych zapisuje wartości wszystkich parametrów, nawet jeśli są wartościami domyślnymi).

```
% Konfiguracja edytora graficznego:
TextColorRGB 123 132 231      % kolor czcionki w formacie RGB
FontSize 12
LabelPositioning NE          % etykieta ma być przesunięta w prawo i w górę
```

10. Wydajność systemu

Program GEM został zaprojektowany z myślą o tworzeniu rysunków o niezbyt wysokim poziomie skomplikowania. Przy spodziewanych kilkudziesięciu obiektach, z których składa się rysunek, powinien działać zadowalająco szybko na średniej klasy komputerach domowych, mających kilka lat.

Wydajność programu będzie zależała przede wszystkim od stopnia złożoności rysunku. Jednak nawet bardzo skomplikowane rysunki nie będą uniemożliwiały pracy programu.

W module sieciowym przepływ informacji przez sieć będzie niewielki, nie należy się więc spodziewać nadmiernego obciążenia sieci.

11. Jakość

11.1. Przenośność

Ze względu na to, że program powstaje w środowisku Java, wydaje się, że nie powinny występować problemy z przenoszeniem programu na inne komputery wyposażone w maszynę wirtualną.

11.2. Możliwości rozbudowywania systemu

System jest podzielony na wiele modułów udostępniających pewne interfejsy, co pozwala na ich łatwą podmianę poprzez moduły udostępniające taki sam lub bogatszy interfejs.

System jest projektowany jako narzędzie, które w przyszłości będzie ulegać zmianom polegającym na dodawaniu kolejnych funkcjonalności celem zaspokajania potrzeb coraz szerszej gamy użytkowników. W związku z tym już obecnie projektowane fragmenty są tworzone z myślą również o przyszłych wyzwaniach, a twórcy poszukują ogólnych mechanizmów, które znajdą zastosowanie również później.

Przykład: tworząc rysunek na kartce papieru malujemy wszystkie obiekty w pewnej kolejności. Kolejność ta wiąże się również z pewnymi zależnościami pomiędzy kawałkami rysunku. Jeśli chcemy narysować prostą prostopadłą do innej prostej i przechodzącą przez dany punkt, to rysując ją, musimy uwzględnić położenie zarówno punktu, jak i prostej. Czego możemy wymagać w takim wypadku od programu graficznego? Na przykład tego, by zmiana położenia punktu pociągała za sobą zmianę położenia prostej prostopadłej. W pierwotnej wersji myślimy o umożliwieniu definiowania stałego przesunięcia jednego obiektu względem drugiego (głównie ze względu na wiązanie etykiet z obiektami graficznymi). Jeśli jednak powstanie pakiet do tworzenia konstrukcji geometrycznych (cyrklem i

linijką), to zależności mogą się bardzo skomplikować, toteż już teraz planujemy stworzenie mechanizmów pozwalających na weryfikację zależności od wielu obiektów.

Pozwoli to w przyszłości dodawać nowe pakiety bez potrzeby poprawiania wielu metod, a jedynie dołożenia w wskazanych miejscach informacji o nowych narzędziach i zaimplementowaniu ich obsługi.

11.3. Bezpieczeństwo

Kwestia bezpieczeństwa jest istotna wyłącznie dla pracy z aplikacją sieciową. Ponieważ jednak nie będą przesyłane żadne dane wymagające szczególnej ochrony, w pierwszej wersji programu problem bezpieczeństwa przesyłanych danych nie jest szczególnie uwzględniany.

12. Historia zmian

Wersja	Data	Autorzy zmian	Zmiany
0.5	3.01.2007	cały Zespół	pierwotna, niepełna wersja
0.6	4.01.2007	cały Zespół	dodane brakujące fragmenty
0.7	5.01.2007	Marysia Donten	poprawiony opis hierarchii klas
0.8	5.01.2007	Marek Grabowski	poprawione początkowe rozdziały oraz opis aplikacji sieciowej
0.9	5.01.2007	Kuba Pochrybniak	dalsze poprawki
1.0	6.01.2007	Kuba Pochrybniak, Marysia Donten	ostateczne poprawki merytoryczne, stylistyczne i językowe
1.1	18.01.2007	Kuba Pochrybniak	poprawki naniesione po zajęciach
1.2	18.01.2007	Marysia Donten	opracowane diagramy klas